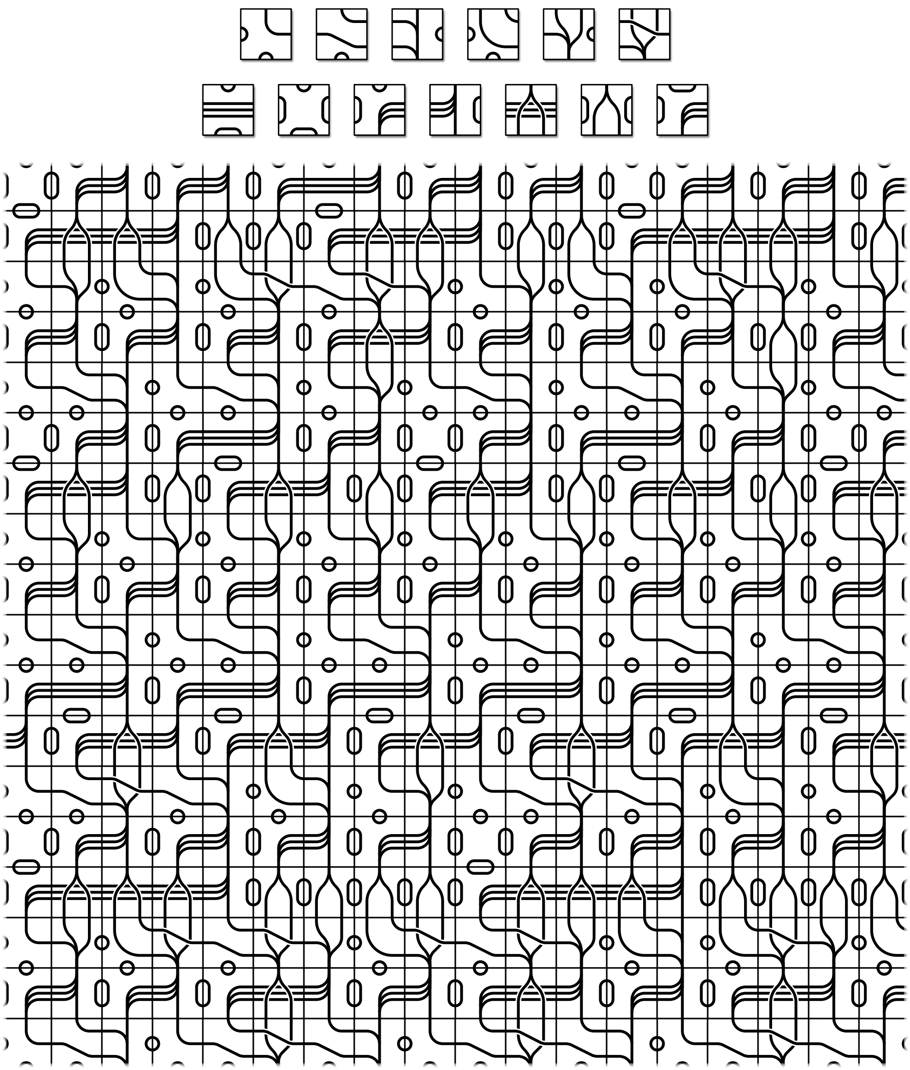
Models of Computation

Jeff Erickson



January 4, 2015

I'm writing a book.
I've got the page numbers done,
so now I just have to fill in the rest.

— Stephen Wright

About These Notes

These are lecture notes that I wrote for the course "Algorithms and Models of Computation" at the University of Illinois, Urbana-Champaign for the first time in Fall 2014. This course is a broad introduction to theoretical computer science, aimed at third-year computer science and computer engineering majors, that covers both fundamental topics in algorithms, for which I already have copious notes, and fundamental topics on formal languages and automata, for which I wrote the notes you are reading now.

The most recent revision of these notes (or nearly so) is available online at http://www.cs.illinois.edu/~jeffe/teaching/algorithms/, along with my algorithms notes and a near-complete archive of past homeworks and exams from all my theoretical computer science classes. I plan to revise and reorganize these whenever I teach this material, so you may find more recent versions on the web page of whatever course I am currently teaching.

About the Exercises

Each note ends with several exercises, many of which I used in homeworks, discussion sections, or exams. *Stars indicate more challenging problems (which I have *not* used in homeworks, discussion sections, or exams). Many of these exercises were contributed by my amazing teaching assistants:

> Alex Steiger, Chao Xu, Connor Clark, Gail Steitz, Grant Czajkowski, Hsien-Chih Chang, Junqing Deng, Nick Bachmair, and Tana Wattanawaroon

Please do not ask me for solutions to the exercises. If you are a student, seeing the solution will rob you of the experience of solving the problem yourself, which is the only way to learn the material. If you are an instructor, you shouldn't ask your students to solve problems that you can't solve yourself. (I don't always follow my own advice, so I'm sure some of the problems are buggy.)

Caveat Lector!

These notes are best viewed as an unfinished first draft. You should assume the notes contain several major errors, in addition to the usual unending supply of typos, fencepost errors, off-by-one errors, and brain farts. Before Fall 2014, I had not taught this material in more than two decades. Moreover, the course itself is still very new—Lenny Pitt and I developed the course and offered the first pilot in Spring 2014 (with Lenny presenting the formal language material)—so even the choice of which material to emphasize, sketch, or exclude is still very much in flux.

I would sincerely appreciate feedback of any kind, especially bug reports.

Thanks, and enjoy!

— *Jeff*

Contents

THOMAS GODFREY, a self-taught mathematician, great in his way, and afterward inventor of what is now called Hadley's Quadrant. But he knew little out of his way, and was not a pleasing companion; as, like most great mathematicians I have met with, he expected universal precision in everything said, or was forever denying or distinguishing upon trifles, to the disturbance of all conversation. He soon left us.

— Benjamin Franklin, *Memoirs, Part 1* (1771)
describing one of the founding members of the Junto

I hope the reader sees that the alphabet can be understood by any intelligent being who has any one of the five senses left him,—by all rational men, that is, excepting the few eyeless deaf persons who have lost both taste and smell in some complete paralysis. . . . Whales in the sea can telegraph as well as senators on land, if they will only note the difference between long spoutings and short ones. . . . A tired listener at church, by properly varying his long yawns and his short ones, may express his opinion of the sermon to the opposite gallery before the sermon is done.

— Edward Everett Hale, "The Dot and Line Alphabet", *Altlantic Monthy* (October 1858)

If indeed, as Hilbert asserted, mathematics is a meaningless game played with meaningless marks on paper, the only mathematical experience to which we can refer is the making of marks on paper.

— Eric Temple Bell, *The Queen of the Sciences* (1931)

1 Strings

Throughout this course, we will discuss dozens of algorithms and computational models that manipulate sequences: one-dimensional arrays, linked lists, blocks of text, walks in graphs, sequences of executed instructions, and so on. Ultimately the input and output of any algorithm must be representable as a finite string of symbols—the raw contents of some contiguous portion of the computer's memory. Reasoning about computation requires reasoning about strings.

This note lists several formal definitions and formal induction proofs related to strings. These definitions and proofs are *intentionally* much more detailed than normally used in practice—most people's intuition about strings is fairly accurate—but the extra precision is necessary for any sort of formal proof. It may be helpful to think of this material as part of the "assembly language" of theoretical computer science. We normally think about computation at a *much* higher level of abstraction, but ultimately every argument must "compile" down to these (and similar) definitions.

1.1 Definitions

Fix an arbitrary finite set $\boldsymbol{\Sigma}$ called the ***alphabet***; the elements of Σ are called ***symbols*** or ***characters***. As a notational convention, I will always use lower-case letters near the start of the English alphabet ($a, b, c, \ldots$) as symbol variables, and *never* as explicit symbols. For explicit symbols, I will always use fixed-width upper-case letters (`A`, `B`, `C`, . . .), digits (`0`, `1`, `2`, . . .), or other symbols (◇, `$`, `#`, •, . . .) that are clearly distinguishable from variables.

A ***string*** (or ***word***) over Σ is a finite sequence of zero or more symbols from Σ. Formally, a string w over Σ is defined recursively as either

- the empty string, denoted by the Greek letter $\boldsymbol{\varepsilon}$ (epsilon),or
- an ordered pair $\boldsymbol{(a, x)}$, where a is a symbol in Σ and x is a string over Σ.

We normally write either $\boldsymbol{a \cdot x}$ or simply $\boldsymbol{ax}$ to denote the ordered pair (a, x). Similarly, we normally write explicit strings as sequences of symbols instead of nested ordered pairs; for example, STRING is convenient shorthand for the formal expression $(\mathtt{S}, (\mathtt{T}, (\mathtt{R}, (\mathtt{I}, (\mathtt{N}, (\mathtt{G}, \varepsilon))))))$. As a notational convention, I will always use lower-case letters near the end of the alphabet $(\ldots, w, x, y, z)$ to represent unknown strings, and SHOUTY◇MONOSPACED◇TEXT to represent explicit symbols and (non-empty) strings.

The set of all strings over Σ is denoted $\boldsymbol{\Sigma^*}$ (pronounced "sigma star"). It is very important to remember that every element of Σ^* is a *finite* string, although Σ^* itself is an infinite set containing strings of every possible *finite* length.

The ***length*** $\boldsymbol{|w|}$ of a string w is the number of symbols in w, defined formally as follows:

$$|w| := \begin{cases} 0 & \text{if } w = \varepsilon, \\ 1 + |x| & \text{if } w = ax. \end{cases}$$

For example, the string SEVEN has length 5. Although they are formally different objects, we do not normally distinguish between symbols and strings of length 1.

The ***concatenation*** of two strings x and y, denoted either $\boldsymbol{x \bullet y}$ or simply $\boldsymbol{xy}$, is the unique string containing the characters of x in order, followed by the characters in y in order. For example, the string NOWHERE is the concatenation of the strings NOW and HERE; that is, NOW • HERE = NOWHERE. (On the other hand, HERE • NOW = HERENOW.) Formally, concatenation is defined recusively as follows:

$$w \bullet z := \begin{cases} z & \text{if } w = \varepsilon \\ a \cdot (x \bullet z) & \text{if } w = ax \end{cases}$$

(Here I'm using a larger dot • to formally distinguish the operator that concatenates two arbitrary strings from from the operator · that builds a string from a single character and a string.)

When we describe the concatenation of more than two strings, we normally omit all dots and parentheses, writing $wxyz$ instead of $(w \bullet (x \bullet y)) \bullet z$, for example. This simplification is justified by the fact (which we will prove shortly) that • is associative.

1.2 Induction on Strings

Induction is *the* standard technique for proving statements about recursively defined objects. Hopefully you are already comfortable proving statements about *natural numbers* via induction, but induction actually a far more general technique. Several different variants of induction can be used to prove statements about more general structures; here I describe the variant that I recommend (and actually use in practice). This variant follows two primary design considerations:

- ***The case structure of the proof should mirror the case structure of the recursive definition.*** For example, if you are proving something about all *strings*, your proof should have two cases: Either $w = \varepsilon$, or $w = ax$ for some symbol a and string x.

- ***The inductive hypothesis should be as strong as possible.*** The (strong) inductive hypothesis for statements about natural numbers is *always* "Assume there is no counterexample k such that $k < n$." I recommend adopting a similar inductive hypothesis for strings: "Assume there is no counterexample x such that $|x| < |w|$." Then for the case $w = ax$, we have $|x| = |w| - 1 < |w|$ by definition of $|w|$, so the inductive hypothesis applies to x.

Thus, string-induction proofs have the following boilerplate structure. Suppose we want to prove that every string is perfectly cromulent, whatever that means. The white boxes hide additional proof details that, among other things, depend on the precise definition of "perfectly cromulent".

> **Proof:** Let w be an arbitrary string.
> Assume, for every string x such that $|x| < |w|$, that x is perfectly cromulent.
> There are two cases to consider.
>
> - Suppose $w = \varepsilon$.
>
>   Therefore, w is perfectly cromulent.
> - Suppose $w = ax$ for some symbol a and string x.
>   The induction hypothesis implies that x is perfectly cromulent.
>
>   Therefore, w is perfectly cromulent.
>
> In both cases, we conclude that w is perfectly cromulent. □

Here are three canonical examples of this proof structure. When developing proofs in this style, I strongly recommend first ***mindlessly*** writing the green text (the boilerplate) with lots of space for each case, then filling in the red text (the actual theorem and the induction hypothesis), and only then starting to actually think.

Lemma 1.1. *For every string w, we have $w \bullet \varepsilon = w$.*

Proof: Let w be an arbitrary string. Assume that $x \bullet \varepsilon = x$ for every string x such that $|x| < |w|$. There are two cases to consider:

- Suppose $w = \varepsilon$.

$$\begin{aligned} w \bullet \varepsilon &= \varepsilon \bullet \varepsilon && \text{because } w = \varepsilon, \\ &= \varepsilon && \text{by definition of concatenation,} \\ &= w && \text{because } w = \varepsilon. \end{aligned}$$

- Suppose $w = ax$ for some symbol a and string x.

$$\begin{aligned} w \bullet \varepsilon &= (a \cdot x) \bullet \varepsilon && \text{because } w = ax, \\ &= a \cdot (x \bullet \varepsilon) && \text{by definition of concatenation,} \\ &= a \cdot x && \text{by the inductive hypothesis,} \\ &= w && \text{because } w = ax. \end{aligned}$$

In both cases, we conclude that $w \bullet \varepsilon = w$. □

Lemma 1.2. *Concatenation adds length: $|w \bullet x| = |w| + |x|$ for all strings w and x.*

Proof: Let w and x be arbitrary strings. Assume that $|y \bullet x| = |y| + |x|$ for every string y such that $|y| < |w|$. (Notice that we are using induction only on w, not on x.) There are two cases to consider:

- Suppose $w = \varepsilon$.

$$\begin{aligned} |w \bullet x| &= |\varepsilon \bullet x| && \text{because } w = \varepsilon \\ &= |x| && \text{by definition of } |\,| \\ &= |\varepsilon| + |x| && |e| = 0 \text{ by definition of } |\,| \\ &= |w| + |x| && \text{because } w = \varepsilon \end{aligned}$$

- Suppose $w = ay$ for some symbol a and string y.

$$\begin{aligned} |w \bullet x| &= |ay \bullet x| && \text{because } w = ay \\ &= |a \cdot (y \bullet x)| && \text{by definition of } \bullet \\ &= 1 + |y \bullet x| && \text{by definition of } |\,| \\ &= 1 + |y| + |x| && \text{by the inductive hypothesis} \\ &= |ay| + |x| && \text{by definition of } |\,| \\ &= |w| + |x| && \text{because } w = ay \end{aligned}$$

In both cases, we conclude that $|w \bullet x| = |w| + |x|$. □

Lemma 1.3. *Concatenation is* associative*:* $(w \bullet x) \bullet y = w \bullet (x \bullet y)$ *for all strings* w*,* x*, and* y*.*

Proof: Let w, x, and y be arbitrary strings. Assume that $(z \bullet x) \bullet y = w \bullet (x \bullet y)$ for every string z such that $|z| < |w|$. (Again, we are using induction only on w.) There are two cases to consider.

- Suppose $w = \varepsilon$.

$$\begin{aligned} (w \bullet x) \bullet y &= (\varepsilon \bullet x) \bullet y && \text{because } w = \varepsilon \\ &= x \bullet y && \text{by definition of } \bullet \\ &= \varepsilon \bullet (x \bullet y) && \text{by definition of } \bullet \\ &= w \bullet (x \bullet y) && \text{because } w = \varepsilon \end{aligned}$$

- Suppose $w = az$ for some symbol a and some string z.

$$\begin{aligned} (w \bullet x) \bullet y &= (az \bullet x) \bullet y && \text{because } w = az \\ &= (a \cdot (z \bullet x)) \bullet y && \text{by definition of } \bullet \\ &= a \cdot ((z \bullet x) \bullet y) && \text{by definition of } \bullet \\ &= a \cdot (z \bullet (x \bullet y)) && \text{by the inductive hypothesis} \\ &= az \bullet (x \bullet y) && \text{by definition of } \bullet \\ &= w \bullet (x \bullet y) && \text{because } w = az \end{aligned}$$

In both cases, we conclude that $(w \bullet x) \bullet y = w \bullet (x \bullet y)$. □

This is not the only boilerplate that one can use for induction proofs on strings. For example, we can modify the inductive case analysis using the following observation: A *non-empty* string w is either a single symbol or the concatenation of two non-empty strings, which (by Lemma 1.2) must be shorter than w. Here is a proof of Lemma 1.3 that uses this alternative recursive structure:

Proof: Let w, x, and y be arbitrary strings. Assume that $(z \bullet x') \bullet y' = z \bullet (x' \bullet y')$ for all strings x', y', and z such that $|z| < |w|$. (We need a stronger induction hypothesis here than in the previous proofs!) There are ***three*** cases to consider.

- Suppose $w = \varepsilon$.

$$\begin{aligned}
(w \bullet x) \bullet y &= (\varepsilon \bullet x) \bullet y && \text{because } w = \varepsilon \\
&= x \bullet y && \text{by definition of } \bullet \\
&= \varepsilon \bullet (x \bullet y) && \text{by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = \varepsilon
\end{aligned}$$

- Suppose w is equal to some symbol a.

$$\begin{aligned}
(w \bullet x) \bullet y &= (a \bullet x) \bullet y && \text{because } w = a \\
&= (a \cdot x) \bullet y && \text{because } a \bullet z = a \cdot z \text{ by definition of } \bullet \\
&= a \cdot (x \bullet y) && \text{by definition of } \bullet \\
&= a \bullet (x \bullet y) && \text{because } a \bullet z = a \cdot z \text{ by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = a
\end{aligned}$$

- Suppose $w = uv$ for some nonempty strings u and v.

$$\begin{aligned}
(w \bullet x) \bullet y &= ((u \bullet v) \bullet x) \bullet y && \text{because } w = uv \\
&= (u \bullet (v \bullet x)) \bullet y && \text{by the inductive hypothesis, because } |u| < |w| \\
&= u \bullet ((v \bullet x) \bullet y) && \text{by the inductive hypothesis, because } |u| < |w| \\
&= u \bullet (v \bullet (x \bullet y)) && \text{by the inductive hypothesis, because } |v| < |w| \\
&= (u \bullet v) \bullet (x \bullet y) && \text{by the inductive hypothesis, because } |u| < |w| \\
&= w \bullet (x \bullet y) && \text{because } w = uv
\end{aligned}$$

In both cases, we conclude that $(w \bullet x) \bullet y = w \bullet (x \bullet y)$. □

1.3 Indices, Substrings, and Subsequences

For any string w and any integer $1 \le i \le |w|$, the expression $\boldsymbol{w_i}$ denotes the ith symbol in w, counting from left to right. More formally, w_i is recursively defined as follows:

$$w_i := \begin{cases} a & \text{if } w = ax \text{ and } i = 1 \\ x_{i-1} & \text{if } w = ax \text{ and } i > 1 \end{cases}$$

As one might reasonably expect, w_i is formally undefined if $i < 1$ or $w = \varepsilon$, and therefore (by induction) if $i > |w|$. The integer i is called the ***index*** of w_i.

We sometimes write strings as a concatenation of their constituent symbols using this subscript notation: $w = w_1 w_2 \cdots w_{|w|}$. While standard, this notation is slightly misleading, since it *incorrectly* suggests that the string w contains at least three symbols, when in fact w could be a single symbol or even the empty string.

In actual code, subscripts are usually expressed using the bracket notation $\boldsymbol{w[i]}$. Brackets were introduced as a typographical convention over a hundred years ago because subscripts and superscripts[1] were difficult or impossible to type.[2] We sometimes write strings as explicit arrays $w[1\,..\,n]$, with the understanding that $n = |w|$. Again, this notation is potentially misleading; always remember that n might be zero; the string/array could be empty.

A ***substring*** of a string w is another string obtained from w by deleting zero or more symbols from the beginning and from the end. Formally, a string y is a substring of w if and only if there are strings x and z such that $w = xyz$. Extending the array notation for strings, we write $\boldsymbol{w[i\,..\,j]}$ to denote the substring of w starting at w_i and ending at w_j. More formally, we define

$$w[i\,..\,j] := \begin{cases} \varepsilon & \text{if } j < i, \\ w_i \cdot w[i+1\,..\,j] & \text{otherwise.} \end{cases}$$

A ***proper substring*** of w is any substring other than w itself. For example, `LAUGH` is a proper substring of `SLAUGHTER`. Whenever y is a (proper) substring of w, we also call w a (proper) ***superstring*** of y.

A ***prefix*** of $w[1\,..\,n]$ is any substring of the form $w[1\,..\,j]$. Equivalently, a string p is a ***prefix*** of another string w if and only if there is a string x such that $px = w$. A ***proper prefix*** of w is any prefix except w itself. For example, `DIE` is a proper prefix of `DIET`.

Similarly, a suffix of $w[1\,..\,n]$ is any substring of the form $w[i\,..\,n]$. Equivalently, a string s is a ***suffix*** of a string w if and only if there is a string x such that $xs = w$. A ***proper suffix*** of w is any suffix except w itself. For example, `YES` is a proper suffix of `EYES`, and `HE` is both a proper prefix and a proper suffix of `HEADACHE`.

A ***subsequence*** of a string w is a strong obtained by deleting zero or more symbols from *anywhere* in w. More formally, z is a subsequence of w if and only if

- $z = \varepsilon$, or
- $w = ax$ for some symbol a and some string x such that z is a subsequence of x.
- $w = ax$ and $z = ay$ for some symbol a and some strings x and y, and y is a subsequence of x.

A ***proper subsequence*** of w is any subsequence of w other than w itself. Whenever z is a (proper) subsequence of w, we also call w a (proper) ***supersequence*** of z.

---

[1]The same bracket notation is also used for bibliographic references, instead of the traditional footnote/endnote superscripts, for exactly the same reasons.

[2]A ***typewriter*** is an obsolete mechanical device loosely resembling a computer keyboard. Pressing a key on a typewriter moves a lever (called a "typebar") that strikes a cloth ribbon full of ink against a piece of paper, leaving the image of a single character. Many historians believe that the ordering of letters on modern keyboards (`QWERTYUIOP`) evolved in the late 1800s, reaching its modern form on the 1874 Sholes & Glidden Type-Writer™, in part to separate many common letter pairs, to prevent typebars from jamming against each other; this is also why the keys on most modern keyboards are arranged in a slanted grid. (The common folk theory that the ordering was deliberately intended to slow down typists doesn't withstand careful scrutiny.) A more recent theory suggests that the ordering was influenced by telegraph[3] operators, who found older alphabetic arrangements confusing, in part because of ambiguities in American Morse Code.

[3]A ***telegraph*** is an obsolete electromechanical communication device consisting of an electrical circuit with a switch at one end and an electromagnet at the other. The sending operator would press and release a key, closing and opening the circuit, originally causing the electromagnet to push a stylus onto a moving paper tape, leaving marks that could be decoded by the receiving operator. (Operators quickly discovered that they could directly decode the clicking sounds made by the electromagnet, and so the paper tape became obsolete almost immediately.) The most common scheme within the US to encode symbols, developed by Alfred Vail and Samuel Morse in 1837, used (mostly) short (·) and long (—) marks—now called "dots" and "dashes", or "dits" and "dahs"—separated by gaps of various lengths. American Morse code (as it became known) was ambiguous; for example, the letter `Z` and the string `SE` were both encoded by the sequence ··· · ("di-di-dit, dit"). This ambiguity has been blamed for the `S` key's position on the

Substrings and subsequences are not the same objects; don't confuse them! Every substring of w is also a subsequence of w, but not every subsequence is a substring. For example, METAL is a subsequence, but not a substring, of MEATBALL. To emphasize the distinction, we sometimes redundantly refer to substrings of w as ***contiguous*** substrings, meaning all their symbols appear together in w.

Exercises

Most of the following exercises ask for proofs of various claims about strings. For each claim, give a complete, self-contained, formal proof by inductive definition-chasing, using the boilerplate structure recommended in Section 1.2. You can use Lemmas 1.1, 1.2, and 1.3, but don't assume any other facts about strings that you have not actually proved. Do not use the words "obvious" or "clearly" or "just". Most of these claims **are** in fact obvious; the real exercise is understanding **why** they're obvious.

1. For any symbol a and any string w, let $\boldsymbol{\#(a,w)}$ denote the number of occurrences of a in w. For example, $\#(\texttt{A},\texttt{BANANA}) = 3$ and $\#(\texttt{X},\texttt{FLIBBERTIGIBBET}) = 0$.

   (a) Give a formal recursive definition of the function $\# \colon \Sigma \times \Sigma^* \to \mathbb{N}$.

   (b) Prove that $\#(a, xy) = \#(a, x) + \#(a, y)$ for every symbol a and all strings x and y. Your proof must rely on both your answer to part (a) and the formal recursive definition of string concatenation.

2. Recursively define a set L of strings over the alphabet $\{\texttt{0},\texttt{1}\}$ as follows:

   - The empty string ε is in L.
   - For any two strings x and y in L, the string $\texttt{0}x\texttt{1}y\texttt{0}$ is also in L.
   - These are the only strings in L.

   (a) Prove that the string 000010101010010100 is in L.

   (b) Prove by induction that every string in L has exactly twice as many 0s as 1s. (You may assume the identity $\#(a, xy) = \#(a, x) + \#(a, y)$ for any symbol a and any strings x and y; see Exercise 1(b).)

   (c) Give an example of a string with exactly twice as many 0s as 1s that is *not* in L.

3. For any string w and any non-negative integer n, let $\boldsymbol{w^n}$ denote the string obtained by concatenating n copies of w; more formally, we define

$$w^n := \begin{cases} \varepsilon & \text{if } n = 0 \\ w \bullet w^{n-1} & \text{otherwise} \end{cases}$$

   For example, $(\texttt{BLAH})^5 = \texttt{BLAHBLAHBLAHBLAHBLAH}$ and $\varepsilon^{374} = \varepsilon$.

   Prove that $w^m \bullet w^n = w^{m+n}$ for every string w and all integers non-negative integer n and m.

---

typewriter keyboard near E and Z.

Vail and Morse were of course not the first people to propose encoding symbols as strings of bits. That honor apparently falls to Francis Bacon, who devised a five-bit binary encoding of the alphabet (except for the letters J and U) in 1605 as the basis for a steganographic code—a method or hiding secret message in otherwise normal text.

4. Let w be an arbitrary string, and let $n = |w|$. Prove each of the following statements.
   (a) w has exactly $n+1$ prefixes.
   (b) w has exactly n proper suffixes.
   (c) w has at most $n(n+1)/2$ distinct substrings.
   (d) w has at most $2^n - 1$ proper subsequences.

5. The ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:
$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$
   (a) Prove that $|w^R| = |w|$ for every string w.
   (b) Prove that $(wx)^R = x^R w^R$ for all strings w and x.
   (c) Prove that $(w^R)^n = (w^n)^R$ for every string w and every integer $n \geq 0$. (See Exercise 1.)
   (d) Prove that $(w^R)^R = w$ for every string w.

6. Let w be an arbitrary string, and let $n = |w|$. Prove the following statements for all indices $1 \leq i \leq j \leq k \leq n$.
   (a) $|w[i\,..\,j]| = j - i + 1$
   (b) $w[i\,..\,j] \bullet w[j+1\,..\,k] = w[i\,..\,k]$
   (c) $w^R[i\,..\,j] = (w[i'\,..\,j'])^R$ where $i' = |w| + 1 - j$ and $j' = |w| + 1 - i$.

7. A ***palindrome*** is a string that is equal to its reversal.
   (a) Give a recursive definition of a palindrome over the alphabet Σ.
   (b) Prove that any string p meets your recursive definition of a palindrome if and only if $p = p^R$.

8. A string $w \in \Sigma^*$ is called a ***shuffle*** of two strings $x, y \in \Sigma^*$ if at least one of the following recursive conditions is satisfied:
   - $w = x = y = \varepsilon$.
   - $w = aw'$ and $x = ax'$ and w' is a shuffle of x' and y, for some $a \in \Sigma$ and some $w', x' \in \Sigma^*$.
   - $w = aw'$ and $y = ay'$ and w' is a shuffle of x and y', for some $a \in \Sigma$ and some $w', y' \in \Sigma^*$.

   For example, the string BANANANANASA is a shuffle of the strings BANANA and ANANAS.

   (a) Prove that if w is a shuffle of x and y, then $|w| = |x| + |y|$.
   (b) Prove that if w is a shuffle of x and y, then w^R is a shuffle of x^R and y^R.

9. Consider the following pair of mutually recursive functions on strings:

$$evens(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ odds(x) & \text{if } w = ax \end{cases} \qquad odds(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ a \cdot evens(x) & \text{if } w = ax \end{cases}$$

   (a) Prove the following identity for all strings w and x:

$$evens(w \bullet x) = \begin{cases} evens(w) \bullet evens(x) & \text{if } |w| \text{ is even,} \\ evens(w) \bullet odds(x) & \text{if } |w| \text{ is odd.} \end{cases}$$

   (b) State and prove a similar identity for $odds(w \bullet x)$.

10. For any positive integer n, the ***Fibonacci string*** $\boldsymbol{F_n}$ is defined recursively as follows:

$$F_n = \begin{cases} \mathtt{0} & \text{if } n = 1, \\ \mathtt{1} & \text{if } n = 2, \\ F_{n-2} \bullet F_{n-1} & \text{otherwise.} \end{cases}$$

For example, $F_6 = \mathtt{10101101}$ and $F_7 = \mathtt{0110110101101}$.

   (a) Prove that for every integer $n \geq 2$, the string F_n can also be obtained from F_{n-1} by replacing every occurrence of `0` with `1` and replacing every occurrence of `1` with `01`. More formally, prove that $F_n = Finc(F_{n-1})$, where

$$Finc(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \mathtt{1} \cdot Finc(x) & \text{if } w = \mathtt{0}x \\ \mathtt{01} \bullet Finc(x) & \text{if } w = \mathtt{1}x \end{cases}$$

   [Hint: First prove that $\mathrm{Finc}(x \bullet y) = \mathrm{Finc}(x) \bullet \mathrm{Finc}(y)$*.]*

   (b) Prove that `00` and `111` are not substrings of any Fibonacci string F_n.

11. Prove that the following three properties of strings are in fact identical.

   - A string $w \in \{\mathtt{0}, \mathtt{1}\}^*$ is ***balanced*** if it satisfies one of the following conditions:
     - $w = \varepsilon$,
     - $w = \mathtt{0}x\mathtt{1}$ for some balanced string x, or
     - $w = xy$ for some balanced strings x and y.
   - A string $w \in \{\mathtt{0}, \mathtt{1}\}^*$ is ***erasable*** if it satisfies one of the following conditions:
     - $w = \varepsilon$, or
     - $w = x\mathtt{01}y$ for some strings x and y such that xy is erasable. (The strings x and y are not necessarily erasable.)
   - A string $w \in \{\mathtt{0}, \mathtt{1}\}^*$ is ***conservative*** if it satisfies ***both*** of the following conditions:
     - w has an equal number of `0`s and `1`s, and
     - no prefix of w has more `0`s than `1`s.

   (a) Prove that every balanced string is erasable.

(b) Prove that every erasable string is conservative.

(c) Prove that every conservative string is balanced.

[Hint: To develop intuition, it may be helpful to think of `0`*s as left brackets and* `1`*s as right brackets, but* ***don't*** *invoke this intuition in your proofs.]*

12. A string $w \in \{\texttt{0}, \texttt{1}\}^*$ ***equitable*** if it has an equal number of `0`s and `1`s.

(a) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

- $w = \varepsilon$,
- $w = \texttt{0}x\texttt{1}$ for some equitable string x,
- $w = \texttt{1}x\texttt{0}$ for some equitable string x, or
- $w = xy$ for some equitable strings x and y.

(b) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

- $w = \varepsilon$,
- $w = x\texttt{01}y$ for some strings x and y such that xy is equitable, or
- $w = x\texttt{10}y$ for some strings x and y such that xy is equitable.

In the last two cases, the individual strings x and y are not necessarily equitable.

(c) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

- $w = \varepsilon$,
- $w = xy$ for some balanced string x and some equitable string y, or
- $w = x^R y$ for some for some balanced string x and some equitable string y.

(See the previous exercise for the definition of "balanced".)

Caveat lector: This is the first edition of this lecture note. Please send bug reports and suggestions to jeffe@illinois.edu.

But the Lord came down to see the city and the tower the people were building. The Lord said, "If as one people speaking the same language they have begun to do this, then nothing they plan to do will be impossible for them. Come, let us go down and confuse their language so they will not understand each other."

— Genesis 11:6–7 (New International Version)

Soyez réglé dans votre vie et ordinaire comme un bourgeois,
afin d'être violent et original dans vos œuvres.

[Be regular and orderly in your life like a bourgeois,
so that you may be violent and original in your work.]

— Gustave Flaubert, in a letter to Gertrude Tennant (December 25, 1876)

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.

— Jamie Zawinski, alt.religion.emacs (August 12, 1997)

I define UNIX as 30 definitions of regular expressions living under one roof.

— Donald Knuth, *Digital Typography* (1999)

2 Regular Languages

2.1 Languages

A ***formal language*** (or just a ***language***) is a set of strings over some finite alphabet Σ, or equivalently, an arbitrary subset of Σ^*. For example, each of the following sets is a language:

- The empty set $\varnothing$.[1]
- The set $\{\varepsilon\}$.
- The set $\{\mathtt{0},\mathtt{1}\}^*$.
- The set $\{\mathtt{THE},\mathtt{OXFORD},\mathtt{ENGLISH},\mathtt{DICTIONARY}\}$.
- The set of all subsequences of THE◇OXFORD◇ENGLISH◇DICTIONARY.
- The set of all words in *The Oxford English Dictionary*.
- The set of all strings in $\{\mathtt{0},\mathtt{1}\}^*$ with an odd number of 1s.
- The set of all strings in $\{\mathtt{0},\mathtt{1}\}^*$ that represent a prime number in base 13.
- The set of all sequences of turns that solve the Rubik's cube (starting in some fixed configuration)
- The set of all python programs that print "Hello World!"

As a notational convention, I will always use italic upper-case letters (usually L, but also A, B, C, and so on) to represent languages.

[1]The empty set symbol $\varnothing$ derives from the Norwegian letter Ø, pronounced like a sound of disgust or a German ö, and *not* from the Greek letter ϕ. Calling the empty set "fie" or "fee" makes the baby Jesus cry.

Formal languages are not "languages" in the same sense that English, Klingon, and Python are "languages". Strings in a formal language do not necessarily carry any "meaning", nor are they necessarily assembled into larger units ("sentences" or "paragraphs" or "packages") according to some "grammar".

It is ***very*** important to distinguish between three "empty" objects. Many beginning students have trouble keeping these straight.

- $\varnothing$ is the empty *language*, which is a set containing zero strings. $\varnothing$ is not a string.
- $\boldsymbol{\{\varepsilon\}}$ is a language containing exactly one string, which has length zero. $\{\varepsilon\}$ is not empty, and it is not a string.
- $\boldsymbol{\varepsilon}$ is the empty *string*, which is a sequence of length zero. ε is not a language.

2.2 Building Languages

Languages can be combined and manipulated just like any other sets. Thus, if A and B are languages over Σ, then their union $A \cup B$, intersection $A \cap B$, difference $A \setminus B$, and symmetric difference $A \oplus B$ are also languages over Σ, as is the complement $\overline{A} := \Sigma^* \setminus A$. However, there are two more useful operators that are specific to sets of *strings*.

The ***concatenation*** of two languages A and B, again denoted $\boldsymbol{A \bullet B}$ or just $\boldsymbol{AB}$, is the set of all strings obtained by concatenating an arbitrary string in A with an arbitrary string in B:

$$A \bullet B := \{xy \mid x \in A \text{ and } y \in B\}.$$

For example, if $A = \{\texttt{HOCUS}, \texttt{ABRACA}\}$ and $B = \{\texttt{POCUS}, \texttt{DABRA}\}$, then

$$A \bullet B = \{\texttt{HOCUSPOCUS}, \texttt{ABRACAPOCUS}, \texttt{HOCUSDABRA}, \texttt{ABRACADABRA}\}.$$

In particular, for every language A, we have

$$\varnothing \bullet A = A \bullet \varnothing = \varnothing \qquad \text{and} \qquad \{\varepsilon\} \bullet A = A \bullet \{\varepsilon\} = A.$$

The ***Kleene closure*** or ***Kleene star***[2] of a language L, denoted $\boldsymbol{L^*}$, is the set of all strings obtained by concatenating a sequence of zero or more strings from L. For example, $\{\texttt{0}, \texttt{11}\}^* = \{\varepsilon, \texttt{0}, \texttt{00}, \texttt{11}, \texttt{000}, \texttt{011}, \texttt{110}, \texttt{0000}, \texttt{0011}, \texttt{0110}, \texttt{1100}, \texttt{1111}, \texttt{00000}, \texttt{00011}, \texttt{00110}, \ldots, \texttt{011110011011}, \ldots\}$. More formally, L^* is defined recursively as the set of all strings w such that either

- $w = \varepsilon$, or
- $w = xy$, for some strings $x \in L$ and $y \in L^*$.

This definition immediately implies that

$$\varnothing^* = \{\varepsilon\}^* = \{\varepsilon\}.$$

For any other language L, the Kleene closure L^* is infinite and contains arbitrarily long (but *finite!*) strings. Equivalently, L^* can also be defined as the smallest superset of L that contains the empty string ε and is closed under concatenation (hence "closure"). The set of all strings Σ^* is, just as the notation suggests, the Kleene closure of the alphabet Σ (where each symbol is viewed as a string of length 1).

[2]after Stephen Kleene, who pronounced his last name "***clay***-knee", not "clean" or "cleanie" or "claynuh" or "dimaggio".

A useful variant of the Kleene closure operator is the ***Kleene plus***, defined as $L^+ := L \bullet L^*$. Thus, L^+ is the set of all strings obtained by concatenating a sequence of ***one*** or more strings from L.

The following identities, which we state here without (easy) proofs, are useful for designing, simplifying, and understanding languages.

Lemma 2.1. *The following identities hold for all languages A, B, and C:*
(a) $\varnothing A = A\varnothing = \varnothing$.
(b) $\varepsilon A = A\varepsilon = A$.
(c) $A + B = B + A$.
(d) $(A + B) + C = A + (B + C)$.
(e) $(AB)C = A(BC)$.
(f) $A(B + C) = AB + AC$.

Lemma 2.2. *The following identities hold for every language L:*
(a) $L^* = \varepsilon + L^+ = L^*L^* = (L + \varepsilon)^* = (L \setminus \varepsilon)^* = \varepsilon + L + L^+L^+$.
(b) $L^+ = L^* \setminus \varepsilon = LL^* = L^*L = L^+L^* = L^*L^+ = L + L^+L^+$.
(c) $L^+ = L^*$ *if and only if* $\varepsilon \in L$.

Lemma 2.3 (Arden's Rule). *For any languages A, B, and L such that $L = AL + B$, we have $A^*B \subseteq L$. Moreover, if A does not contain the empty string, then $L = AL + B$ if and only if $L = A^*B$.*

2.3 Regular Languages and Regular Expressions

A language L is ***regular*** if and only if it satisfies one of the following (recursive) conditions:

- L is empty;
- L contains a single string (which could be the empty string ε);
- L is the union of two regular languages;
- L is the concatenation of two regular languages; or
- L is the Kleene closure of a regular language.

Regular languages are normally described using more compact notation, which omits braces around one-string sets, uses $+$ to represent union instead of $\cup$, and juxtaposes subexpressions to represent concatenation instead of using an explicit operator $\bullet$; the resulting string of symbols is called a ***regular expression***. By convention, in the absence of parentheses, the * operator has highest precedence, followed by the (implicit) concatenation operator, followed by $+$. Thus, for example, the regular expression $\mathtt{10}^*$ is shorthand for the language $\{\mathtt{1}\} \bullet \{\mathtt{0}\}^*$ (containing all strings consisting of a $\mathtt{1}$ followed by zero or more $\mathtt{0}$s), and *not* the language $\{\mathtt{10}\}^*$ (containing all strings of even length that start with $\mathtt{1}$ and alternate between $\mathtt{1}$ and $\mathtt{0}$). As a larger example, the regular expression

$$\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$$

represents the language

$$\{\mathtt{0}\} \cup \Big(\{\mathtt{0}\}^* \bullet \{\mathtt{1}\} \bullet \big((\{\mathtt{1}\} \bullet \{\mathtt{0}\}^* \bullet \{\mathtt{1}\}) \cup (\{\mathtt{0}\} \bullet \{\mathtt{1}\}^* \bullet \{\mathtt{0}\})\big)^* \bullet \{\mathtt{1}\} \bullet \{\mathtt{0}\}^*\Big).$$

Here are a few more examples of regular expressions and the languages they represent.

- $\mathtt{0}^*$ — the set of all strings of $\mathtt{0}$s, including the empty string.

- $\mathtt{00000}^*$ — the set of all strings consisting of at least four $\mathtt{0}$s.
- $(\mathtt{00000})^*$ — the set of all strings of $\mathtt{0}$s whose length is a multiple of 5.
- $(\varepsilon + \mathtt{1})(\mathtt{01})^*(\varepsilon + \mathtt{0})$ — the set of all strings of alternating $\mathtt{0}$s and $\mathtt{1}$s, or equivalently, the set of all binary strings that do not contain the substrings $\mathtt{00}$ or $\mathtt{11}$.
- $((\varepsilon + \mathtt{0} + \mathtt{00} + \mathtt{000})\mathtt{1})^*(\varepsilon + \mathtt{0} + \mathtt{00} + \mathtt{000})$ — the set of all binary strings that do not contain the substring $\mathtt{0000}$.
- $((\mathtt{0}+\mathtt{1})(\mathtt{0}+\mathtt{1}))^*$ — the set of all binary strings whose length is even.
- $\mathtt{1}^*(\mathtt{01}^*\mathtt{01}^*)^*$ — the set of all binary strings with an even number of $\mathtt{0}$s.
- $\mathtt{0} + \mathtt{1}(\mathtt{0}+\mathtt{1})^*\mathtt{00}$ — the set of all non-negative binary numerals divisible by 4 and with no redundant leading $\mathtt{0}$s.
- $\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$ — the set of all non-negative binary numerals divisible by 3, possibly with redundant leading $\mathtt{0}$s.

The last example should ***not*** be obvious. It is straightforward, but rather tedious, to prove by induction that every string in $\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$ is the binary representation of a non-negative multiple of 3. It is similarly straightforward, and similarly tedious, to prove that the binary representation of *every* non-negative multiple of 3 matches this regular expression. In a later note, we will see a systematic method for deriving regular expressions for some languages that avoids (or more accurately, automates) this tedium.

Most of the time we do not distinguish between regular expressions and the languages they represent, for the same reason that we do not normally distinguish between the arithmetic expression "2+2" and the integer 4, or the symbol π and the area of the unit circle. However, we sometimes need to refer to regular expressions themselves *as strings*. In those circumstances, we write $\boldsymbol{L(R)}$ to denote the language represented by the regular expression R. String w ***matches*** regular expression R if and only if $w \in L(R)$. Two regular expressions R and R' are ***equivalent*** if they describe the same language; for example, the regular expressions $(\mathtt{0}+\mathtt{1})^*$ and $(\mathtt{1}+\mathtt{0})^*$ are equivalent, because the union operator is commutative.

Almost every regular language can be represented by infinitely many distinct but equivalent regular expressions, even if we ignore ultimately trivial equivalences like $L = (L\varnothing)^*L\varepsilon + \varnothing$.

2.4 Things What Ain't Regular Expressions

Many computing environments and programming languages support patterns called ***regexen*** (singular ***regex***, pluralized like *ox*) that are considerably more general and powerful than regular expressions. Regexen include special symbols representing negation, character classes (for example, upper-case letters, or digits), contiguous ranges of characters, line and word boundaries, limited repetition (as opposed to the unlimited repetition allowed by *), back-references to earlier subexpressions, and even local variables. Despite its obvious etymology, a regex is ***not*** necessarily a regular expression, and it does ***not*** necessarily describe a regular language![3]

Another type of pattern that is often confused with regular expression are ***globs***, which are patterns used in most Unix shells and some scripting languages to represent sets file names. Globs include symbols for arbitrary single characters (`?`), single characters from a

[3]However, regexen are not all-powerful, either; see http://stackoverflow.com/a/1732454/775369.

specified range (`[a-z]`), arbitrary substrings (`*`), and substrings from a specified finite set (`{foo,ba{r,z}}`). Globs are significantly *less* powerful than regular expressions.

2.5 Not Every Language is Regular

You may be tempted to conjecture that *all* languages are regular, but in fact, the following cardinality argument *almost all* languages are *not* regular. To make the argument concrete, let's consider languages over the single-symbol alphabet $\{\diamond\}$.

- Every regular expression over the one-symbol alphabet $\{\diamond\}$ is itself a string over the 7-symbol alphabet $\{\diamond, +, (,), *, \varepsilon, \emptyset\}$. By interpreting these symbols as the digits 1 through 7, we can interpret any string over this larger alphabet as the base-8 representation of some unique integer. Thus, the set of all regular expressions over $\{\diamond\}$ is *at most* as large as the set of integers, and is therefore countably infinite. It follows that the set of all regular *languages* over $\{\diamond\}$ is also countably infinite.

- On the other hand, for any real number $0 \le \alpha < 1$, we can define a corresponding language

  $$L_\alpha = \left\{ \diamond^n \;\middle|\; \alpha 2^n \bmod 1 \ge 1/2 \right\}.$$

  In other words, L_α contains the string $\diamond^n$ if and only if the $(n+1)$th bit in the binary representation of α is equal to 1. For any distinct real numbers $\alpha \ne \beta$, the binary representations of α and β must differ in some bit, so $L_\alpha \ne L_\beta$. We conclude that the set of ***all*** languages over $\{\diamond\}$ is *at least* as large as the set of real numbers between 0 and 1, and is therefore uncountably infinite.

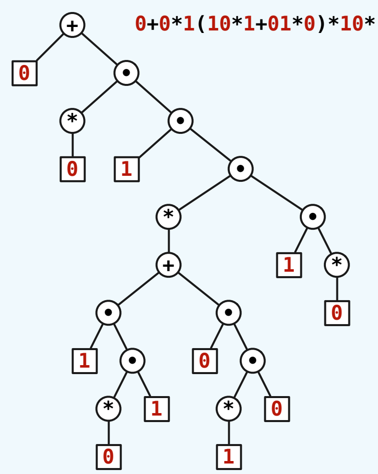
We will see several explicit examples of non-regular languages in future lectures. For example, the set of all regular expressions over $\{0, 1\}$ is not itself a regular language!

2.6 Parsing Regular Expressions

Most algorithms for regular expressions require them in the form of *regular expression trees*, rather than as raw strings. A regular expression tree is one of the following:

- A leaf node labeled $\varnothing$.
- A leaf node labeled with a string in Σ^*.
- A node labeled $+$ with two children, each the root of an expression tree.
- A node labeled $*$ with one child, which is the root of an expression tree.
- A node labeled $\bullet$ with two children, each the root of an expression tree.

In other words, a regular expression tree directly encodes a sequence of alternation, concatenation and Kleene closure operations that defines a regular language. Similarly, when we want to prove things about regular expressions or regular languages, it is more natural to think of subexpressions *as subtrees* rather than *as substrings*.



Given any regular expression of length n, we can ***parse*** it into an equivalent regular expression tree in $O(n)$ time. Thus, when we see an algorithmic problem that starts "Given a regular expression...", we can assume without loss of generality that we're actually given a regular expression *tree*.

We'll see more on this topic later.

Exercises

1. (a) Prove that $\{\varepsilon\} \bullet L = L \bullet \{\varepsilon\} = L$, for any language L.

   (b) Prove that $\varnothing \bullet L = L \bullet \varnothing = \varnothing$, for any language L.

   (c) Prove that $(A \bullet B) \bullet C = A \bullet (B \bullet C)$, for all languages A, B, and C.

   (d) Prove that $|A \bullet B| = |A| \cdot |B|$, for all languages A and B. (The second $\cdot$ is multiplication!)

   (e) Prove that L^* is finite if and only if $L = \varnothing$ or $L = \{\varepsilon\}$.

   (f) Prove that $AB = BC$ implies $A^*B = BC^* = A^*BC^*$, for all languages A, B, and C.

2. Recall that the reversal w^R of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

The reversal L^R of any language L is the set of reversals of all strings in L:

$$L^R := \{w^R \mid w \in L\}.$$

(a) Prove that $(AB)^R = B^R A^R$ for all languages A and B.

(b) Prove that $(L^R)^R = L$ for every language L.

(c) Prove that $(L^*)^R = (L^R)^*$ for every language L.

3. Prove that each of the following regular expressions is equivalent to $(0+1)^*$.

(a) $\varepsilon + 0(0+1)^* + 1(1+0)^*$

(b) $0^* + 0^*1(0+1)^*$

(c) $((\varepsilon + 0)(\varepsilon + 1))^*$

(d) $0^*(10^*)^*$

(e) $(1^*0)^*(0^*1)^*$

4. For each of the following languages in $\{0,1\}^*$, describe an equivalent regular expression. There are infinitely many correct answers for each language. (This problem will become significantly simpler after we've seen finite-state machines, in the next lecture note.)

(a) Strings that end with the suffix $0^9 = 000000000$.

(b) All strings except 010.

(c) Strings that contain the substring 010.

(d) Strings that contain the subsequence 010.

(e) Strings that do not contain the substring 010.

(f) Strings that do not contain the subsequence 010.

(g) Strings that contain an even number of occurrences of the substring 010.

⋆(h) Strings that contain an even number of occurrences of the substring 000.

(i) Strings in which every occurrence of the substring 00 appears before every occurrence of the substring 11.

(j) Strings w such that *in every prefix of* w, the number of 0s and the number of 1s differ by at most 1.

⋆(k) Strings w such that *in every prefix of* w, the number of 0s and the number of 1s differ by at most 2.

⋆(l) Strings in which the number of 0s and the number of 1s differ by a multiple of 3.

⋆(m) Strings that contain an even number of 1s and an odd number of 0s.

★(n) Strings that represent a number divisible by 5 in binary.

5. Prove that for any regular expression R such that $L(R)$ is nonempty, there is a regular expression equivalent to R that does not use the empty-set symbol $\varnothing$.

6. Prove that if L is a regular language, then L^R is also a regular language. *[Hint: How do you reverse a regular expression?]*

7. (a) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)$ is empty.

   (b) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)$ is infinite.

   In each problem, assume you are given R as a regular expression tree, not just a raw string.

> **Caveat lector!** This is the first edition of this lecture note. A few topics are missing, and there are almost certainly a few serious errors. Please send bug reports and suggestions to jeffe@illinois.edu.

Life only avails, not the having lived. Power ceases in the instant of repose;
it resides in the moment of transition from a past to a new state,
in the shooting of the gulf, in the darting to an aim.

— Ralph Waldo Emerson, "Self Reliance", *Essays, First Series* (1841)

O Marvelous! what new configuration will come next?
I am bewildered with multiplicity.

— William Carlos Williams, "At Dawn" (1914)

3 Finite-State Machines

3.1 Intuition

Suppose we want to determine whether a given string $w[1..n]$ of bits represents a multiple of 5 in binary. After a bit of thought, you might realize that you can read the bits in w one at a time, from left to right, keeping track of the value modulo 5 of the prefix you have read so far.

```
MultipleOf5(w[1..n]):
  rem ← 0
  for i ← 1 to n
      rem ← (2 · rem + w[i]) mod 5
  if rem = 0
      return True
  else
      return False
```

Aside from the loop index i, which we need just to read the entire input string, this algorithm has a single local variable *rem*, which has only four different values (0, 1, 2, 3, or 4).

This algorithm already runs in $O(n)$ time, which is the best we can hope for—after all, we have to read every bit in the input—but we can speed up the algorithm *in practice*. Let's define a ***change*** or ***transition*** function $\delta\colon \{0,1,2,3,4\} \times \{\mathtt{0},\mathtt{1}\} \to \{0,1,2,3,4\}$ as follows:

$$\delta(q,a) = (2q + a) \bmod 5.$$

(Here I'm implicitly converting the symbols $\mathtt{0}$ and $\mathtt{1}$ to the corresponding integers 0 and 1.) Since we already know all values of the transition function, we can store them in a precomputed table, and then replace the computation in the main loop of MultipleOf5 with a simple array lookup.

We can also modify the return condition to check for different values modulo 5. To be completely general, we replace the final if-then-else lines with another array lookup, using an array $A[0..4]$ of booleans describing which final mod-5 values are "acceptable".

After both of these modifications, our algorithm can be rewritten as follows, either iteratively or recursively (with $q = 0$ in the initial call):

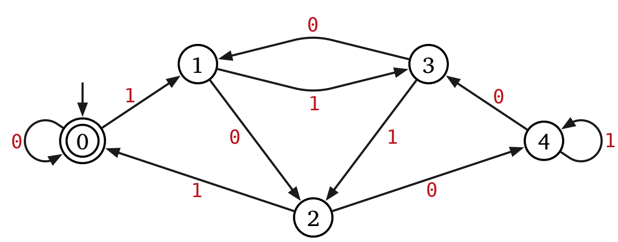
```
DoSomethingCool(w[1..n]):
  q ← 0
  for i ← 1 to n
      q ← δ[q, w[i]]
  return A[q]
```

```
DoSomethingCool(q, w):
  if w = ε
      return A[q]
  else
      decompose w = a · x
      return DoSomethingCool(δ(q, a), x)
```

If we want to use our new DoSomethingCool algorithm to implement MultipleOf5, we simply give the arrays δ and A the following hard-coded values:

q	$\delta[q, 0]$	$\delta[q, 1]$	$A[q]$
0	0	1	True
1	2	3	False
2	4	0	False
3	1	2	False
4	3	4	False

We can also visualize the behavior of DoSomethingCool by drawing a directed graph, whose vertices represent possible values of the variable q—the possible ***states*** of the algorithm—and whose edges are labeled with input symbols to represent transitions between states. Specifically, the graph includes the labeled directed edge $p \xrightarrow{a} q$ if and only if $\delta(p, a) = q$. To indicate the proper return value, we draw the "acceptable" final states using doubled circles. Here is the resulting graph for MultipleOf5:



State-transition graph for MultipleOf5

If we run the MultipleOf5 algorithm on the string `00101110110` (representing the number 374 in binary), the algorithm performs the following sequence of transitions:

$$0 \xrightarrow{0} 0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{0} 2 \xrightarrow{1} 0 \xrightarrow{1} 1 \xrightarrow{1} 3 \xrightarrow{0} 1 \xrightarrow{1} 3 \xrightarrow{1} 2 \xrightarrow{0} 4$$

Because the final state is not the "acceptable" state 0, the algorithm correctly returns False. We can also think of this sequence of transitions as a walk in the graph, which is completely determined by the start state 0 and the sequence of edge labels; the algorithm returns True if and only if this walk ends at an "acceptable" state.

3.2 Formal Definitions

The object we have just described is an example of a ***finite-state machine***. A finite-state machine is a formal model of any system/machine/algorithm that can exist in a finite number of ***states*** and that transitions among those states based on sequence of ***input*** symbols.

Finite-state machines are also commonly called ***deterministic finite-state automata***, abbreviated ***DFAs***. The word "deterministic" means that the behavior of the machine is completely *determined* by the input string; we'll discuss nondeterministic automata in the next lecture. The word "automaton" (plural "automata") comes from ancient Greek $\alpha\upsilon\tau o\mu\alpha\tau o\varsigma$ meaning "self-acting", from the roots $\alpha\upsilon\tau o$- ("self") and -$\mu\alpha\tau o\varsigma$ ("thinking, willing", the root of Latin *mentus*).

Formally, every finite-state machine consists of five components:

- An arbitrary finite set $\boldsymbol{\Sigma}$, called the ***input alphabet***.
- Another arbitrary finite set $\boldsymbol{Q}$, whose elements are called ***states***.
- An arbitrary ***transition*** function $\boldsymbol{\delta} : Q \times \Sigma \to Q$.
- A ***start state*** $\boldsymbol{s} \in Q$.
- A subset $\boldsymbol{A} \subseteq Q$ of ***accepting states***.

The behavior of a finite-state machine is governed by an ***input string*** w, which is a finite sequence of symbols from the input alphabet Σ. The machine ***reads*** the symbols in w one at a time in order (from left to right). At all times, the machine has a *current state* q; initially q is the machine's start state s. Each time the machine reads a symbol a from the input string, its current state ***transitions*** from q to $\delta(q,a)$. After all the characters have been read, the machine ***accepts*** w if the current state is in A and ***rejects*** w otherwise. In other words, every finite state machine runs the algorithm DoSomethingCool! The ***language*** of a finite state machine M, denoted $\boldsymbol{L(M)}$ is the set of all strings that M accepts.

More formally, we extend the transition function $\delta : Q \times \Sigma^* \to Q$ of any finite-state machine to a function $\delta^* : Q \times \Sigma^* \to Q$ that transitions on *strings* as follows:

$$\delta^*(q,w) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*(\delta(q,a),x) & \text{if } w = ax. \end{cases}$$

Finally, a finite-state machine ***accepts*** a string w if and only if $\delta^*(s,w) \in A$, and ***rejects*** w otherwise. (Compare this definition with the recursive formulation of DoSomethingCool!)

For example, our final MultipleOf5 algorithm is a DFA with the following components:

- input alphabet: $\Sigma = \{\mathtt{0}, \mathtt{1}\}$
- state set: $Q = \{0,1,2,3,4\}$
- transition function: $\delta(q,a) = (2q + a) \bmod 5$
- start state: $s = 0$
- accepting states: $A = \{0\}$

This machine rejects the string `00101110110`, because

$$\begin{aligned}
\delta^*(0,\mathtt{00101110110}) &= \delta^*(\delta(0,\mathtt{0}),\mathtt{0101110110}) \\
&= \delta^*(0,\mathtt{0101110110}) = \delta^*(\delta(0,\mathtt{0}),\mathtt{101110110}) \\
&= \delta^*(0,\mathtt{101110110}) = \delta^*(\delta(0,\mathtt{1}),\mathtt{01110110}) = \cdots \\
&\vdots \\
\cdots &= \delta^*(1,\mathtt{110}) = \delta^*(\delta(1,\mathtt{1}),\mathtt{10}) \\
&= \delta^*(3,\mathtt{10}) = \delta^*(\delta(3,\mathtt{1}),\mathtt{0}) \\
&= \delta^*(2,\mathtt{0}) = \delta^*(\delta(3,\mathtt{0}),\varepsilon) \\
&= \delta^*(4,\varepsilon) = 4 \notin A.
\end{aligned}$$

We have already seen a more graphical representation of this entire sequence of transitions:

$$0 \xrightarrow{\texttt{0}} 0 \xrightarrow{\texttt{0}} 0 \xrightarrow{\texttt{1}} 1 \xrightarrow{\texttt{0}} 2 \xrightarrow{\texttt{1}} 0 \xrightarrow{\texttt{1}} 1 \xrightarrow{\texttt{1}} 3 \xrightarrow{\texttt{0}} 1 \xrightarrow{\texttt{1}} 3 \xrightarrow{\texttt{1}} 2 \xrightarrow{\texttt{0}} 4$$

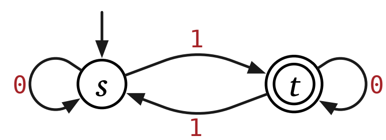
The arrow notation is easier to read and write for specific examples, but surprisingly, most people actually find the more formal functional notation easier to use in formal proofs. Try them both!

We can equivalently define a DFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ, such that every vertex has exactly one outgoing edge with each label. In our drawings of finite state machines, the start state s is always indicated by an incoming arrow, and the accepting states A are always indicted by doubled circles. By induction, for any string $w \in \Sigma^*$, this graph contains a unique walk that starts at s and whose edges are labeled with the symbols in w in order. The machine accepts w if this walk ends at an accepting state. This graphical formulation of DFAs is incredibly useful for developing intuition and even designing DFAs. For proofs, it's largely a matter of taste whether to write in terms of extended transition functions or labeled graphs, but (as much as I wish otherwise) I actually find it easier to write ***correct*** proofs using the functional formulation.

3.3 Another Example

The following drawing shows a finite-state machine with input alphabet $\Sigma = \{\texttt{0}, \texttt{1}\}$, state set $Q = \{s, t\}$, start state s, a single accepting state t, and the transition function

$$\delta(s, \texttt{0}) = s, \quad \delta(s, \texttt{1}) = t, \quad \delta(t, \texttt{0}) = t, \quad \delta(t, \texttt{1}) = s.$$



A simple finite-state machine.

For example, the two-state machine M at the top of this page accepts the string `00101110100` after the following sequence of transitions:

$$s \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{1}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{1}} s \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{0}} t.$$

The same machine M rejects the string `11100101` after the following sequence of transitions:

$$s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{1}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{0}} s \xrightarrow{\texttt{1}} t \xrightarrow{\texttt{0}} t \xrightarrow{\texttt{1}} s.$$

Finally, M rejects the empty string, because the start state s is not an accepting state.

From these examples and others, it is easy to conjecture that the language of M is the set of all strings of `0`s and `1`s with an odd number of `1`s. So let's prove it!

Proof (tedious case analysis): Let $\#(a, w)$ denote the number of times symbol a appears in string w. We will prove the following stronger claims, for any string w.

$$\delta^*(s, w) = \begin{cases} s & \text{if } \#(\texttt{1}, w) \text{ is even} \\ t & \text{if } \#(\texttt{1}, w) \text{ is odd} \end{cases} \qquad \text{and} \qquad \delta^*(t, w) = \begin{cases} t & \text{if } \#(\texttt{1}, w) \text{ is even} \\ s & \text{if } \#(\texttt{1}, w) \text{ is odd} \end{cases}$$

Let w be an arbitrary string. Assume that for any string x that is shorter than w, we have $\delta^*(s, x) = s$ and $\delta^*(t, x) = t$ if x has an even number of `1`s, and $\delta^*(s, x) = t$ and $\delta^*(t, x) = s$ if x has an odd number of `1`s. There are five cases to consider.

- If $w = \varepsilon$, then w contains an even number of 1s and $\delta^*(s, w) = s$ and $\delta^*(t, w) = t$ by definition.
- Suppose $w = 1x$ and $\#(1, w)$ is even. Then $\#(1, x)$ is odd, which implies

$$\begin{aligned}
\delta^*(s,w) &= \delta^*(\delta(s,1),x) && \text{by definition of } \delta^* \\
&= \delta^*(t,x) && \text{by definition of } \delta \\
&= s && \text{by the inductive hypothesis}
\end{aligned}$$

$$\begin{aligned}
\delta^*(t,w) &= \delta^*(\delta(t,1),x) && \text{by definition of } \delta^* \\
&= \delta^*(s,x) && \text{by definition of } \delta \\
&= T && \text{by the inductive hypothesis}
\end{aligned}$$

  Since the remaining cases are similar, I'll omit the line-by-line justification.

- If $w = 1x$ and $\#(1, w)$ is odd, then $\#(1, x)$ is even, so the inductive hypothesis implies

$$\delta^*(s,w) = \delta^*(\delta(s,1),x) = \delta^*(t,x) = t$$
$$\delta^*(t,w) = \delta^*(\delta(t,1),x) = \delta^*(s,x) = s$$

- If $w = 0x$ and $\#(1, w)$ is even, then $\#(1, x)$ is even, so the inductive hypothesis implies

$$\delta^*(s,w) = \delta^*(\delta(s,0),x) = \delta^*(s,x) = s$$
$$\delta^*(t,w) = \delta^*(\delta(t,0),x) = \delta^*(t,x) = t$$

- Finally, if $w = 0x$ and $\#(1, w)$ is odd, then $\#(1, x)$ is odd, so the inductive hypothesis implies

$$\delta^*(s,w) = \delta^*(\delta(s,0),x) = \delta^*(s,x) = t$$
$$\delta^*(t,w) = \delta^*(\delta(t,0),x) = \delta^*(t,x) = s \qquad \square$$

Notice that this proof contains $|Q|^2 \cdot |\Sigma| + |Q|$ separate inductive arguments. For every pair of states p and q, we must argue about the language so strings w such that $\delta^*(p, w) = q$, and we must consider each first symbol in w. We must also argue about $\delta(p, \varepsilon)$ for every state p. Each of those arguments is typically straightforward, but it's easy to get lost in the deluge of cases.

For this particular proof, however, we can reduce the number of cases by switching from tail recursion to *head* recursion. The following identity holds for all strings $x \in \Sigma^*$ and symbols $a \in \Sigma$:

$$\boxed{\delta^*(q, xa) = \delta(\delta^*(q, x), a)}$$

We leave the inductive proof of this identity as a straightforward exercise (hint, hint).

Proof (clever renaming, head induction): Let's rename the states 0 and 1 instead of s and t. Then the transition function can be described concisely as $\boldsymbol{\delta(q, a) = (q + a) \bmod 2}$.

Now we claim that for every string w, we have $\delta^*(0, w) = \#(1, w) \bmod 2$. So let w be an arbitrary string, and assume that for any string x that is shorter than w that $\delta^*(0, x) = \#(1, x) \bmod 2$. There are only two cases to consider: either w is empty or it isn't.

- If $w = \varepsilon$, then $\delta^*(0, w) = 0 = \#(1, w) \bmod 2$ by definition.

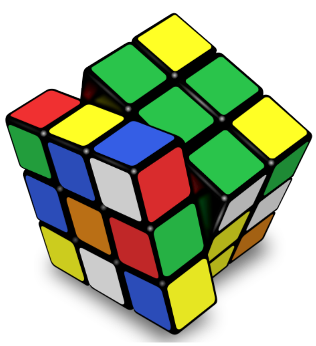
- Otherwise, $w = xa$ for some string x and some symbol a, and we have

$$\begin{aligned}
\delta^*(0, w) &= \delta(\delta^*(0, x), a) \\
&= \delta(\#(\mathtt{1}, x) \bmod 2, a) && \text{by the inductive hypothesis} \\
&= (\#(\mathtt{1}, x) \bmod 2 + a) \bmod 2 && \text{by definition of } \delta \\
&= (\#(\mathtt{1}, x) + a) \bmod 2 && \text{by definition of mod 2} \\
&= (\#(\mathtt{1}, x) + \#(\mathtt{1}, a)) \bmod 2 && \text{because } \#(\mathtt{1}, \mathtt{0}) = 0 \text{ and } \#(\mathtt{1}, \mathtt{1}) = 1 \\
&= (\#(\mathtt{1}, xa)) \bmod 2 && \text{by definition of } \# \\
&= (\#(\mathtt{1}, w)) \bmod 2 && \text{because } w = xa \qquad \square
\end{aligned}$$

Hmmm. This "clever" proof is certainly shorter than the earlier brute-force proof, but is it really "better"? "Simpler"? More intuitive? Easier to understand? I'm skeptical. Sometimes brute force really is more effective.

3.4 Yet Another Example

As a more complex example, consider the ***Rubik's cube***, a well-known mechanical puzzle invented independently by Ern Rubik in Hungary and Terutoshi Ishigi in Japan in the mid-1970s. This puzzle has precisely 519,024,039,293,878,272,000 distinct configurations. In the unique *solved* configuration, each of the six faces of the cube shows exactly one color. We can change the configuration of the cube by rotating one of the six faces of the cube by 90 degrees, either clockwise or counterclockwise. The cube has six faces (front, back, left, right, up, and down), so there are exactly twelve possible turns, typically represented by the symbols $\mathtt{R}, \mathtt{L}, \mathtt{F}, \mathtt{B}, \mathtt{U}, \mathtt{D}, \bar{\mathtt{R}}, \bar{\mathtt{L}}, \bar{\mathtt{F}}, \bar{\mathtt{B}}, \bar{\mathtt{U}}, \bar{\mathtt{D}}$, where the letter indicates which face to turn and the presence or absence of a bar over the letter indicates turning counterclockwise or clockwise, respectively. Thus, we can represent a Rubik's cube as a finite-state machine with 519,024,039,293,878,272,000 states and an input alphabet with 12 symbols; or equivalently, as a directed graph with 519,024,039,293,878,272,000 vertices, each with 12 outgoing edges. In practice, the number of states is *far* too large for us to actually draw the machine or explicitly specify its transition function; nevertheless, the number of states is still finite. If we let the start state s and the sole accepting state be the solved state, then the language of this finite state machine is the set of all move sequences that leave the cube unchanged.



A complicated finite-state machine.

3.5 Building DFAs

This section describes a few examples of building DFAs that accept particular languages, thereby proving that those languages are automatic. As usual in algorithm design, there is no purely mechanical recipe—no *automatic* method—no *algorithm*—for building DFAs in general. However, the following examples show several useful design strategies.

3.5.1 Superstrings

Perhaps the simplest rule of thumb is to try to construct an algorithm that looks like MultipleOf5: A simple for-loop through the symbols, using a *constant* number of variables, where each variable (except the loop index) has only a *constant* number of possible values. Here, "constant" means an actual number that is not a function of the input size n. You should be able to compute the number of possible values for each variable *at compile time*.

For example, the following algorithm determines whether a given string in $\Sigma = \{0, 1\}$ contains the substring 11.

```
Contains11(w[1..n]):
  found ← False
  for i ← 1 to n
      if i = 1
          last2 ← w[1]
      else
          last2 ← w[1] · w[2]
      if last = 11
          found ← True
  return found
```

Aside from the loop index, this algorithm has exactly two variables.

- A boolean flag *found* indicating whether we have seen the substring 11. This variable has exactly two possible values: True and False.
- A string *last2* containing the last (up to) three symbols we have read so far. This variable has exactly 7 possible values: ε, 0, 1, 00, 01, 10, and 11.

Thus, altogether, the algorithm can be in at most $2 \times 7 = 14$ possible states, one for each possible pair (*found*, *last2*). Thus, we can encode the behavior of Contains11 as a DFA with fourteen states, where the start state is (False, ε) and the accepting states are all seven states of the form (True, $*$). The transition function is described in the following table (split into two parts to save space):

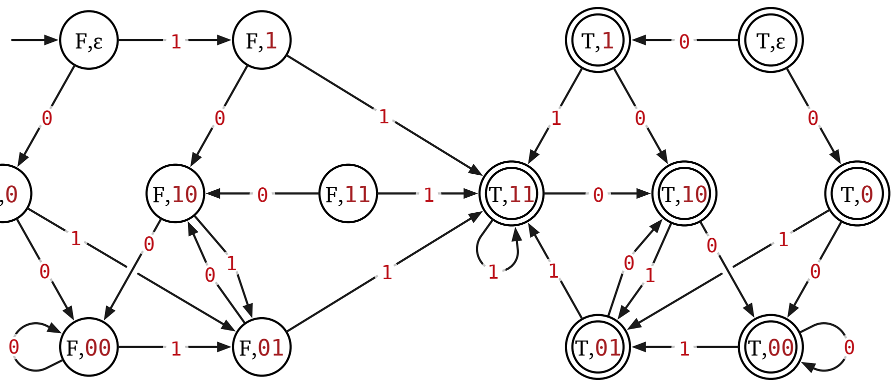
q	$\delta[q, 0]$	$\delta[q, 1]$	q	$\delta[q, 0]$	$\delta[q, 1]$
(False, ε)	(False, 0)	(False, 1)	(True, ε)	(True, 0)	(True, 1)
(False, 0)	(False, 00)	(False, 01)	(True, 0)	(True, 00)	(True, 01)
(False, 1)	(False, 10)	(**True**, 11)	(True, 1)	(True, 10)	(True, 11)
(False, 00)	(False, 00)	(False, 01)	(True, 00)	(True, 00)	(True, 01)
(False, 01)	(False, 10)	(**True**, 11)	(True, 01)	(True, 10)	(True, 11)
(False, 10)	(False, 00)	(False, 01)	(True, 10)	(True, 00)	(True, 01)
(False, 11)	(False, 10)	(**True**, 11)	(True, 11)	(True, 10)	(True, 11)

For example, given the input string 1001011100, this DFA performs the following sequence of transitions and then accepts.

$$(\textsc{False}, \varepsilon) \xrightarrow{1} (\textsc{False}, 1) \xrightarrow{0} (\textsc{False}, 10) \xrightarrow{0} (\textsc{False}, 00) \xrightarrow{1}$$
$$(\textsc{False}, 01) \xrightarrow{0} (\textsc{False}, 10) \xrightarrow{1} (\textsc{False}, 01) \xrightarrow{1}$$
$$(\textsc{True}, 11) \xrightarrow{1} (\textsc{True}, 11) \xrightarrow{0} (\textsc{True}, 10) \xrightarrow{0} (\textsc{True}, 00)$$

3.5.2 Reducing states

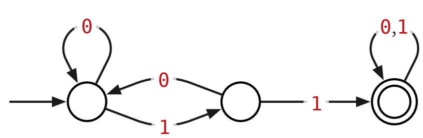
You can probably guess that the brute-force DFA we just constructed has considerably more states than necessary, especially after seeing its transition graph:



Our brute-force DFA for strings containing the substring 11

For example, we don't need actually to remember both of the last two symbols, but only the penultimate symbol, because the last symbol is the one we're currently reading. This observation allows us to reduce the number of states from fourteen to only six. Once the flag part of the state is set to TRUE, we know the machine will eventually accept, so we might as well merge the two accepting states together. Finally, and more subtly, because all transitions out of $(\textsc{False}, \varepsilon)$ and $(\textsc{False}, 0)$ are identical, we can merge those two states together as well. In the end, we obtain the following DFA with just three states:

- The start state, which indicates that the machine has not read the substring 11 an did not just read the symbol 1.
- An intermediate state, which indicates that the machine has not read the substring 11 but just read the symbol 1.
- A unique accept state, which indicates that the machine has read the substring 11.



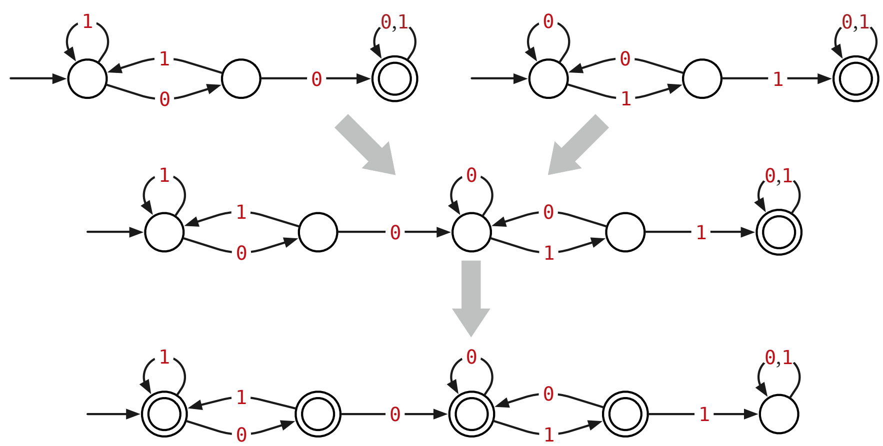
A minimal DFA for superstrings of 11

At the end of this note, I'll describe an efficient algorithm to transform any given DFA into an equivalent DFA with the fewest possible states. Given that this minimization algorithm exists, there is very little incentive to optimize DFAs *by hand*. Clarity is infinitely more important than brevity, especially in this class.

3.5.3 Every this after that

Suppose we want to accept the set of strings in which every occurrence of the substring `00` occurs after every occurrence of the substring `11`. Equivalently, we want to *reject* every string in which some `00` occurs before `11`. Often the easiest way to design a DFA to check whether a string is *not* in some set is first to build a DFA that *is* in that set and then invert which states in that machine are accepting.

From the previous example, we know that there is a three-state DFA M_{11} that accepts the set of strings with the substring `11` and a nearly identical DFA M_{00} that accepts the set of strings containing the substring `00`. By identifying the accept state of M_{00} with the start state of M_{11}, we obtain a five-state DFA that accepts the set of strings with `00` before `11`. Finally, by inverting which states are accepting, we obtain the DFA we want.



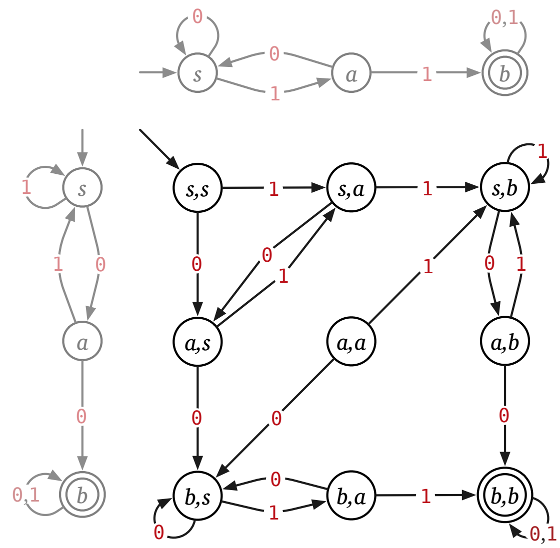
Building a DFA for the language of strings in which every `00` is after every `11`.

3.5.4 Both This and That: The Product Construction

Now suppose we want to accept all strings that contain both `00` and `11` as substrings, in either order. Intuitively, we'd like to run two of our earlier DFAs in parallel—the DFA M_{00} to detect superstrings of `00` and the DFA M_{11} to detect superstrings of `11`—and then accept the input string if and only if both of these DFAs accept. In fact, we can encode precisely this "parallel computation" into a single DFA, whose states are all ordered pairs (p,q), where p is a state in M_{00} and q is a state in M_{11}. The new "parallel" DFA includes the transition $(p,q) \xrightarrow{a} (p',q')$ if and only if M_{00} contains the transition $p \xrightarrow{a} p'$ and M_{11} contains the transition $q \xrightarrow{a} q'$. Finally, the state (p,q) is accepting if and only if p and q are accepting states in their respective machines. The resulting nine-state DFA is shown on the next page.

More generally, let $M_1 = (\Sigma, Q_1, \delta_1, s_1, A_1)$ be an arbitrary DFA that accepts some language L_1, and let $M_2 = (\Sigma, Q_2, \delta_2, s_2, A_2)$ be an arbitrary DFA that accepts some language L_2 (over the same alphabet Σ). We can construct a third DFA $M = (\Sigma, Q, \delta, s, A)$ that accepts the intersection language $L_1 \cap L_2$ as follows.

$$
\begin{aligned}
Q &:= Q_1 \times Q_2 = \big\{(p,q) \bigm| p \in Q_1 \text{ and } q \in Q_2\big\} \\
s &:= (s_1, s_2) \\
A &:= A_1 \times A_2 = \big\{(p,q) \bigm| p \in A_1 \text{ and } q \in A_2\big\} \\
\delta((p,q),a) &:= \big(\delta_1(p,a),\ \delta_2(q,a)\big)
\end{aligned}
$$



Building a DFA for the language of strings in which every `00` is after every `11`.

To convince yourself that this product construction is actually correct, consider the extended transition function $\delta^* : (Q \times Q') \times \Sigma^* \to (Q \times Q')$, which acts on strings instead of individual symbols. Recall that this function is defined recursively as follows:

$$\delta^*\big((p,q),w\big) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*\big(\delta((p,q),a),\ x\big) & \text{if } w = ax. \end{cases}$$

Inductive definition-chasing gives us the identity $\delta^*((p,q),w) = \big(\delta_1^*(p,w),\ \delta_2^*(q,w)\big)$ for any string w:

$$\begin{aligned} \delta^*\big((p,q),\varepsilon\big) &= (p,q) && \text{by the definition of } \delta^* \\ &= \big(\delta_1^*(p,\varepsilon),\ \delta_2^*(q,\varepsilon)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*; \\ \delta^*\big((p,q),ax\big) &= \delta^*\big(\delta((p,q),a),\ x\big) && \text{by the definition of } \delta^* \\ &= \delta^*\big((\delta_1(p,a),\ \delta_2(q,a)),\ x\big) && \text{by the definition of } \delta \\ &= \big(\delta_1^*((\delta_1(p,a),\ x),\ \delta_2^*(\delta_2(q,a),x)\big) && \text{by the induction hypothesis} \\ &= \big(\delta_1^*(p,ax),\ \delta_2^*(q,ax)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*. \end{aligned}$$

It now follows from this seemingly impenetrable wall of notation that for any string w, we have $\delta^*(s,w) \in A$ if and only if both $\delta_1^*(s_1,w) \in A_1$ and $\delta_2^*(s_2,w) \in A_2$. In other words, M accepts w if and only if both M_1 and M_2 accept w, as required.

As usual, this construction technique does not necessarily yield *minimal* DFAs. For example, in our first example of a product DFA, illustrated above, the central state (a,a) cannot be reached by any other state and is therefore redundant. Whatever.

Similar product constructions can be used to build DFAs that accept any other boolean combination of languages; in fact, the only part of the construction that needs to be changed is the choice of accepting states. For example:

- To accept the union $L_1 \cup L_2$, define $A = \big\{(p,q) \bigm| p \in A_1 \textbf{ or } q \in A_2\big\}$.

- To accept the difference $L_1 \setminus L_2$, define $A = \{(p,q) \mid p \in A_1 \textbf{ but not } q \notin A_2\}$.
- To accept the symmetric difference $L_1 \oplus L_2$, define $A = \{(p,q) \mid p \in A_1 \textbf{ xor } q \in A_2\}$.

Moreover, by cascading this product construction, we can construct DFAs that accept arbitrary boolean combinations of arbitrary finite collections of regular languages.

3.6 Decision Algorithms

★★★

It's unclear how much we can say here, since we haven't yet talked about graph algorithms, or even really about graphs. Perhaps this discussion should simply be moved to the graph-traversal notes.

- ***Is $w \in L(M)$?*** Follow the unique path from q_0 with label w. By definition, $w \in L(M)$ if and only if this path leads to an accepting state.
- ***Is $L(M)$ empty?*** The language $L(M)$ is empty if and only if no accepting state is reachable from q_0. This condition can be checked in $O(n)$ time via whatever-first search, where n is the number of states. Alternatively, but less usefully, $L(M) = \varnothing$ if and only if $L(M)$ contains no string w such that $|w| < n$.
- ***Is $L(M)$ finite?*** Remove all states unreachable from q_0 (via whatever first search). Then $L(M)$ is finite if and only if the reduced DFA is a dag; this condition can be checked by depth-first search. Alternatively, but less usefully, $L(M)$ is finite if and only if $L(M)$ contains no string w such that $n \le |w| < 2n$.
- ***Is $L(M) = \Sigma^*$?*** Remove all states unreachable from q_0 (via whatever first search). Then $L(M) = \Sigma^*$ if and only if every state in M is an accepting state.
- ***Is $L(M) = L(M')$?*** Build a DFA N such that $L(N) = L(M) \setminus L(M')$ using a standard product construction, and then check whether $L(N) = \varnothing$.

3.7 Closure Properties

★★★

We haven't yet proved that automatic languages are regular yet, so formally, for now, some of these are closure properties of ***automatic*** languages.

- Complement (easy for DFAs, hard for regular expressions.)
- Concatenation (trivial for regular expressions, hard for DFAs)
- Union (trivial for regular expressions, easy for DFAs via product)
- Intersection (hard for regular expressions, easy for DFAs via product)
- Difference (hard for regular expressions, easy for DFAs via product)
- Kleene star: wait for NFAs (trivial for regular expression, hard for DFAs)
- Homomorphism: only mention in passing
- Inverse homomorphism: only mention in passing

3.8 Fooling Sets

Fix an arbitrary language L over an arbitrary alphabet Σ. For any strings $x, y, z \in \Sigma^*$, we say that ***z distinguishes x from y*** if exactly one of the strings xz and yz is in L. If no string distinguishes x and y, we say that x and y are ***L-equivalent*** and write $\boldsymbol{x \equiv_L y}$. Thus,

$$x \equiv_L y \iff \text{For every string } z \in \Sigma^*, \text{ we have } xz \in L \text{ if and only if } yz \in L.$$

For example, let L_{eo} denote the language of strings over $\{0,1\}$ with an even number of 0s and an odd number of 1s. Then the strings $x = 01$ and $y = 0011$ are distinguished by the string $z = 100$, because

$$xz = 01 \bullet 100 = 01100 \in L_{eo}$$
$$yz = 0011 \bullet 100 = 0011100 \notin L_{eo}.$$

On the other hand, it is quite easy to prove (hint, hint) that the strings 0001 and 1011 are L_{eo}-equivalent.

Let M be an arbitrary DFA for an arbitrary language L, and let x be y be arbitrary strings. If x and y lead to the same state in M—that is, if $\delta^*(s,x) = \delta^*(s,y)$—then we have

$$\delta^*(s,xz) = \delta^*(\delta^*(s,x),z) = \delta^*(\delta^*(s,y),z) = \delta^*(s,yz)$$

for any string z. In particular, either M accepts both x and y, or M rejects both x and y, and therefore $x \equiv_L y$. It follows that if x and y are not L-equivalent, then ***any*** DFA that accepts L has at least two distinct states $\delta^*(s,x) \neq \delta^*(s,y)$.

Finally, a ***fooling set*** for L is a set F of strings such that *every* pair of strings in F has a distinguishing suffix. For example, $F = \{01, 101, 010, 1010\}$ is a fooling set for the language L_{eo} of strings with an even number of 0s and an odd number of 1s, because each pair of strings in F has a distinguishing suffix:

- 0 distinguishes 01 and 101;
- 0 distinguishes 01 and 010;
- 0 distinguishes 01 and 1010;
- 10 distinguishes 101 and 010;
- 1 distinguishes 101 and 1010;
- 1 distinguishes 010 and 1010.

The pigeonhole principle now implies that for any integer k, if language L is accepted by a DFA with k states, then *every* fooling set for L contains at most k strings. This simple observation has two immediate corollaries.

First, for any integer k, if L has a fooling set of size k, then *every* DFA that accepts L has at least k states. For example, the fooling set $\{01, 101, 010, 1010\}$ proves that any DFA for L_{eo} has at least four states. Thus, we can use fooling sets to prove that certain DFAs are as small as possible.

Second, and more interestingly, if a language L is accepted by *any* DFA, then *every* fooling set for L must be finite. Equivalently:

If L has an infinite fooling set, then L is not accepted by any DFA.

This is arguably both the simplest and most powerful method for proving that a language is non-regular. Here are a few canonical examples of the fooling-set technique in action.

Lemma 3.1. *The language $L = \{0^n 1^n \mid n \geq 0\}$ is not regular.*

Proof: Consider the set $F = \{0^n \mid n \geq 0\}$, or more simply $F = 0^*$. Let x and y be arbitrary distinct strings in F. Then we must have $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$. The suffix $z = 1^i$ distinguishes x and y, because $xz = 0^i 1^i \in L$, but $yz = 0^i 1^j \notin L$. We conclude that F is a fooling set for L. Because F is infinite, L cannot be regular. □

Lemma 3.2. *The language $L = \{ww^R \mid w \in \Sigma^*\}$ of even-length palindromes is not regular.*

Proof: Let x and y be arbitrary distinct strings in $\mathtt{0}^*\mathtt{1}$. Then we must have $x = \mathtt{0}^i\mathtt{1}$ and $y = \mathtt{0}^j\mathtt{1}$ for some integers $i \neq j$. The suffix $z = \mathtt{10}^i$ distinguishes x and y, because $xz = \mathtt{0}^i\mathtt{110}^i \in L$, but $yz = \mathtt{0}^i\mathtt{110}^j \notin L$. We conclude that $\mathtt{0}^*\mathtt{1}$ is a fooling set for L. Because $\mathtt{0}^*\mathtt{1}$ is infinite, L cannot be regular. □

Lemma 3.3. *The language $L = \{\mathtt{0}^{2^n} \mid n \geq 0\}$ is not regular.*

Proof: Let x and y be arbitrary distinct strings in L. Then we must have $x = \mathtt{0}^{2^i}$ and $y = \mathtt{0}^{2^j}$ for some integers $i \neq j$. The suffix $z = \mathtt{0}^{2^i}$ distinguishes x and y, because $xz = \mathtt{0}^{2^i+2^i} = \mathtt{0}^{2^{i+1}} \in L$, but $yz = \mathtt{0}^{2^i+2^j} \notin L$. We conclude that L itself is a fooling set for L. Because L is infinite, L cannot be regular. □

Lemma 3.4. *The language $L = \{\mathtt{0}^p \mid p \text{ is prime}\}$ is not regular.*

Proof: Again, we use $\mathtt{0}^*$ as our fooling set, but but the actual argument is somewhat more complicated than in our earlier examples.

Let x and y be arbitrary distinct strings in $\mathtt{0}^*$. Then we must have $x = \mathtt{0}^i$ and $y = \mathtt{0}^j$ for some integers $i \neq j$. Without loss of generality, assume that $i < j$. Let p be any prime number larger than i. Because $p + 0(j - i)$ is prime and $p + p(j - i) > p$ is not, there must be a positive integer $k \leq p$ such that $p + (k-1)(j-i)$ is prime but $p + k(j-i)$ is not. Then the suffix $\mathtt{0}^{p+(k-1)j-ki}$ distinguishes x and y:

$$\begin{aligned} xz &= \mathtt{0}^i\,\mathtt{0}^{p+(k-1)j-ki} = \mathtt{0}^{p+(k-1)(j-i)} \in L && \text{because } p+(k-1)(j-i) \text{ is prime;} \\ yz &= \mathtt{0}^j\,\mathtt{0}^{p+(k-1)j-ki} = \mathtt{0}^{p+k(j-i)} \notin L && \text{because } p+k(j-i) \text{ is not prime.} \end{aligned}$$

(Because $i < j$ and $i < p$, the suffix $\mathtt{0}^{p+(k-1)j-ki} = \mathtt{0}^{(p-i)+(k-1)(j-i)}$ has positive length and therefore *actually exists!*) We conclude that $\mathtt{0}^*$ is indeed a fooling set for L, which implies that L is not regular. □

One natural question that many students ask is "How did you come up with that fooling set?" Perhaps the simplest rule of thumb is that for most languages L—in particular, for almost all languages that students are asked to prove non-regular on homeworks or exams—either some simple regular language like $\mathtt{0}^*$ or $\mathtt{10}^*\mathtt{1}$ is a fooling set, or the language L itself is a fooling set. (Of course, there are well-engineered counterexamples.)

★3.9 The Myhill-Nerode Theorem

The fooling set technique implies a *necessary* condition for a language to be accepted by a DFA—the language must have no infinite fooling sets. In fact, this condition is also *sufficient*. The following powerful theorem was first proved by Anil Nerode in 1958, strengthening a 1957 result of John Myhill.[1]

The Myhill-Nerode Theorem. *For any language L, the following are equal:*

[1]Myhill considered the finer equivalence relation $x \sim_L y$, meaning $wxz \in L$ if and only if $wyz \in L$ for all strings w and z, and proved that L is regular if and only if $\sim_L$ defines a finite number of equivalence classes. Like most of Myhill's early automata research, this result appears in an unpublished Air Force technical report. The modern Myhill-Nerode theorem appears (in an even more general form) as a minor lemma in Nerode's 1958 paper, which (not surprisingly) does not cite Myhill.

(a) the minimum number of states in a DFA that accepts L,
(b) the maximum size of a fooling set for L, and
(c) the number of equivalence classes of $\equiv_L$.
In particular, L is accepted by a DFA if and only if every fooling set for L is finite.

Proof: Let L be an arbitrary language.

We have already proved that the size of any fooling set for L is at most the number of states in any DFA that accepts L, so (a)≤(b). It also follows directly from the definitions that $F \subseteq \Sigma^*$ is a fooling set for L if and only if F contains at most one string in each equivalence class of $\equiv_L$; thus, (b)=(c). We complete the proof by showing that (a)≥(c).

We have already proved that if $\equiv_L$ has an infinite number of equivalence classes, there is no DFA that accepts L, so assume that the number of equivalence classes is finite. For any string w, let $[w]$ denote its equivalence class. We define a DFA $M_\equiv = (\Sigma, Q, s, A, \delta)$ as follows:

$$\begin{aligned} Q &:= \left\{[w] \;\middle|\; w \in \Sigma^*\right\} \\ s &:= [\varepsilon] \\ A &:= \left\{[w] \;\middle|\; w \in L\right\} \\ \delta([w], a) &:= [w \bullet a] \end{aligned}$$

We claim that this DFA accepts the language L; this claim completes the proof of the theorem.

But before we can prove anything about this DFA, we first need to verify that it is actually well-defined. Let x and y be two strings such that $[x] = [y]$. By definition of L-equivalence, for any string z, we have $xz \in L$ if and only if $yz \in L$. It immediately follows that for any symbol $a \in \Sigma$ and any string z', we have $xaz' \in L$ if and only if $yaz' \in L$. Thus, by definition of L-equivalence, we have $[xa] = [ya]$ for every symbol $a \in \Sigma$. We conclude that the function δ is indeed well-defined.

An easy inductive proof implies that $\delta^*([\varepsilon], x) = [x]$ for every string x. Thus, M accepts string x if and only if $[x] = [w]$ for some string $w \in L$. But if $[x] = [w]$, then by definition (setting $z = \varepsilon$), we have $x \in L$ if and only if $w \in L$. So M accepts x if and only if $x \in L$. In other words, M accepts L, as claimed, so the proof is complete. □

⋆3.10 Minimal Automata

Given a DFA $M = (\Sigma, Q, s, A, \delta)$, suppose we want to find another DFA $M' = (\Sigma, Q', s', A', \delta')$ with the fewest possible states that accepts the same language. In this final section, we describe an efficient algorithm to minimize DFAs, first described (in slightly different form) by Edward Moore in 1956. We analyze the running time of Moore's in terms of two parameters: $n = |Q|$ and $\sigma = |\Sigma|$.

In the preprocessing phase, we find and remove any states that cannot be reached from the start state s; this filtering can be performed in $O(n\sigma)$ time using any graph traversal algorithm. So from now on we assume that all states are reachable from s.

Now define two states p and q in the trimmed DFA to be ***distingusiable***, written $\boldsymbol{p \not\sim q}$, if at least one of the following conditions holds:

- $p \in A$ and $q \notin A$,
- $p \notin A$ and $q \in A$, or
- $\delta(p, a) \not\sim \delta(q, a)$ for some $a \in \Sigma$.

Equivalently, $p \not\sim q$ if and only if there is a string z such that exactly one of the states $\delta^*(p,z)$ and $\delta^*(q,z)$ is accepting. (Sound familiar?) Intuitively, the main algorithm assumes that all states are equivalent until proven otherwise, and then repeatedly looks for state pairs that can be proved distinguishable.

The main algorithm maintains a two-dimensional table, indexed by the states, where $Dist[p,q] = \text{True}$ indicates that we have proved states p and q are distinguishable. Initially, for all states p and q, we set $Dist[p,q] \leftarrow \text{True}$ if $p \in A$ and $q \notin A$ or vice versa, and $Dist[p,q] = \text{False}$ otherwise. Then we repeatedly consider each pair of states and each symbol to find more distinguished pairs, until we make a complete pass through the table without modifying it. The table-filling algorithm can be summarized as follows:[2]

```
MinDFATable(Σ, Q, s, A, δ):
  for all p ∈ Q
      for all q ∈ Q
          if (p ∈ A and q ∉ A) or (p ∉ A and q ∈ A)
              Dist[p,q] ← True
          else
              Dist[p,q] ← False
  notdone ← True
  while notdone
      notdone ← False
      for all p ∈ Q
          for all q ∈ Q
              if Dist[p,q] = False
              for all a ∈ Σ
                  if Dist[δ(p,a), δ(q,a)]
                      Dist[p,q] ← True
                      notdone ← True
  return Dist
```

The algorithm must eventually halt, because there are only a finite number of entries in the table that can be marked. In fact, the main loop is guaranteed to terminate after at most n iterations, which implies that the entire algorithm runs in $\boldsymbol{O(\sigma n^3)}$ time. Once the table is filled, any two states p and q such that $Dist(p,q) = \text{False}$ are equivalent and can be merged into a single state. The remaining details of constructing the minimized DFA are straightforward.

★★★ Need to prove that the main loop terminates in at most n iterations.

With more care, Moore's minimization algorithm can be modified to run in $O(\sigma n^2)$ time. A faster DFA minimization algorithm, due to John Hopcroft, runs in $O(\sigma n \log n)$ time.

---

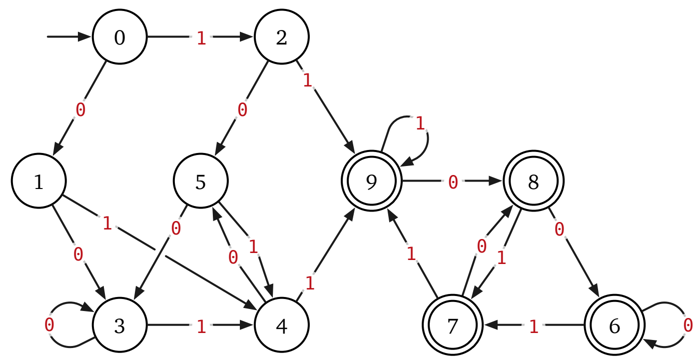
[2]More experienced readers should become queasy at the mere suggestion that any algorithm merely *fills in a table*, as opposed to *evaluating a recurrence*. This algorithm is no exception. Consider the boolean function $Dist(p,q,k)$, which equals True if and only if p and q can be distinguished by some string of length at most k. This function obeys the following recurrence:

$$Dist(p,q,k) = \begin{cases} (p \in A) \oplus (q \in A) & \text{if } k = 0, \\ Dist(p,q,k-1) \vee \bigvee_{a \in \Sigma} Dist\big(\delta(p,a), \delta(q,a), k-1\big) & \text{otherwise.} \end{cases}$$

Moore's "table-filling" algorithm is just a space-efficient dynamic programming algorithm to evaluate this recurrence.

Example

To get a better idea how this algorithm works, let's visualize the algorithm running on our earlier brute-force DFA for strings containing the substring `11`. This DFA has four unreachable states: (FALSE, `11`), (TRUE, ε), (TRUE, `0`), and (TRUE, `1`). We remove these states, and relabel the remaining states for easier reference. (In an actual implementation, the states would almost certainly be represented by indices into an array anyway, not by mnemonic labels.)



Our brute-force DFA for strings containing the substring `11`, after removing all four unreachable states

The main algorithm initializes (the bottom half of) a 10×10 table as follows. (In the implementation, cells marked $\not\sim$ have value TRUE and blank cells have value FALSE.)

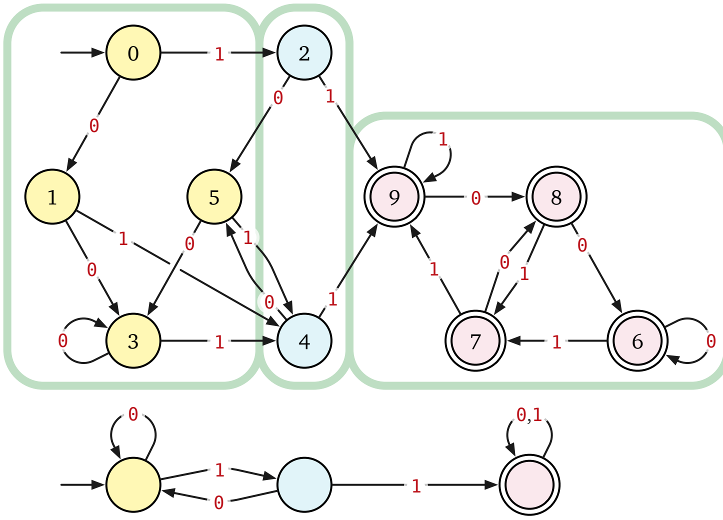
	0	1	2	3	4	5	6	7	8
1									
2									
3									
4									
5									
6	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
7	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
8	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
9	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			

In the first iteration of the main loop, the algorithm discovers several distinguishable pairs of states. For example, the algorithm sets $Dist[0,2] \leftarrow$ TRUE because $Dist[\delta(0,\texttt{1}),\delta(2,\texttt{1})] = Dist[2,9] =$ TRUE. After the iteration ends, the table looks like this:

	0	1	2	3	4	5	6	7	8
1									
2	$\not\sim$	$\not\sim$							
3			$\not\sim$						
4	$\not\sim$	$\not\sim$		$\not\sim$					
5			$\not\sim$		$\not\sim$				
6	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
7	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
8	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			
9	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$	$\not\sim$			

The second iteration of the while loop makes no further changes to the table—We got lucky!—so the algorithm terminates.

The final table implies that the states of our trimmed DFA fall into exactly three equivalence classes: $\{0,1,3,5\}$, $\{2,4\}$, and $\{6,7,8,9\}$. Replacing each equivalence class with a single state gives us the three-state DFA that we already discovered.



Equivalence classes of states in the trimmed DFA, and the resulting minimal equivalent DFA.

Exercises

1. For each of the following languages in $\{0,1\}^*$, describe a deterministic finite-state machine that accepts that language. There are infinitely many correct answers for each language. "Describe" does not necessarily mean "draw".

   (a) Only the string `0110`.

   (b) Every string except `0110`.

   (c) Strings that contain the substring `0110`.

   (d) Strings that do not contain the substring `0110`.

   *(e) Strings that contain an even number of occurrences of the substring `0110`. (For example, this language contains the strings `0110110` and `01011`.)

   (f) Strings that contain the *subsequence* `0110`.

   (g) Strings that do not contain the *subsequence* `0110`.

   (h) Strings that contain an even number of `1`s and an odd number of `0`s.

   (i) Strings that represent a number divisible by 7 in binary.

   (j) Strings whose reversals represent a number divisible by 7 in binary.

   (k) Strings in which the substrings `01` and `10` appear the same number of times.

   (l) Strings such that in every prefix, the number of `0`s and the number of `1`s differ by at most 1.

   (m) Strings such that in every prefix, the number of `0`s and the number of `1`s differ by at most 4.

   (n) Strings that end with 0^{10} = `0000000000`.

(o) Strings in which the number of 1s is even, the number of 0s is divisible by 3, the overall length is divisible by 5, the binary value is divisible by 7, the binary value of the reversal is divisible by 11, and does not contain thirteen 1s in a row. *[Hint: This is more tedious than difficult.]*

2. (a) Let $L \subseteq \mathtt{0}^*$ be an arbitrary *unary* language. Prove that L^* is regular.

   (b) Prove that there is a binary language $L \subseteq (\mathtt{0}+\mathtt{1})^*$ such that L^* is not regular.

3. Describe and analyze algorithms for the following problems. In each case, the input is a DFA M over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$.

   (a) Does M accept any string whose length is a multiple of 5?

   (b) Does M accept every string that represents a number divisible by 7 in binary?

   (c) Does M accept an infinite number of strings containing an odd number of 0s?

   (d) Does M accept a finite number of strings that contain the substring 0110110 and whose length is divisible by five?

   (e) Does M accept *only* strings whose lengths are perfect squares?

   (f) Does M accept any string whose length is *composite*?

   ★(g) Does M accept any string whose length is *prime*?

★★★ Move these to the graph traversal notes?

4. Prove that each of the following languages cannot be accepted by a DFA.

   (a) $\{\mathtt{0}^{n^2} \mid n \geq 0\}$

   (b) $\{\mathtt{0}^{n^3} \mid n \geq 0\}$

   (c) $\{\mathtt{0}^{f(n)} \mid n \geq 0\}$, where $f(n)$ is *any* fixed polynomial in n with degree at least 2.

   (d) $\{\mathtt{0}^n \mid n \text{ is composite}\}$

   (e) $\{\mathtt{0}^n\mathtt{1}\mathtt{0}^n \mid n \geq 0\}$

   (f) $\{\mathtt{0}^i\mathtt{1}^j \mid i \neq j\}$

   (g) $\{\mathtt{0}^i\mathtt{1}^j \mid i < 3j\}$

   (h) $\{\mathtt{0}^i\mathtt{1}^j \mid i \text{ and } j \text{ are relatively prime}\}$

   (i) $\{\mathtt{0}^i\mathtt{1}^j \mid j - i \text{ is a perfect square}\}$

   (j) $\{w\#w \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (k) $\{ww \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (l) $\{w\#\mathtt{0}^{|w|} \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (m) $\{w\mathtt{0}^{|w|} \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (n) $\{xy \mid w, x \in (\mathtt{0}+\mathtt{1})^* \text{ and } |x| = |y| \text{ but } x \neq y\}$

   (o) $\{\mathtt{0}^m\mathtt{1}^n\mathtt{0}^{m+n} \mid m, n \geq 0\}$

(p) $\{0^m 1^n 0^{mn} \mid m, n \geq 0\}$

(q) Strings in which the substrings 00 and 11 appear the same number of times.

(r) The set of all palindromes in $(0+1)^*$ whose length is divisible by 7.

(s) $\{w \in (0+1)^* \mid w$ is the binary representation of a perfect square$\}$

★(t) $\{w \in (0+1)^* \mid w$ is the binary representation of a prime number$\}$

5. For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, either describe a DFA that accepts the language or prove that no such DFA exists. Recall that Σ^+ denotes the set of all *nonempty* strings over Σ. *[Hint: Believe it or not, most of these languages **can** be accepted by DFAs.]*

(a) $\left\{0^n w 1^n \mid w \in \Sigma^* \text{ and } n \geq 0\right\}$

(b) $\left\{0^n 1^n w \mid w \in \Sigma^* \text{ and } n \geq 0\right\}$

(c) $\left\{w 0^n 1^n x \mid w, x \in \Sigma^* \text{ and } n \geq 0\right\}$

(d) $\left\{0^n w 1^n x \mid w, x \in \Sigma^* \text{ and } n \geq 0\right\}$

(e) $\left\{0^n w 1 x 0^n \mid w, x \in \Sigma^* \text{ and } n \geq 0\right\}$

(f) $\left\{wxw \mid w, x \in \Sigma^*\right\}$

(g) $\left\{wxw \mid w, x \in \Sigma^+\right\}$

(h) $\left\{wxw^R \mid w, x \in \Sigma^+\right\}$

(i) $\left\{wwx \mid w, x \in \Sigma^+\right\}$

(j) $\left\{ww^R x \mid w, x \in \Sigma^+\right\}$

(k) $\left\{wxwy \mid w, x, y \in \Sigma^+\right\}$

(l) $\left\{wxw^R y \mid w, x, y \in \Sigma^+\right\}$

(m) $\left\{xwwy \mid w, x, y \in \Sigma^+\right\}$

(n) $\left\{xww^R y \mid w, x, y \in \Sigma^+\right\}$

(o) $\left\{wxxw \mid w, x \in \Sigma^+\right\}$

★(p) $\left\{wxw^R x \mid w, x \in \Sigma^+\right\}$

Caveat lector! This is the first edition of this lecture note. Some topics are incomplete, and there are almost certainly a few serious errors. Please send bug reports and suggestions to jeffe@illinois.edu.

Nothing is better than simplicity . . . .
nothing can make up for excess or for the lack of definiteness.
— Walt Whitman, Preface to *Leaves of Grass* (1855)

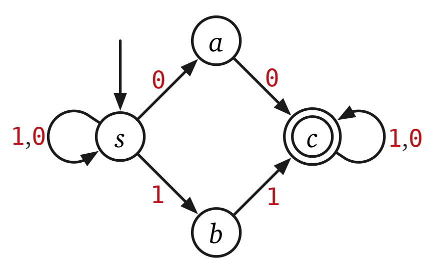
Freedom of choice
Is what you got.
Freedom from choice
Is what you want.
— Devo, "Freedom of Choice", *Freedom of Choice* (1980)

Nondeterminism means never having to say you are wrong.
— BSD 4.3 fortune(6) file (c.1985)

4 Nondeterminism

4.1 Nondeterministic State Machines

The following diagram shows something that looks like a finite-state machine over the alphabet $\{0, 1\}$, but on closer inspection, it is not consistent with our earlier definitions. On one hand, there are two transitions out of s for each input symbol. On the other hand, states a and b are each missing an outgoing transition.



A nondeterministic finite-state automaton

Nevertheless, there is a sense in which this machine "accepts" the set of all strings that contain either 00 or 11 as a substring. Imagine that when the machine reads a symbol in state s, it makes a choice about which transition to follow. If the input string contains the substring 00, then it is *possible* for the machine to end in the accepting state c, by *choosing* to move into state a when it reads a 0 immediately before another 0. Similarly, if the input string contains the substring 11, it is *possible* for the machine to end in the accepting state c. On the other hand, if the input string does not contain either 00 or 11—or in other words, if the input alternates between 0 and 1—there are no choices that lead the machine to the accepting state. If the machine incorrectly chooses to transition to state a and then reads a 1, or transitions to b and then reads 0, it explodes; the only way to avoid an explosion is to stay in state s.

This object is an example of a ***nondeterministic finite-state automaton***, or ***NFA***, so named because its behavior is not uniquely *determined* by the input string. Formally, every NFA has five components:

- An arbitrary finite set $\mathbf{\Sigma}$, called the ***input alphabet***.
- Another arbitrary finite set $\boldsymbol{Q}$, whose elements are called ***states***.
- An arbitrary ***transition*** function $\boldsymbol{\delta}: Q \times \Sigma \to 2^Q$.
- A ***start state*** $\boldsymbol{s} \in Q$.
- A subset $\boldsymbol{A} \subseteq Q$ of ***accepting states***.

The only difference from the formal definition of *deterministic* finite-state automata is the domain of the transition function. In a DFA, the transition function always returns a single state; in an NFA, the transition function returns a *set* of states, which could be empty, or all of Q, or anything in between.

Just like DFAs, the behavior of an NFA is governed by an ***input string*** $w \in \Sigma^*$, which the machine reads one symbol at a time, from left to right. Unlike DFAs, however, an NFA does not maintain a single current state, but rather a *set* of current states. Whenever the NFA reads a symbol a, its set of current states changes from C to $\bigcup_{q \in C} \delta(q,a)$. After all symbols have been read, the NFA ***accepts*** w if its current state set contains *at least one* accepting state and ***rejects*** w otherwise. In particular, if the set of current states ever becomes empty, it will stay empty forever, and the NFA will reject.

More formally, we define the function $\delta^*: Q \times \Sigma^* \to 2^Q$ that transitions on *strings* as follows:

$$\delta^*(q,w) := \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \delta(q,a)} \delta^*(r,x) & \text{if } w = ax. \end{cases}$$

The NFA $(Q, \Sigma, \delta, s, A)$ ***accepts*** $w \in \Sigma^*$ if and only if $\delta^*(s,w) \cap A \neq \varnothing$.

We can equivalently define an NFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ. We no longer require that every vertex has exactly one outgoing edge with each label; it may have several such edges or none. An NFA accepts a string w if the graph contains *at least one* walk from the start state to an accepting state whose label is w.

4.2 Intuition

There are at least three useful ways to think about non-determinism.

Clairvoyance. Whenever an NFA reads symbol a in state q, it *chooses* the next state from the set $\delta(q,a)$, always *magically* choosing a state that leads to the NFA accepting the input string, unless no such choice is possible. As the BSD fortune file put it, "Nondeterminism means never having to say you're wrong."[1] Of course real machines can't actually look into the future; that's why I used the word "magic".

Parallel threads. An arguably more "realistic" view is that when an NFA reads symbol a in state q, it spawns an independent execution thread for each state in $\delta(q,a)$. In particular, if $\delta(q,a)$ is empty, the current thread simply dies. The NFA accepts if *at least one* thread is in an accepting state after it reads the last input symbol.

Equivalently, we can imagine that when an NFA reads symbol a in state q, it branches into several parallel universes, one for each state in $\delta(q,a)$. If $\delta(q,a)$ is empty, the NFA destroys the

[1] This sentence is a riff on a horrible aphorism that was (sadly) popular in the US in the 70s and 80s. Fortunately, everyone seems to have forgotten the original saying, except for that one time it was parodied on the Simpsons.

universe (including itself). Similarly, if the NFA finds itself in a non-accepting state when the input ends, the NFA destroys the universe. Thus, when the input is gone, only universes in which the NFA somehow chose a path to an accept state still exist. One slight disadvantage of this metaphor is that if an NFA reads a string that is not in its language, it destroys *all* universes.

Proofs/oracles. Finally, we can treat NFAs not as a mechanism for *computing* something, but only as a mechanism for *verifying proofs*. If we want to *prove* that a string w contains one of the suffixes `00` or `11`, it suffices to demonstrate a single walk in our example NFA that starts at s and ends at c, and whose edges are labeled with the symbols in w. Equivalently, whenever the NFA faces a nontrivial choice, the prover can simply tell the NFA which state to move to next.

This intuition can be formalized as follows. Consider a *deterministic* finite state machine whose input alphabet is the product $\Sigma \times \Omega$ of an ***input*** alphabet Σ and an ***oracle*** alphabet Ω. Equivalently, we can imagine that this DFA reads simultaneously from two strings of the same length: the *input* string w and the *oracle* string ω. In either formulation, the transition function has the form $\delta : Q \times \Sigma \times \Omega \to Q$. As usual, this DFA accepts the pair $(w, \omega) \in (\Sigma \times \Gamma)^*$ if and only if $\delta^*(s, w, \omega) \in A$. Finally, M ***nondeterministically accepts*** the string $w \in \Sigma^*$ if there is an oracle string $\omega \in \Omega^*$ with $|\omega| = |w|$ such that $(w, \omega) \in L(M)$.

4.3 ε-Transitions

It is fairly common for NFAs to include so-called ***ε-transitions***, which allow the machine to change state without reading an input symbol. An NFA with ε-transitions accepts a string w if and only if there is a sequence of transitions $s \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_\ell} q_\ell$ where the final state q_ℓ is accepting, each a_i is either ε or a symbol in Σ, and $a_1 a_2 \cdots a_\ell = w$.

More formally, the transition function in an NFA with ε-transitions has a slightly larger domain $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. The ***$\varepsilon$-reach*** of a state $q \in Q$ consists of all states r that satisfy one of the following conditions:

- $r = q$
- $r \in \delta(q', \varepsilon)$ for some state q' in the ε-reach of q.

In other words, r is in the ε-reach of q if there is a (possibly empty) sequence of ε-transitions leading from q to r. Now we redefine the extended transition function $\delta^* : Q \times \Sigma^* \to 2^Q$, which transitions on arbitrary strings, as follows:

$$\delta^*(q, w) := \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \varepsilon\text{-reach}(q)} \; \bigcup_{r' \in \delta(r, a)} \delta^*(r', x) & \text{if } w = ax. \end{cases}$$

As usual, the modified NFA accepts a string w if and only if $\delta^*(s, w) \cap A \neq \varnothing$.

Given an NFA $M = (\Sigma, Q, s, A, \delta)$ with ε-transitions, we can easily construct an equivalent NFA $M' = (\Sigma, Q', s', A', \delta')$ without ε-transitions as follows:

$$\begin{aligned} Q' &:= Q \\ s' &= s \\ A' &= \big\{ q \in Q \;\big|\; \varepsilon\text{-reach}(q) \cap A \neq \varnothing \big\} \\ \delta'(q, a) &= \bigcup_{r \in \varepsilon\text{-reach}(q)} \delta(r, a) \end{aligned}$$

Straightforward definition-chasing implies that M and M' accept exactly the same language. Thus, whenever we reason about or design NFAs, we are free to either allow or forbid ε-transitions, whichever is more convenient for the task at hand.

4.4 Kleene's Theorem

We are now finally in a position to prove the following fundamental fact, first observed by Steven Kleene:

Theorem 4.1. *A language L can be described by a regular expression if and only if L is the language accepted by a DFA.*

We will prove Kleene's fundamental theorem in four stages:

- Every DFA can be transformed into an equivalent NFA.
- Every NFA can be transformed into an equivalent DFA.
- Every regular expression can be transformed into an NFA.
- Every NFA can be transformed into an equivalent regular expression.

The first of these four transformations is completely trivial; a DFA is just a special type of NFA where the transition function always returns a single state. Unfortunately, the other three transformations require a bit more work.

4.5 DFA from NFAs: The Subset Construction

In the parallel-thread model of NFA execution, an NFA does not have a single current state, but rather a *set* of current states. The evolution of this set of states is *determined* by a modified transition function $\delta' \colon 2^Q \times \Sigma \to 2^Q$, defined by setting $\delta'(P, a) := \bigcup_{p \in P} \delta(p, a)$ for any set of states $P \subseteq Q$ and any symbol $a \in \Sigma$. When the NFA finishes reading its input string, it accepts if and only if the current set of states intersects the set A of accepting states.

This formulation makes the NFA completely deterministic! We have just shown that any NFA $M = (\Sigma, Q, s, A, \delta)$ is equivalent to a DFA $M' = (\Sigma, Q', s', A', \delta')$ defined as follows:

$$\begin{aligned} Q' &:= 2^Q \\ s' &:= \{s\} \\ A' &:= \left\{ S \subseteq Q \,\middle|\, S \cap A \neq \varnothing \right\} \\ \delta'(q', a) &:= \bigcup_{p \in q'} \delta(p, a) \qquad \text{for all } q' \subseteq Q \text{ and } a \in \Sigma. \end{aligned}$$

Similarly, any NFA with ε-transitions is equivalent to a DFA with the transition function

$$\delta'(q', a) = \bigcup_{p \in q'} \; \bigcup_{r \in \varepsilon\text{-reach}(p)} \delta(r, a)$$

for all $q' \subseteq Q$ and $a \in \Sigma$. This conversion from NFA to DFA is often called the ***subset construction***, but that name is somewhat misleading; it's not a "construction" so much as a change in perspective.

One disadvantage of this "construction" is that it usually leads to DFAs with far more states than necessary, in part because most of those states are unreachable. These unreachable states can be avoided by constructing the DFA incrementally, essentially by performing a breadth-first search of the DFA graph, starting at its start state.

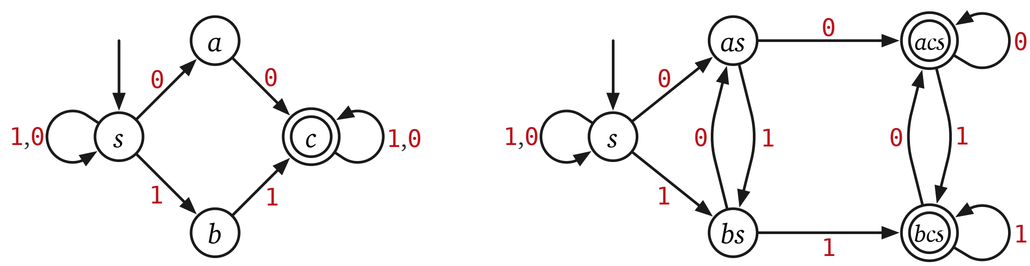
To execute this algorithm by hand, we prepare a table with $|\Sigma| + 3$ columns, with one row for each DFA state we discover. In order, these columns record the following information:

- The DFA state (as a subset of NFA states)
- The ε-reach of the corresponding subset of NFA states
- Whether the DFA state is accepting (that is, whether the ε-reach intersects A)
- The output of the transition function for each symbol in Σ.

We start with DFA-state s in the first row and first column. Whenever we discover an unexplored state in one of the last $|\Sigma|$ columns, we copy it to the left column in a new row.

For example, given the NFA on the first page of this note, this incremental algorithm produces the following table, yielding a five-state DFA. For this example, the second column is redundant, because the NFA has no ε-transitions, but we will see another example with ε-transitions in the next subsection. To simplify notation, we write each set of states as a simple string, omitting braces and commas.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	s		as	bs
as	as		acs	bs
bs	bs		as	bcs
acs	acs	✓	acs	bcs
bcs	bcs	✓	acs	bcs



Our example NFA, and the output of the incremental subset construction for that NFA.

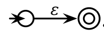
4.6 NFAs from Regular Expressions: Thompson's Algorithm

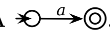
Lemma 4.2. *Every regular language is accepted by a non-deterministic finite automaton.*

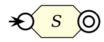
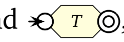
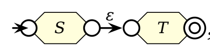
Proof: In fact, we will prove the following stronger claim: Every regular language is accepted by an NFA *with exactly one accepting state, which is different from its start state*. The following construction was first described by Ken Thompson in 1968. Thompson's algorithm actually proves a stronger statement: For any regular language L, there is an NFA that accepts L that has exactly one accepting state t, which is distinct from the starting state s.

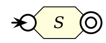
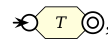
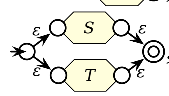
Let R be an arbitrary regular expression over an arbitrary finite alphabet Σ. Assume that for any sub-expression S of R, the language described by S is accepted by an NFA with one accepting state distinct from its start state, which we denote pictorially by S. There are six cases to consider—three base cases and three recursive cases—mirroring the recursive definition of a regular expression.

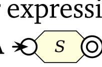
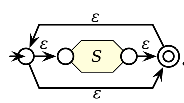
- If $R = \varnothing$, then $L(R) = \varnothing$ is accepted by the empty NFA: .

- If $R = \varepsilon$, then $L(R) = \{\varepsilon\}$ is accepted by the NFA .

- If $R = a$ for some symbol $a \in \Sigma$, then $L(R) = \{a\}$ is accepted by the NFA . (The case where R is a single *string* with length greater than 1 reduces to the single-symbol case by concatenation, as described in the next case.)

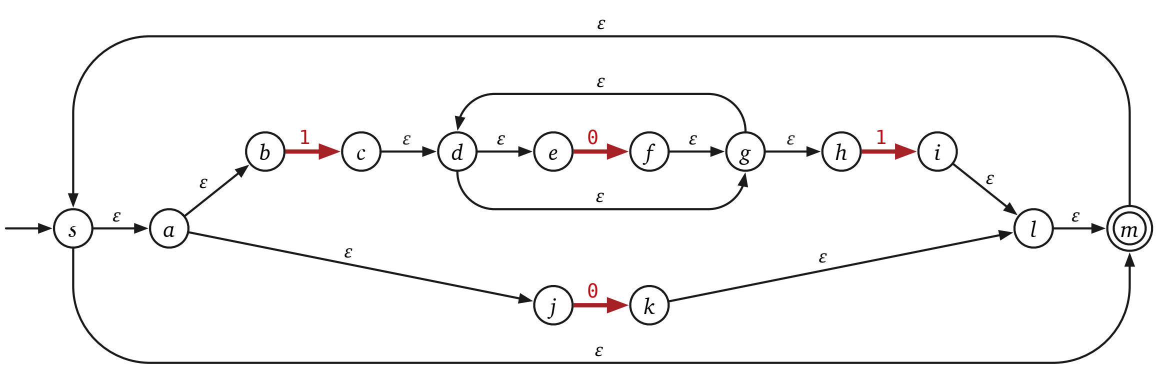
- Suppose $R = ST$ for some regular expressions S and T. The inductive hypothesis implies that the languages $L(S)$ and $L(T)$ are accepted by NFAs  and , respectively. Then $L(R) = L(ST) = L(S) \bullet L(T)$ is accepted by the NFA , built by connecting the two component NFAs in series.

- Suppose $R = S + T$ for some regular expressions S and T. The inductive hypothesis implies that the language $L(S)$ and $L(T)$ are accepted by NFAs  and , respectively. Then $L(R) = L(S + T) = L(S) \cup L(T)$ is accepted by the NFA , built by connecting the two component NFAs in parallel with new start and accept states.

- Finally, suppose $R = S^*$ for some regular expression S. The inductive hypothesis implies that the language $L(S)$ is accepted by an NFA . Then the language $L(R) = L(S^*) = L(S)^*$ is accepted by the NFA .

In all cases, the language $L(R)$ is accepted by an NFA with one accepting state, which is different from its start state, as claimed. □

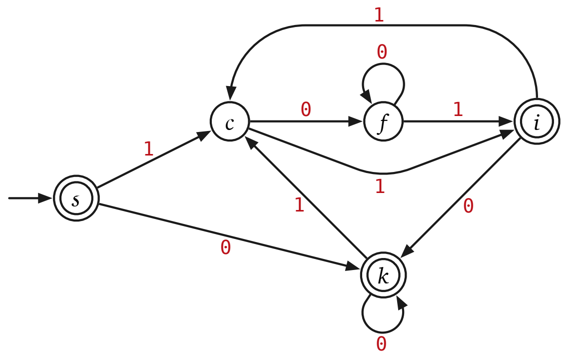
As an example, given the regular expression $(0 + 10^*1)^*$ of strings containing an even number of 1s, Thompson's algorithm produces a 14-state NFA shown on the next page. As this example shows, Thompson's algorithm tends to produce NFAs with many redundant states. Fortunately, just as there are for DFAs, there are algorithms that can reduce any NFA to an equivalent NFA with the smallest possible number of states.



The NFA constructed by Thompson's algorithm for the regular expression $(0 + 10^*1)^*$.
The four non-ε-transitions are drawn with with bold red arrows for emphasis.

Interestingly, applying the incremental subset algorithm to Thompson's NFA tends to yield a DFA with relatively *few* states, in part because the states in Thompson's NFA tend to have large ε-reach, and in part because relatively few of those states are the targets of non-ε-transitions. Starting with the NFA shown above, for example, the incremental subset construction yields a DFA for the language $(0 + 10^*1)^*$ with just five states:

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', \texttt{0})$	$\delta'(q', \texttt{1})$
s	$sabjm$	✓	k	c
k	$sabjklm$	✓	k	c
c	$cdegh$		f	i
f	$degh$		f	i
i	$sabjilm$	✓	k	c



The DFA computed by the incremental subset algorithm from Thompson's NFA for $(\texttt{0} + \texttt{10}^*\texttt{1})^*$.

This DFA can be further simplified to just two states, by observing that all three accepting states are equivalent, and that both non-accepting states are equivalent. But still, five states is pretty good, especially compared with the $2^{13} = 8096$ states that the naïve subset construction would yield!

*4.7 NFAs from Regular Expressions: Glushkov's Algorithm

Thompson's algorithm is actually a modification of an earlier algorithm, which was independently discovered by Robert McNaughton and Hisao Yamada in 1960 and by V. I. Glushkov in 1961. Given a regular expression containing n symbols (not counting the parentheses and pluses and stars), Glushkov's algorithm produces an NFA with exactly $n+1$ states.

Glushkov's algorithm combines six functions on regular expressions:

- *index*(R) is the regular expression obtained by replacing the symbols in R with the integers 1 through n, in order from left to right. For example, *index*$((\texttt{0} + \texttt{10}^*\texttt{1})^*) = (1 + 23^*4)^*$.
- *symbols*(R) denotes the string obtained by removing all non-symbols from R. For example, *symbols*$((\texttt{0} + \texttt{10}^*\texttt{1})^*) = \texttt{0101}$.
- *has-*$\varepsilon(R)$ is TRUE if $\varepsilon \in L(R)$ and FALSE otherwise.
- *first*(R) is the set of all initial symbols of strings in $L(R)$.
- *last*(R) is the set of all final symbols of strings in $L(R)$.
- *middle*(R) is the set of all pairs (a, b) such that ab is a substring of some string in $L(R)$.

The last four functions obey the following recurrences:

$$\begin{aligned}
\textit{has-}\varepsilon(\varnothing) &= \varnothing \\
\textit{has-}\varepsilon(w) &= \begin{cases} \text{TRUE} & \text{if } w = \varepsilon \\ \text{FALSE} & \text{otherwise} \end{cases} \\
\textit{has-}\varepsilon(S+T) &= \textit{has-}\varepsilon(S) \vee \textit{has-}\varepsilon(T) \\
\textit{has-}\varepsilon(ST) &= \textit{has-}\varepsilon(S) \wedge \textit{has-}\varepsilon(T) \\
\textit{has-}\varepsilon(S^*) &= \text{TRUE}
\end{aligned}$$

$$\begin{aligned}
\textit{first}(\varnothing) &= \varnothing \\
\textit{first}(w) &= \begin{cases} \varnothing & \text{if } w = \varepsilon \\ \{a\} & \text{if } w = ax \end{cases} \\
\textit{first}(S+T) &= \textit{first}(S) \cup \textit{first}(T) \\
\textit{first}(ST) &= \begin{cases} \textit{first}(S) \cup \textit{first}(T) & \text{if } \textit{has-}\varepsilon(S) \\ \textit{first}(T) & \text{otherwise} \end{cases} \\
\textit{first}(S^*) &= \textit{first}(S)
\end{aligned}$$

$$\begin{aligned}
\textit{last}(\varnothing) &= \varnothing \\
\textit{last}(w) &= \begin{cases} \varnothing & \text{if } w = \varepsilon \\ \{a\} & \text{if } w = xa \end{cases} \\
\textit{last}(S+T) &= \textit{last}(S) \cup \textit{last}(T) \\
\textit{last}(ST) &= \begin{cases} \textit{last}(S) \cup \textit{last}(T) & \text{if } \textit{has-}\varepsilon(T) \\ \textit{last}(T) & \text{otherwise} \end{cases} \\
\textit{last}(S^*) &= \textit{last}(S)
\end{aligned}$$

$$\begin{aligned}
\textit{middle}(\varnothing) &= \varnothing \\
\textit{middle}(w) &= \begin{cases} \varnothing & \text{if } |w| \le 1 \\ \{(a,b)\} \cup \textit{middle}(bx) & \text{if } w = abx \end{cases} \\
\textit{middle}(S+T) &= \textit{middle}(S) \cup \textit{middle}(T) \\
\textit{middle}(ST) &= \textit{middle}(S) \cup (\textit{last}(S) \times \textit{first}(T)) \cup \textit{middle}(T) \\
\textit{middle}(S^*) &= \textit{middle}(S) \cup (\textit{last}(S) \times \textit{first}(S))
\end{aligned}$$

For example, the set $\textit{middle}((1+23^*4)^*)$ can be computed recursively as follows. If we're doing this by hand, we can skip many of the steps in this derivation, because we know what the functions *first*, *middle*, *last*, and *has-ε* actually mean, but a mechanical recursive evaluation would necessarily evaluate every step.

$$\begin{aligned}
&\textit{middle}((1+23^*4)^*) \\
&\quad= \textit{middle}(1+23^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad= \textit{middle}(1) \cup \textit{middle}(23^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad= \varnothing \cup \textit{middle}(23^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad= \textit{middle}(2) \cup \big(\textit{last}(2) \times \textit{first}(3^*4)\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad= \varnothing \cup \big(\{2\} \times \textit{first}(3^*4)\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad= \big(\{2\} \times \big(\textit{first}(3^*) \cup \textit{first}(4)\big)\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad= \big(\{2\} \times \big(\textit{first}(3) \cup \textit{first}(4)\big)\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad= \big(\{2\} \times \{3,4\}\big) \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad= \big\{(2,3),(3,4)\big\} \cup \textit{middle}(3^*4) \cup \big(\textit{last}(1+23^*4) \times \textit{first}(1+23^*4)\big) \\
&\quad\ \vdots \\
&\quad= \big\{(1,1),(1,2),(2,3),(2,4),(3,3),(3,4),(4,1),(4,2)\big\}
\end{aligned}$$

Finally, given any regular expression R, Glushkov's algorithm constructs the NFA $M_R = (\Sigma, Q, s, A, \delta)$ that accepts exactly the language $L(R)$ as follows:

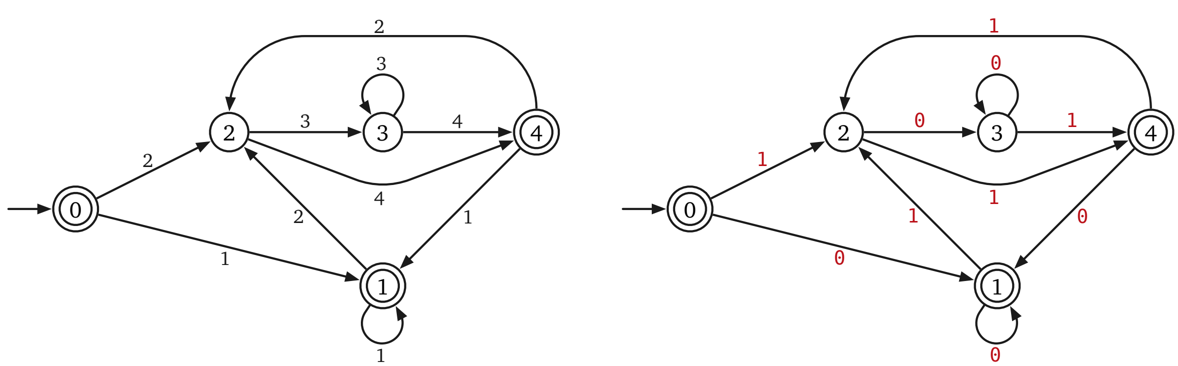
$$
\begin{aligned}
Q &= \big\{0, 1, \ldots, |\mathit{symbols}(R)|\big\} \\
s &= 0 \\
A &= \begin{cases} \{0\} \cup \mathit{last}(\mathit{index}(R)) & \text{if } \mathit{has}\text{-}\varepsilon(R) \\ \mathit{last}(\mathit{index}(R)) & \text{otherwise} \end{cases} \\
\delta(0, a) &= \big\{ j \in \mathit{first}(\mathit{index}(R)) \;\big|\; a = \mathit{symbols}(R)[j] \big\} \\
\delta(i, a) &= \big\{ j \;\big|\; (i, j) \in \mathit{middle}(\mathit{index}(R)) \text{ and } a = \mathit{symbols}(R)[j] \big\}
\end{aligned}
$$

There are a few natural ways to think about Glushkov's algorithm that are somewhat less impenetrable than the previous wall of definitions. One viewpoint is that Glushkov's algorithm first computes a *DFA* for the indexed regular expression *index*(R)—in fact, a DFA with the fewest possible states, except for an extra start state—and then replaces each index with the corresponding symbol in *symbols*(R) to get an NFA for the original expression R. Another useful observation is that Glushkov's NFA is identical to the result of removing all ε-transitions from Thompson's NFA for the same regular expression.

For example, given the regular expression $R = (\mathtt{0} + \mathtt{1}\mathtt{0}^*\mathtt{1})^*$, Glushkov's algorithm computes

$$
\begin{aligned}
\mathit{index}(R) &= (1 + 23^*4)^* \\
\mathit{symbols}(R) &= \mathtt{0101} \\
\mathit{has}\text{-}\varepsilon(R) &= \textsc{True} \\
\mathit{first}(\mathit{index}(R)) &= \{1, 2\} \\
\mathit{last}(\mathit{index}(R)) &= \{1, 4\} \\
\mathit{middle}(\mathit{index}(R)) &= \big\{(1,1), (1,2), (2,3), (2,4), (3,3), (3,4), (4,1), (4,2)\big\}
\end{aligned}
$$

and then constructs the following five-state NFA.

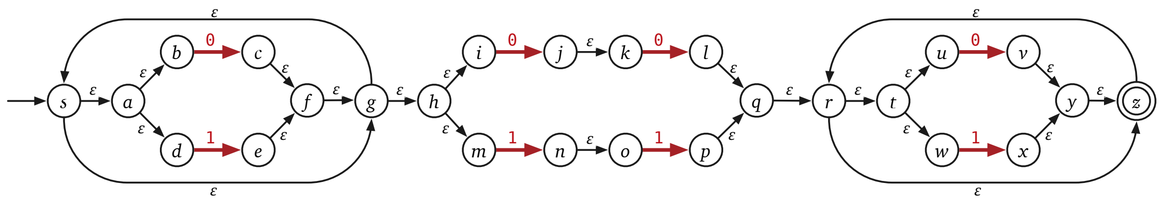


Glushkov's DFA for the index expression $(1 + 23^*4)^*$ and Glushkov's NFA for the regular expression $(\mathtt{0} + \mathtt{1}\mathtt{0}^*\mathtt{1})^*$.

Hey, look, Glushkov's algorithm actually gave us a DFA! In fact, it gave us *precisely* the same DFA that we constructed earlier by sending Thompson's NFA through the incremental subset algorithm! Unfortunately, that's just a coincidence; in general the output of Glushkov's algorithm is *not* deterministic. We'll see a more typical example in the next section.

4.8 Another Example

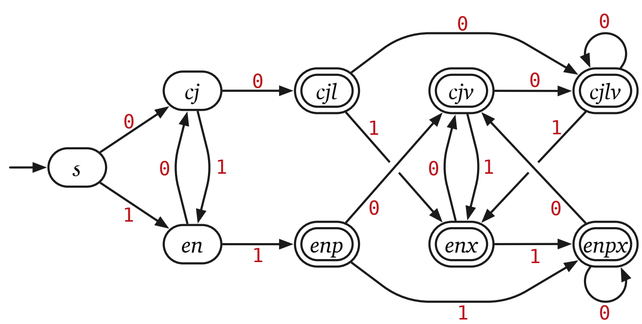
Here is another example of all the algorithms we've seen so far, starting with the regular expression $(0+1)^*(00+11)(0+1)^*$, which describes the language accepted by our very first example NFA. Thompson's algorithm constructs the following 26-state monster:



Thompson's NFA for the regular expression $(0+1)^*(00+11)(0+1)^*$

Given this NFA as input, the incremental subset construction computes the following table, leading to a DFA with just nine states. Yeah, the ε-reaches get a bit ridiculous; unfortunately, this *is* typical for Thompson's NFA.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	$sabdghim$		cj	en
cj	$sabdfghijkm$		cjl	en
en	$sabdfghmno$		cj	enp
cjl	$sabdfghijklmqrtuwz$	✓	$cjlv$	enx
enp	$sabdfghmnopqrtuwz$	✓	cjv	$enpx$
$cjlv$	$sabdfghijklmqrtuvwyz$	✓	$cjlv$	enx
enx	$sabdfghmnopqrtuwxyz$	✓	cjv	$enpx$
cjv	$sabdfghijkmrtuvwyz$	✓	$cjlv$	enx
$enpx$	$sabdfghmnopqrtuwxyz$	✓	cjv	$enpx$



The DFA computed by the incremental subset algorithm from Thompson's NFA for $(0+1)^*(00+11)(0+1)^*$.

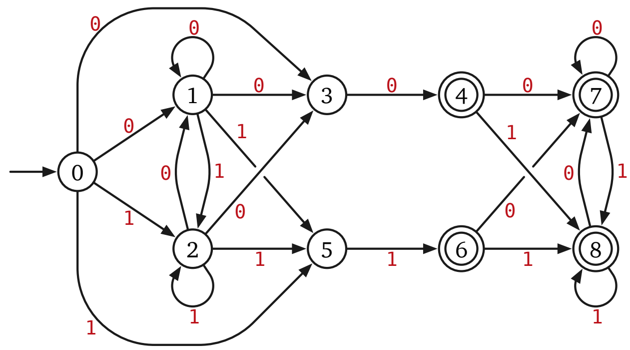
This DFA has far more states that necessary, intuitively because it keeps looking for 00 and 11 substrings even after it's already found one. After all, when Thompson's NFA finds a00 and 11 substring, it doesn't kill all the other parallel threads, because it *can't*. NFAs often have significantly fewer states than equivalent DFAs, but that efficiency also makes them kind of stupid.

Glushkov's algorithm recursively computes the following values for the same regular expression

$R = (0 + 1)^*(00 + 11)(0 + 1)^*$:

$$\begin{aligned}
\textit{index}(R) &= (1+2)^*(34+56)(7+8)^* \\
\textit{symbols}(R) &= 01001101 \\
\textit{has-}\varepsilon(R) &= \textsc{False} \\
\textit{first}(\textit{index}(R)) &= \{1,2,3,5\} \\
\textit{last}(\textit{index}(R)) &= \{4,6,7,8\} \\
\textit{middle}(\textit{index}(R)) &= \big\{(1,1),(1,2),(2,1),(2,2),(1,3),(1,5),(2,3),(2,5),(3,4), \\
&\qquad (5,6),(4,7),(4,8),(6,7),(6,8),(7,7),(7,8),(8,7),(8,8)\big\}
\end{aligned}$$

These values imply the nine-state NFA shown below. Careful readers should confirm that running the incremental subset construction on this NFA yields exactly the same DFA (with different state names) as it did for Thompson's NFA.



Glushkov's NFA for $(0 + 1)^*(00 + 11)(0 + 1)^*$

*4.9 Regular Expressions from NFAs: Han and Wood's Algorithm

The only component of Kleene's theorem we still have to prove is that every language accepted by a DFA or NFA is regular. As usual, it is actually easier to prove a stronger result. We consider a more general class of finite-state machines called ***expression automata***, introduced by Yo-Sub Han and Derick Wood in 2005.[2] Formally, an expression automaton consists of the following components:

- A finite set Σ called the ***input alphabet***
- Another finite set Q whose elements are called ***states***
- A ***start state*** $s \in Q$
- A single ***terminal state*** $t \in Q \setminus \{s\}$
- A ***transition function*** $R\colon (Q \setminus \{t\}) \times (Q \setminus \{s\}) \to \textit{Reg}(\Sigma)$, where $\textit{Reg}(\Sigma)$ is the set of regular expressions over Σ

Less formally, an expression automaton is a directed graph that includes a directed edge $p{\to}q$ labeled with a regular expression $R(p{\to}q)$, from *every* vertex p to *every* vertex q (including $q = p$), where by convention, we require that $R(q{\to}s) = R(t{\to}q) = \varnothing$ for every vertex q.

[2]Yo-Sub Han* and Derick Wood. The generalization of generalized automata: Expression automata. International Journal of Foundations of Computer Science 16(3):499–510, 2005.

We say that string w ***matches*** a transition $p \to q$ if w matches the regular expression $R(p \to q)$. In particular, if $R(p \to q) = \varnothing$, then no string matches $p \to q$. More generally, w matches a sequence of states $q_0 \to q_1 \to \cdots \to q_k$ if w matches the regular expression $R(q_0 \to q_1) \bullet R(q_1 \to q_2) \bullet \cdots \bullet R(q_{k-1} \to q_k)$. Equivalently, w matches the sequence $q_0 \to q_1 \to \cdots \to q_k$ if either

- $w = \varepsilon$ and the sequence has only one state ($k = 0$), or
- $w = xy$ for some string x that matches the regular expression $R(q_0 \to q_1)$ and some string y that matches the remaining sequence $q_1 \to \cdots \to q_k$.

An expression automaton ***accepts*** any string that matches at least one sequence of states that starts at s and ends at t. The ***language*** of an expression automaton E is the set of all strings that E accepts.

Expression automata are nondeterministic. A single string could match several (even infinitely many) state sequences that start with s, and it could match each of those state sequences in several different ways. A string is accepted if *at least one* of the state sequences it matches ends at t. Conversely, a string might match *no* state sequences; all such strings are rejected.

Two special cases of expression automata are already familiar. First, every regular language is clearly the language of an expression automaton with exactly two states. Second, with only minor modifications, any DFA or NFA can be converted into an expression automaton with trivial transition expressions. Thompson's algorithm can be used to transform any expression automaton into an NFA, by recursively expanding any nontrivial transition. To complete the proof of Kleene's theorem, we show how to convert any expression automaton into a regular expression by repeatedly deleting vertices.

Lemma 4.3. *Every expression automaton accepts a regular language.*

Proof: Let $E = (Q, \Sigma, R, s, t)$ be an arbitrary expression automaton. Assume that any expression automaton with fewer states than E accepts a regular language. There are two cases to consider, depending on the number of states in Q:

- If $Q = \{s, t\}$, then trivially, E accepts the regular language $R(s \to t)$.
- On the other hand, suppose there is a state $q \in Q \setminus \{s, a\}$. We can modify the expression automaton so that state q is redundant and can be removed. Define a new transition function $R' \colon Q \times Q \to \mathit{Reg}(\Sigma)$ by setting

  $$R'(p \to r) := R(p \to r) + R(p \to q)\, R(q \to q)^*\, R(q \to r).$$

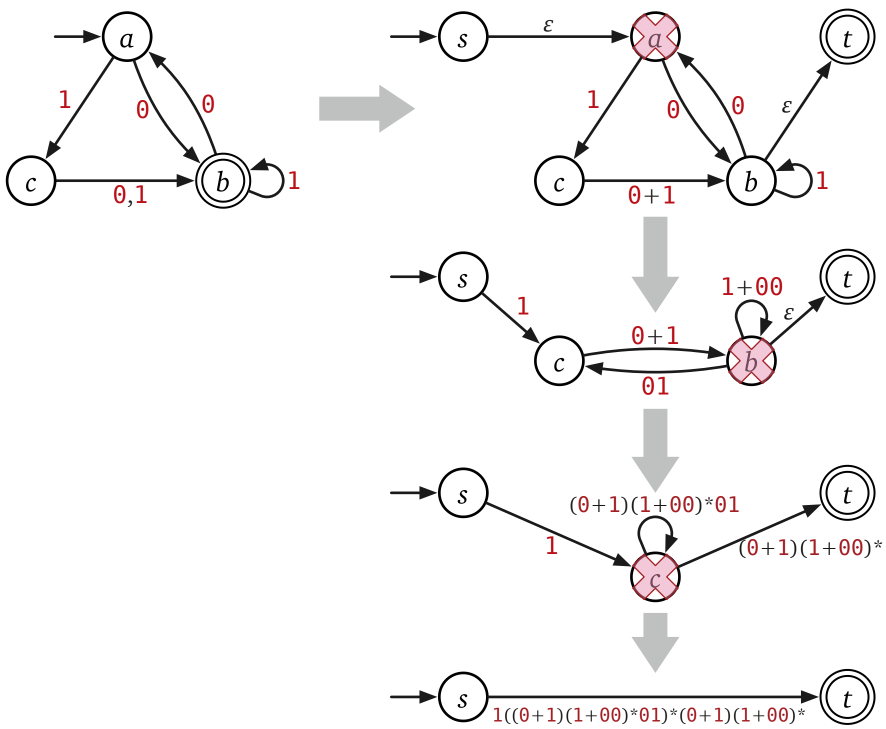
  With this modified transition function in place, any string w that matches the sequence $p \to q \to q \to \cdots \to q \to r$ with any number of q's also matches the single transition $p \to r$. Thus, by induction, if w matches a sequence of states, it also matches the subsequence obtained by removing all q's. Let E' be the expression automaton with states $Q' = Q \setminus \{q\}$ that uses this modified transition function R'. This new automaton accepts exactly the same strings as the original automaton E. Because E' has fewer states than E, the inductive hypothesis implies E' accepts a regular language.

In both cases, we conclude that E accepts a regular language. $\square$

This proof can be mechanically translated into an algorithm to convert any NFA—in particular, any DFA—into an equivalent regular expression. Given an NFA with n states (including s and

a), the algorithm iteratively removes $n-2$ states, updating $O(n^2)$ transition expressions in each iteration. If the concatenation and Kleene star operations could be performed in constant time, the resulting algorithm would run in $O(n^3)$ time. However, in each iteration, the transition expressions grows in length by roughly a factor of 4 in the worst case, so the final expression has length $\Theta(4^n)$. If we insist on representing the expressions as explicit strings, the worst-case running time is actually $\Theta(4^n)$.

A figure on the next page shows this conversion algorithm in action for a simple DFA. First we convert the DFA into an expression automaton by adding new start and accept states and merging two transitions, and then we remove each of the three original states, updating the transition expressions between any remaining states at each iteration. For the sake of clarity, edges $p\to q$ with $R(p\to q) = \varnothing$ are omitted from the figures.



Converting a DFA into an equivalent regular expression.

Exercises

1. For each of the following NFAs, describe an equivalent DFA. ("Describe" does not necessarily mean "draw"!)

★★★ Half a dozen examples.

2. For each of the following regular expressions, draw an equivalent NFA.

★★★ Half a dozen examples.

3. For each of the following regular expressions, describe an equivalent DFA. ("Describe" does not necessarily mean "draw"!)

★★★

> Half a dozen examples.

4. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular.

   (a) $\mathit{ones}(L) := \left\{ w \in 1^* \;\middle|\; |w| = |x| \text{ for some } x \in L \right\}$

   (b) $\mathit{reverse}(L) := \left\{ w \in \Sigma^* \;\middle|\; w^R \in L \right\}$. (Recall that w^R denotes the reversal of string w.)

   (c) $\mathit{prefix}(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^*\}$

   (d) $\mathit{suffix}(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^*\}$

   (e) $\mathit{substring}(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^*\}$

   (f) $\mathit{cycle}(L) := \{xy \mid x, y \in \Sigma^* \text{ and } yx \in L\}$

   (g) $\mathit{prefmax}(L) := \{x \in L \mid xy \in L \iff y = \varepsilon\}$.

   (h) $\mathit{sufmin}(L) := \{xy \in L \mid y \in L \iff x = \varepsilon\}$.

   (i) $\mathit{everyother}(L) := \{\mathit{everyother}(w) \mid w \in L\}$, where $\mathit{everyother}(w)$ is the subsequence of w containing every other symbol. For example, $\mathit{everyother}(\texttt{EVERYOTHER}) = \texttt{VROHR}$.

   (j) $\mathit{rehtoyreve}(L) := \{w \in \Sigma^* \mid \mathit{everyother}(w) \in L\}$.

   (k) $\mathit{subseq}(L) := \{x \in \Sigma^* \mid x \text{ is a subsequence of some } y \in L\}$

   (l) $\mathit{superseq}(L) := \{x \in \Sigma^* \mid \text{some } y \in L \text{ is a subsequence of } x\}$

   (m) $\mathit{left}(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^* \text{ where } |x| = |y|\}$

   (n) $\mathit{right}(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^* \text{ where } |x| = |y|\}$

   (o) $\mathit{middle}(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^* \text{ where } |x| = |y| = |z|\}$

   (p) $\mathit{half}(L) := \{w \in \Sigma^* \mid ww \in L\}$

   (q) $\mathit{third}(L) := \{w \in \Sigma^* \mid www \in L\}$

   (r) $\mathit{reflect}(L) := \left\{ w \in \Sigma^* \;\middle|\; ww^R \in L \right\}$

   *(s) $\mathit{sqrt}(L) := \left\{ x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = |x|^2 \right\}$

   *(t) $\mathit{log}(L) := \left\{ x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = 2^{|x|} \right\}$

   *(u) $\mathit{flog}(L) := \left\{ x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = F_{|x|} \right\}$, where F_n is the nth Fibonacci number.

*5. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular. *[Hint: For each language, there is an accepting NFA with at most q^q states, where q is the number of states in some DFA that accepts L.]*

(a) $repeat(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \geq 0\}$

(b) $allreps(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for every } n \geq 0\}$

(c) $manyreps(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for infinitely many } n \geq 0\}$

(d) $fewreps(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for finitely many } n \geq 0\}$

(e) $powers(L) := \left\{w \in \Sigma^* \;\middle|\; w^{2^n} \in L \text{ for some } n \geq 0\right\}$

★(f) $whatthe_N(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \in N\}$, where N is an ***arbitrary*** fixed set of non-negative integers. *[Hint: You only have to prove that an accepting NFA exists; you don't have to describe how to construct it.]*

6. For each of the following expression automata, describe an equivalent DFA *and* an equivalent regular expression.

★★★ Half a dozen examples.

Caveat lector: This is the first edition of this lecture note. Please send bug reports and suggestions to jeffe@illinois.edu.

*Imagine a piano keyboard, eh, 88 keys, only 88 and yet, and yet, hundreds of new melodies, new tunes, new harmonies are being composed upon hundreds of different keyboards every day in Dorset alone. Our language, tiger, our language: hundreds of thousands of available words, frillions of legitimate new ideas, so that I can say the following sentence and be utterly sure that nobody has ever said it before in the history of human communication: "**Hold the newsreader's nose squarely, waiter, or friendly milk will countermand my trousers.**" Perfectly ordinary words, but never before put in that precise order. A unique child delivered of a unique mother.*

— Stephen Fry, *A Bit of Fry and Laurie*, Series 1, Episode 3 (1989)

5 Context-Free Languages and Grammars

5.1 Definitions

Intuitively, a language is regular if it can be built from individual strings by concatenation, union, and repetition. In this note, we consider a wider class of ***context-free*** languages, which are languages that can be built from individual strings by concatenation, union, and *recursion*.

Formally, a language is context-free if and only if it has a certain type of recursive description known as a ***context-free grammar***, which is a structure with the following components:

- A finite set Σ, whose elements are called ***symbols*** or ***terminals***.
- A finite set Γ disjoint from Σ, whose elements are called ***non-terminals***.
- A finite set R of ***production rules*** of the form $A \to w$, where $A \in \Gamma$ is a non-terminal and $w \in (\Sigma \cup \Gamma)^*$ is a string of symbols and variables.
- A ***starting*** non-terminal, typically denoted S.

For example, the following eight production rules describe a context free grammar with terminals $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ and non-terminals $\Gamma = \{S, A, B\}$:

$$\begin{array}{llll} S \to A & A \to \mathtt{0}A & B \to B\mathtt{1} & C \to \varepsilon \\ S \to B & A \to \mathtt{0}C & B \to C\mathtt{1} & C \to \mathtt{0}C\mathtt{1} \end{array}$$

Normally we write grammars more compactly by combining the right sides of all rules for each non-terminal into one list, with alternatives separated by vertical bars.[1] For example, the previous grammar can be written more compactly as follows:

$$\begin{aligned} S &\to A \mid B \\ A &\to \mathtt{0}A \mid \mathtt{0}C \\ B &\to B\mathtt{1} \mid C\mathtt{1} \\ C &\to \varepsilon \mid \mathtt{0}C\mathtt{1} \end{aligned}$$

For the rest of this lecture, I will *almost* always use the following notational conventions.

[1]Yes, this means we now have *three* symbols $\cup$, $+$, and $\mid$ with exactly the same meaning. Sigh.

- Monospaced digits (`0`, `1`, `2`, ...), and symbols ($\diamond$, `$`, `#`, $\bullet$, ...) are explicit terminals.
- Early lower-case Latin letters $(a, b, c, \ldots)$ represent unknown or arbitrary terminals in Σ.
- Upper-case Latin letters $(A, B, C, \ldots)$ and the letter S represent non-terminals in Γ.
- Late lower-case Latin letters $(\ldots, w, x, y, z)$ represent strings in $(\Sigma \cup \Gamma)^*$, whose characters could be either terminals or non-terminals.

We can ***apply*** a production rule to a string in $(\Sigma \cup \Gamma)^*$ by replacing any instance of the non-terminal on the left of the rule with the string on the right. More formally, for any strings $x, y, z \in (\Sigma \cup \Gamma)^*$ and any non-terminal $A \in \Gamma$, applying the production rule $A \to y$ to the string xAz yields the string xyz. We use the notation $\boldsymbol{xAz \rightsquigarrow xyz}$ to describe this application. For example, we can apply the rule $C \to \texttt{0}C\texttt{1}$ to the string $\texttt{00}C\texttt{1}BAC\texttt{0}$ in two different ways:

$$\texttt{00}\underbrace{C}\texttt{1}BAC\texttt{0} \rightsquigarrow \texttt{00}\underbrace{\texttt{0}C\texttt{1}}\texttt{1}BAC\texttt{0} \qquad \texttt{00}C\texttt{1}BA\underbrace{C}\texttt{0} \rightsquigarrow \texttt{00}C\texttt{1}BA\underbrace{\texttt{0}C\texttt{1}}\texttt{0}$$

More generally, for any strings $x, z \in (\Sigma \cup \Gamma)^*$, we say that ***$z$ derives from x***, written $\boldsymbol{x \rightsquigarrow^* z}$, if we can transform x into z by applying a finite sequence of production rules, or more formally, if either

- $x = z$, or
- $x \rightsquigarrow y$ and $y \rightsquigarrow^* z$ for some string $y \in (\Sigma \cup \Gamma)^*$.

Straightforward definition-chasing implies that, for any strings $w, x, y, z \in (\sigma \cup \gamma)^*$, if $x \rightsquigarrow^* y$, then $wxz \rightsquigarrow^* wyz$.

The ***language $L(w)$*** of any string $w \in (\Sigma \cup \Gamma)^*$ is the set of all strings in Σ^* that derive from w:

$$L(w) := \{x \in \Sigma^* \mid w \rightsquigarrow^* x\}.$$

The language ***generated by*** a context-free grammar G, denoted $\boldsymbol{L(G)}$, is the language of its starting non-terminal. Finally, a language is ***context-free*** if it is generated by some context-free grammar.

Context-free grammars are sometimes used to model natural languages. In this context, the symbols are *words*, and the strings in the languages are *sentences*. For example, the following grammar describes a simple subset of English sentences. (Here I diverge from the usual notation conventions. Strings in ⟨angle brackets⟩ are non-terminals, and regular strings are terminals.)

⟨sentence⟩ → ⟨noun phrase⟩⟨verb phrase⟩⟨noun phrase⟩
⟨noun phrase⟩ → ⟨adjective phrase⟩⟨noun⟩
⟨adj. phrase⟩ → ⟨article⟩ | ⟨possessive⟩ | ⟨adjective phrase⟩⟨adjective⟩
⟨verb phrase⟩ → ⟨verb⟩ | ⟨adverb⟩⟨verb phrase⟩

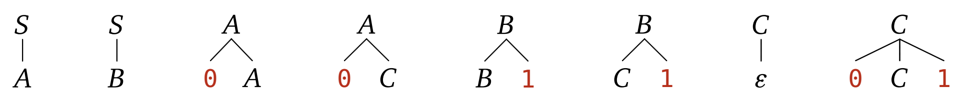
⟨noun⟩ → dog | trousers | daughter | nose | homework | time lord | pony | ···
⟨article⟩ → the | a | some | every | that | ···
⟨possessive⟩ → ⟨noun phrase⟩'s | my | your | his | her | ···
⟨adjective⟩ → friendly | furious | moist | green | severed | timey-wimey | little | ···
⟨verb⟩ → ate | found | wrote | killed | mangled | saved | invented | broke | ···
⟨adverb⟩ → squarely | incompetently | barely | sort of | awkwardly | totally | ···

5.2 Parse Trees

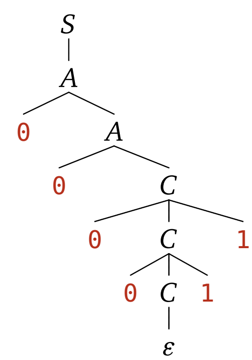
It is often useful to visualize derivations of strings in $L(G)$ using a ***parse tree***. The parse tree for a string $w \in L(G)$ is a rooted ordered tree where

- Each leaf is labeled with a terminal or the empty string ε. Concatenating these in order from left to right yields the string w.
- Each internal node is labeled with a non-terminal. In particular, the root is labeled with the start non-terminal S.
- For each internal node v, there is a production rule $A \to \omega$ where A is the label of v and the symbols in ω are the labels of the children of v in order from left to right.

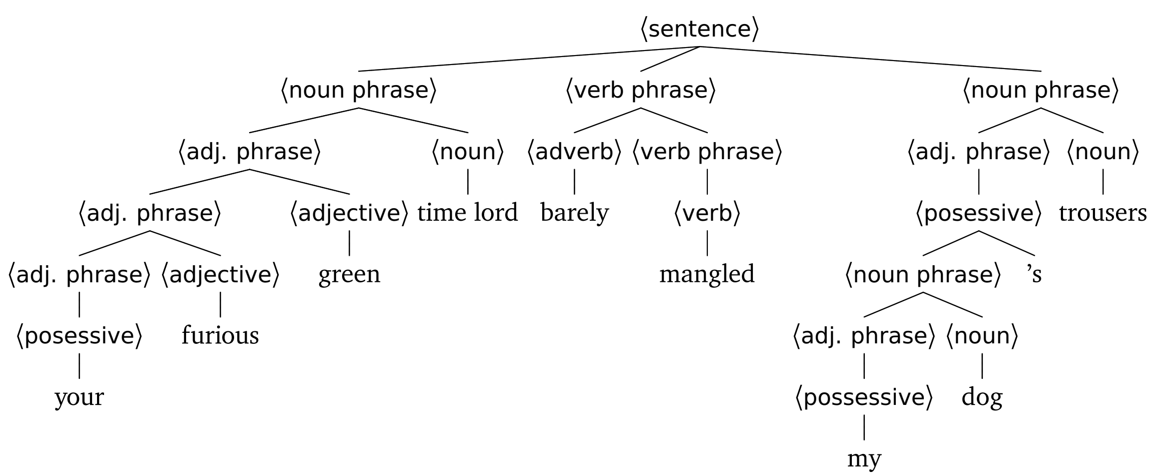
In other words, the production rules of the grammar describe *template trees* that can be assembled into larger parse trees. For example, the simple grammar on the previous page has the following templates, one for each production rule:



The same grammar gives us the following parse tree for the string `000011`:

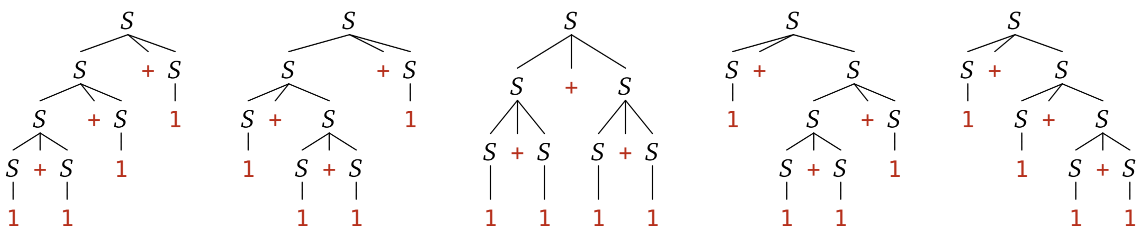


Our more complicated "English" grammar gives us parse trees like the following:



Any parse tree that contains at least one node with more than one non-terminal child corresponds to several different derivations. For example, when deriving an "English" sentence, we have a choice of whether to expand the first ⟨noun phrase⟩ ("your furious green time lord") before or after the second ("my dog's trousers").

A string w is ***ambiguous*** with respect to a grammar if there is more than one parse tree for w, and a grammar G is ***ambiguous*** is some string is ambiguous with respect to G. Neither of the previous example grammars is ambiguous. However, the grammar $S \to \mathtt{1} \mid S\mathtt{+}S$ is ambiguous, because the string `1+1+1+1` has five different parse trees:



A context-free language L is ***inherently ambiguous*** if every context-free grammar that generates L is ambiguous. The language generated by the previous grammar (the regular language $(\mathtt{1+})^*\mathtt{1}$) is *not* inherently ambiguous, because the unambiguous grammar $S \to \mathtt{1} \mid \mathtt{1+}S$ generates the same language.

5.3 From Grammar to Language

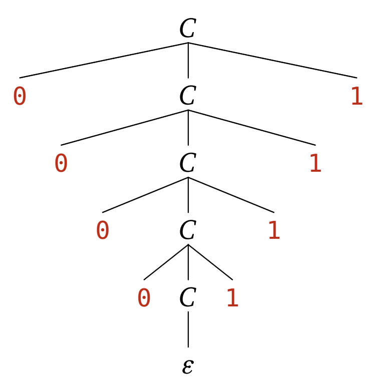
Let's figure out the language generated by our first example grammar

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}.$$

Since the production rules for non-terminal C do not refer to any other non-terminal, let's begin by figuring out $L(C)$. After playing around with the smaller grammar $C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$ for a few seconds, you can probably guess that its language is $\{\varepsilon, \mathtt{01}, \mathtt{0011}, \mathtt{000111}, \ldots\}$, that is, the set all of strings of the form $\mathtt{0}^n\mathtt{1}^n$ for some integer n. For example, we can derive the string `00001111` from the start non-terminal S using the following derivation:

$$C \rightsquigarrow \mathtt{0}C\mathtt{1} \rightsquigarrow \mathtt{00}C\mathtt{11} \rightsquigarrow \mathtt{000}C\mathtt{111} \rightsquigarrow \mathtt{0000}C\mathtt{1111} \rightsquigarrow \mathtt{0000}\varepsilon\mathtt{1111} = \mathtt{00001111}$$

The same derivation can be viewed as the following parse tree:



In fact, it is not hard to *prove* by induction that $L(C) = \{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ as follows. As usual when we prove that two sets X and Y are equal, the proof has two stages: one stage to prove $X \subseteq Y$, the other to prove $Y \subseteq X$.

- First we prove that $C \rightsquigarrow^* 0^n 1^n$ for every non-negative integer n.

  Fix an arbitrary non-negative integer n. Assume that $C \rightsquigarrow^* 0^k 1^k$ for every non-negative integer $k < n$. There are two cases to consider.

  – If $n = 0$, then $0^n 1^n = \varepsilon$. The rule $C \to \varepsilon$ implies that $C \rightsquigarrow \varepsilon$ and therefore $C \rightsquigarrow^* \varepsilon$.
  – Suppose $n > 0$. The inductive hypothesis implies that $C \rightsquigarrow^* 0^{n-1} 1^{n-1}$. Thus, the rule $C \to 0C1$ implies that $C \rightsquigarrow 0C1 \rightsquigarrow^* 0(0^{n-1}1^{n-1})1 = 0^n 1^n$.

  In both cases, we conclude that that $C \rightsquigarrow^* 0^n 1^n$, as claimed.

- Next we prove that for every string $w \in \Sigma^*$ such that $C \rightsquigarrow^* w$, we have $w = 0^n 1^n$ for some non-negative integer n.

  Fix an arbitrary string w such that $C \rightsquigarrow^* w$. Assume that for any string x such that $|x| < |w|$ and $C \rightsquigarrow^* x$, we have $x = 0^k 1^k$ for some non-negative integer k. There are two cases to consider, one for each production rule.

  – If $w = \varepsilon$, then $w = 0^0 1^0$.
  – Suppose $w = 0x1$ for some string x such that $C \rightsquigarrow^* x$. Because $|x| = |w| - 2 < |w|$, the inductive hypothesis implies that $x = 0^k 1^k$ for some integer k. Then we have $w = 0^{k+1} 1^{k+1}$.

  In both cases, we conclude that that $w = 0^n 1^n$ for some non-negative integer n, as claimed.

The first proof uses induction on strings, following the boilerplate proposed in the previous lecture; in particular, the case analysis mirrors the recursive definition of "string". The second proof uses *structural induction* on the grammar; the case analysis mirrors the recursive definition of the language of S, as described by the production rules. In both proofs, the inductive hypothesis is "Assume there is no smaller counterexample."

Similar analysis implies that $L(A) = \{0^m 1^n \mid m > n\}$ and $L(B) = \{0^m 1^n \mid m < n\}$, and therefore $L(S) = \{0^m 1^n \mid m \neq n\}$.

5.4 More Examples

★★★

Give three or four examples of simple but interesting context-free grammars. Some possibilities:

- Same number of 0s and 1s
- Different number of 0s and 1s
- Palindromes
- Balanced parentheses
- Arithmetic/algebraic expressions
- Regular expressions

5.5 Regular Languages are Context-Free

The following inductive argument proves that every regular language is also a context-free language. Let L be an arbitrary regular language, encoded by some regular expression R. Assume that any regular expression shorter than R represents a context-free language. ("Assume no smaller counterexample.") We construct a context-free grammar for L as follows. There are several cases to consider.

- Suppose L is empty. Then L is generated by the trivial grammar $S \to S$.
- Suppose $L = \{w\}$ for some string $w \in \Sigma^*$. Then L is generated by the grammar $S \to w$.
- Suppose L is the union of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1 \cup L_2$ is generated by the production rule $S \to S_1 \mid S_2$.
- Suppose L is the concatenation of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1 L_2$ is generated by the production rule $S \to S_1 S_2$.
- Suppose L is the Kleene closure of some regular language L_1. The inductive hypothesis implies that L_1 is context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1. Then $L = L_1^*$ is generated by the production rule $S \to \varepsilon \mid S_1 S$.

In every case, we have found a context-free grammar that generates L, which means L is context-free.

In the next lecture note, we will prove that the context-free language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$ is not regular. (In fact, this is the *canonical example* of a non-regular language.) Thus, context-free grammars are strictly more expressive than regular expressions.

5.6 Not Every Language is Context-Free

Again, you may be tempted to conjecture that *every* language is context-free, but a variant of our earlier cardinality argument implies that this is not the case.

Any context-free grammar over the alphabet Σ can be encoded as a string over the alphabet $\Sigma \cup \Gamma \cup \{\mathtt{Ɛ}, \rightarrow, \mathtt{|}, \mathtt{\$}\}$, where $\mathtt{\$}$ indicates the end of the production rules for each non-terminal. For example, our example grammar

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

can be encoded as the string

$$\mathtt{S{\rightarrow}A|B\$A{\rightarrow}0A|0C\$B{\rightarrow}B1|C1\$C{\rightarrow}Ɛ|0C1\$}$$

We can further encode any such string as a *binary* string by associating each symbol in the set $\Sigma \cup \Gamma \cup \{\mathtt{Ɛ}, \rightarrow, \mathtt{|}, \mathtt{\$}\}$ with a different binary substring. Specifically, if we encode each of the grammar symbols $\mathtt{Ɛ}, \rightarrow, \mathtt{|}, \mathtt{\$}$ as a string of the form $\mathtt{11}^*\mathtt{0}$, each terminal in Σ as a string of the form $\mathtt{011}^*\mathtt{0}$, and each non-terminal as a string of the form $\mathtt{0011}^*\mathtt{0}$, we can unambiguously recover the grammar from the encoding. For example, applying the code

$$\begin{array}{lll} \mathtt{Ɛ} \mapsto \mathtt{10} & \mathtt{0} \mapsto \mathtt{010} & \mathtt{S} \mapsto \mathtt{0010} \\ \rightarrow \mapsto \mathtt{110} & \mathtt{1} \mapsto \mathtt{0110} & \mathtt{A} \mapsto \mathtt{00110} \\ \mathtt{|} \mapsto \mathtt{1110} & & \mathtt{B} \mapsto \mathtt{001110} \\ \mathtt{\$} \mapsto \mathtt{11110} & & \mathtt{C} \mapsto \mathtt{0011110} \end{array}$$

transforms our example grammar into the 135-bit string

```
00101100011011100011101111000110
11001000110111001000111101111000
11101100011100110111000111100110
11110001111011010111001000111100
1011110.
```

Adding a 1 to the start of this bit string gives us the binary encoding of the integer

$$51\,115\,617\,766\,581\,763\,757\,672\,062\,401\,233\,529\,937\,502.$$

Our construction guarantees that two different context-free grammars over the same language (ignoring changing the names of the non-terminals) yield different positive integers. Thus, the set of context-free grammars over any alphabet is *at most* as large as the set of integers, and is therefore countably infinite. (Most integers are not encodings of context-free grammars, but that only helps us.) It follows that the set of all context-free *languages* over any fixed alphabet is also countably infinite. But we already showed that the set of *all* languages over any alphabet is uncountably infinite. So almost all languages are non-context-free!

Although we will probably not see them in this course, there are techniques for proving that certain languages are not context-free, just as there are for proving certain languages are not regular. In particular, the $\{\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$ is not context-free. (In fact, this is the *canonical example* of a non-context-free language.)

★5.7 Recursive Automata

All the flavors of finite-state automata we have seen so far describe/encode/accept/compute *regular* languages; these are precisely the languages that can be constructed from individual strings by union, concatenation, and unbounded repetition. Just as context-free grammars are recursive generalizations of regular expressions, we can define a class of machines called *recursive automata*, which generalize (nondeterministic) finite-state automata.

Formally, a ***recursive automaton*** consists of the following components:

- A non-empty finite set Σ, called the ***input alphabet***
- Another non-empty finite set N, disjoint from Σ, whose elements are called ***module names***
- A ***start name*** $S \in N$
- A set $M = \{M_A \mid A \in N\}$ of NFAs over the alphabet $\Sigma \cup N$ called ***modules***, each with a single accepting state. Each module M_A has the following components:
  - A finite set Q_A of ***states***, such that $Q_A \cap Q_B = \varnothing$ for all $A \neq B$
  - A ***start*** state $s_A \in Q_A$
  - A ***terminal*** or ***accepting*** state $t_A \in Q_A$
  - A ***transition function*** $\delta_A \colon Q_A \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^{Q_A}$.

Equivalently, we have a single global transition function $\delta \colon Q \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^Q$, where $Q = \bigcup_{A \in N} Q_A$, such that for any name A and any state $q \in Q_A$ we have $\delta(q) \subseteq Q_A$. Machine M_S is called the ***main module***.

A ***configuration*** of a recursive automaton is a triple (w, q, s), where w is a string in Σ^* called the ***input***, q is a state in Q called the ***local state***, and σ is a string in Q^* called the ***stack***. The module containing the local state q is called the ***active module***. A configuration can be changed by three types of transitions.

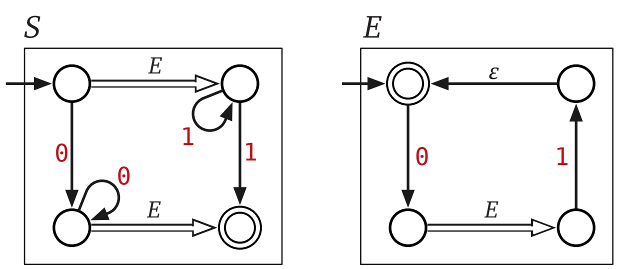
- A ***read*** transition consumes the first symbol in the input and changes the local state within the current module, just like a standard NFA.
- An ***epsilon*** transition changes the local state within the current module, without consuming any input characters, just like a standard NFA.
- A ***call*** transition chooses an arbitrary name A, changes the current state q_0 to some state in $\delta(q, A)$, and pushes the corresponding start state s_A onto the stack (thereby changing the active module to M_A), without consuming any input characters.
- Finally, if the current state is the terminal state of some module *and* the stack is non-empty, a ***return*** transition pops the top state off the stack and makes it the new local state (thereby possibly changing the active module), without consuming any input characters.

Symbolically, we can describe these transitions as follows:

$$\begin{array}{rrl}
\textbf{read:} & (ax, q, \sigma) \longmapsto (x, q', \sigma) & \text{for some } q' \in \delta(q, a) \\
\textbf{epsilon:} & (w, q, \sigma) \longmapsto (w, q', \sigma) & \text{for some } q' \in \delta(q, \varepsilon) \\
\textbf{call:} & (w, q, \sigma) \longmapsto (w, s_A, q' \cdot \sigma) & \text{for some } A \in N \text{ and } q' \in \delta(q, A) \\
\textbf{return:} & (w, t_A, q \cdot \sigma) \longmapsto (w, q, \sigma) &
\end{array}$$

A recursive automaton ***accepts*** a string w if there is a *finite* sequence of transitions starting at the start configuration (w, s_S, ε) and ending at the terminal configuration $(\varepsilon, t_S, \varepsilon)$.

For example, the following recursive automaton accepts the language $\{\mathtt{0}^m\mathtt{1}^n \mid m \neq n\}$. The recursive automaton has two component machines; the start machine named S and a "subroutine" named E (for "equal") that accepts the language $\{\mathtt{0}^n\mathtt{1}^n \mid n \geq 0\}$. White arrows indicate recursive transitions.



A recursive automaton for the language $\{\mathtt{0}^m\mathtt{1}^n \mid m \neq n\}$

Lemma 5.1. *Every context-free language is accepted by a recursive automaton.*

Proof:

★★★ Direct construction from the CFG, with one module per nonterminal.

□

For example, the context-free grammar

$$\begin{aligned}
S &\to 0A \mid B1 \\
A &\to 0A \mid E \\
B &\to B1 \mid E \\
E &\to \varepsilon \mid 0E0
\end{aligned}$$

leads to the following recursive automaton with four modules:

★★★ Figure!

Lemma 5.2. *Every recursive automaton accepts a context-free language.*

Proof (sketch): Let $R = (\Sigma, N, S, \delta, M)$ be an arbitrary recursive automaton. We define a context-free grammar G that describes the language accepted by R as follows.

The set of nonterminals in G is isomorphic the state set Q; that is, for each state $q \in Q$, the grammar contains a corresponding nonterminal $[q]$. The language of $[q]$ will be the set of strings w such that there is a finite sequence of transitions starting at the start configuration (w, q, ε) and ending at the terminal configuration $(\varepsilon, t, \varepsilon)$, where t is the terminal state of the module containing q.

The grammar has four types of production rules, corresponding to the four types of transitions:

- ***read:*** For each symbol a and each pair of states p and q such that $p \in \delta(q, a)$, the grammar contains the production rule $[q] \to a[p]$.
- ***epsilon:*** For any two states p and q such that $p \in \delta(q, \varepsilon)$, the grammar contains the production rule $[q] \to [p]$.
- ***call:*** Each name A and each pair of states states p and q such that $p \in \delta(q, A)$, the grammar contains the production rule $[q] \to [s_A][p]$.
- ***return:*** Each name A, the grammar contains the production rule $[t_A] \to \varepsilon$.

Finally, the starting nonterminal of G is $[s_S]$, which corresponds to the start state of the main module.

We can now argue inductively that the grammar G and the recursive automaton R describe the same language. Specifically, any sequence of transitions in R from (w, s_S, ε) to $(\varepsilon, t_S, \varepsilon)$ can be transformed mechanically into a derivation of w from the nonterminal $[s_S]$ in G. Symmetrically, the ***leftmost*** derivation of any string w in G can be mechanically transformed into an accepting sequence of transitions in R. We omit the straightforward but tedious details. □

For example, the recursive automaton on the previous page gives us the following context-free grammar. To make the grammar more readable, I've renamed the nonterminals corresponding to start and terminal states: $S = [s_S]$, $T = [t_S]$, and $E = [s_E] = [t_E]$:

$$\begin{array}{ll}
S \to EA \mid \mathtt{0}B & \qquad E \to \varepsilon \mid \mathtt{0}X \\
A \to \mathtt{1}A \mid \mathtt{1}T & \qquad X \to EY \\
B \to \mathtt{0}B \mid ET & \qquad Y \to \mathtt{1}Z \\
T \to \varepsilon & \qquad Z \to E
\end{array}$$

Our earlier proofs imply that we can forbid ε-transitions or even allow regular-expression transitions in our recursive automata without changing the set of languages they accept.

*5.8 Chomsky Normal Form

For many algorithmic problems involving context-free grammars, it is helpful to consider grammars with a particular special structure called ***Chomsky normal form***, abbreviated ***CNF***:

- The starting non-terminal S does not appear on the right side of any production rule.

- The starting non-terminal S *may* have the production rule $S \to \varepsilon$.
- The right side of every other production rule is either a single terminal symbol or a string of exactly two non-terminals—that is, every other production rule has the form $A \to BC$ or $A \to a$.

A particularly attractive feature of CNF grammars is that they yield *full binary* parse trees; in particular, every parse tree for a string of length $n > 0$ has exactly $2n-1$ non-terminal nodes. Consequently, any string of length n in the language of a CNF grammar can be derived in exactly $2n-1$ production steps. It follows that we can actually determine whether a string belongs to the language of a CNF grammar by brute-force consideration of all possible derivations of the appropriate length.

For arbitrary context-free grammars, there is no similar upper bound on the length of a derivation, and therefore no similar brute-force membership algorithm, because the grammar may contain additional ***ε-productions*** of the form $A \to \varepsilon$ and/or ***unit productions*** of the form $A \to B$, where both A and B are non-terminals. Unit productions introduce nodes of degree 1 into any parse tree, and ε-productions introduce leaves that do not contribute to the word being parsed.

Fortunately, it *is* possible to determine membership in the language of an arbitrary context-free grammar, thanks to the following theorem. Two context-free grammars are ***equivalent*** if they define the same language.

Every context-free grammar is equivalent to a grammar in Chomsky normal form.

To be more specific, define the ***total length*** of a context-free grammar to be the number of symbols needed to write down the grammar; up to constant factors, the total length is the sum of the lengths of the production rules.

Theorem 5.3. *For every context-free grammar with total length L, there is an equivalent grammar in Chomsky normal form with total length $O(L^2)$, which can be computed in $O(L^2)$ time.*

Converting an arbitrary grammar into Chomsky normal form is a complex task. Fortunately, for most applications of context-free grammars, it's enough to know that the algorithm exists. For the sake of completeness, however, I will describe one such conversion algorithm here. This algorithm consists of several relatively straightforward stages. Efficient implementation of some of these stages requires standard graph-traversal algorithms, which we will describe much later in the course.

0. Add a new starting non-terminal. Add a new non-terminal S' and a production rule $S' \to S$, where S is the starting non-terminal for the given grammar. S' will be the starting non-terminal for the resulting CNF grammar. (In fact, this step is necessary only when $S \rightsquigarrow^* \varepsilon$, but at this point in the conversion process, we don't yet know whether that's true.)

1. Decompose long production rules. For each production rule $A \to \omega$ whose right side w has length greater than two, add new production rules of length two that still permit the derivation $A \rightsquigarrow^* \omega$. Specifically, suppose $\omega = \alpha\chi$ for some symbol $\alpha \in \Sigma \cup \Gamma$ and string $\chi \in (\Sigma \cup \Gamma)^*$. The algorithm replaces $A \to \omega$ with two new production rules $A \to \alpha B$ and $B \to \chi$, where B is a new non-terminal, and then (if necessary) recursively decomposes the production rule $B \to \chi$. For example, we would replace the long production rule $A \to \mathtt{0}BC\mathtt{1}CB$ with the following sequence of short production rules, where each A_i is a new non-terminal:

$$A \to \mathtt{0}A_1 \qquad A_1 \to BA_2 \qquad A_2 \to CA_3 \qquad A_3 \to \mathtt{1}A_4 \qquad A_4 \to CB$$

This stage can significantly increase the number of non-terminals and production rules, but it increases the *total length* of all production rules by at most a small constant factor.[2] Moreover, for the remainder of the conversion algorithm, every production rule has length at most two. The running time of this stage is $O(L)$.

2. Identify nullable non-terminals. A non-terminal A is ***nullable*** if and only if $A \rightsquigarrow^* \varepsilon$. The recursive definition of $\rightsquigarrow^*$ implies that A is nullable if and only if the grammar contains a production rule $A \to \omega$ where ω consists entirely of nullable non-terminals (in particular, if $\omega = \varepsilon$). You may be tempted to transform this recursive characterization directly into a recursive algorithm, but this is a bad idea; the resulting algorithm would fall into an infinite loop if (for example) the same non-terminal appeared on both sides of the same production rule. Instead, we apply the following ***fixed-point*** algorithm, which repeatedly scans through the entire grammar until a complete scan discovers no new nullable non-terminals.

```
NULLABLES(Σ, Γ, R, S):
  Γ_ε ← ∅        ⟨⟨known nullable non-terminals⟩⟩
  done ← FALSE
  while ¬done
      done ← TRUE
      for each non-terminal A ∈ Γ \ Γ_ε
          for each production rule A → ω
              if ω ∈ Γ_ε*
                  add A to Γ_ε
                  done ← FALSE
  return Γ_ε
```

At this point in the conversion algorithm, if S' is ***not*** identified as nullable, then we can safely remove it from the grammar and use the original starting nonterminal S instead.

As written, NULLABLES runs in $O(nL) = O(L^2)$ time, where n is the number of non-terminals in Γ. Each iteration of the main loop except the last adds at least one non-terminal to Γ_ε, so the algorithm halts after at most $n + 1 \leq L$ iterations, and in each iteration, we examine at most L production rules. There is a faster implementation of NULLABLES that runs in $O(n + L) = O(L)$ time,[3] but since other parts of the conversion algorithm already require $O(L^2)$ time, we needn't bother.

3. Eliminate *ε*-productions. First, remove every production rule of the form $A \to \varepsilon$. Then for each production rule $A \to w$, add all possible new production rules of the form $A \to w'$, where w'

---

[2] In most textbook descriptions of this conversion algorithm, this stage is performed *last*, after removing ε-productions and unit productions. But with the stages in that traditional order, removing ε-productions could *exponentially* increase the length of the grammar in the worst case! Consider the production rule $A \to (BC)^k$, where B is nullable but C is not. Decomposing this rule first and then removing ε-productions introduces about $3k$ new rules; whereas, removing ε-productions first introduces 2^k new rules, most of which then must then be further decomposed.

[3] Consider the bipartite graph whose vertices correspond to non-terminals and the right sides of production rules, with one edge per rule. The faster algorithm is a modified breadth-first search of this graph, starting at the vertex representing ε.

is a ***non-empty*** string obtained from w by removing one nullable non-terminal. For example, if if the grammar contained the production rule $A \to BC$, where B and C are both nullable, we would add two new production rules $A \to B \mid C$. Finally, if the starting nonterminal S' was identified as nullable in the previous stage, add the production rule $S' \to \varepsilon$; this will be the *only* ε-production in the final grammar. This phase of the conversion runs in $O(L)$ time and at most triples the number of production rules.

4. Merge equivalent non-terminals. We say that two non-terminals A and B are ***equivalent*** if they can be derived from each other: $A \rightsquigarrow^* B$ and $B \rightsquigarrow^* A$. Because we have already removed ε-productions, any such derivation must consist entirely of unit productions. For example, in the grammar

$$S \to B \mid C, \qquad A \to B \mid D \mid CC \mid \mathtt{0}, \qquad B \to C \mid AD \mid \mathtt{1}, \qquad C \to A \mid DA, \qquad D \to BA \mid CS,$$

non-terminals A, B, C are all equivalent, but S is not in that equivalence class (because we cannot derive S from A) and neither is D (because we cannot derive A from D).

Construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Then two non-terminals are equivalent if and only if they are in the same strong component of G. Compute the strong components of G in $O(L)$ time using, for example, the algorithm of Kosaraju and Sharir. Then merge all the non-terminals in each equivalence class into a single non-terminal. Finally, remove any unit productions of the form $A \to A$. The total running time for this phase is $O(L)$. Starting with our example grammar above, merging B and C with A and removing the production $A \to A$ gives us the simpler grammar

$$S \to A, \qquad A \to AA \mid D \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

We could further simplify the grammar by merging all non-terminals reachable from S using only unit productions (in this case, merging non-terminals S and S), but this further simplification is unnecessary.

5. Remove unit productions. Once again, we construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Because no two non-terminals are equivalent, G is acyclic. Thus, using topological sort, we can index the non-terminals $A_1, A_2, \ldots, A_n$ such that for every unit production $A_i \to A_j$ we have $i < j$, again in $O(L)$ time; moreover, we can assume that the starting non-terminal is A_1. (In fact, both the dag G and the linear ordering of non-terminals was already computed in the previous phase!!)

Then for each index j in decreasing order, for each unit production $A_i \to A_j$ and each production $A_j \to \omega$, we add a new production rule $A_i \to \omega$. At this point, all unit productions are redundant and can be removed. Applying this algorithm to our example grammar above gives us the grammar

$$S \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad A \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

In the worst case, each production rule for A_n is copied to each of the other $n-1$ non-terminals. Thus, this phase runs in $\Theta(nL) = O(L^2)$ time and increases the length of the grammar to $\Theta(nL) = O(L^2)$ in the worst case.

This phase dominates the running time of the CNF conversion algorithm. Unlike previous phases, no faster algorithm for removing unit transformations is known! There are grammars of length L with unit productions such that any equivalent grammar without unit productions has length $\Omega(L^{1.499999})$ (for any desired number of 9s), but this lower bound does not rule out the possibility of an algorithm that runs in only $O(L^{3/2})$ time. Closing the gap between $\Omega(L^{3/2-\varepsilon})$ and $O(L^2)$ has been an open problem since the early 1980s.

6. Protect terminals. Finally, for each terminal $a \in \Sigma$, we introduce a new non-terminal A_a and a new production rule $A_a \to a$, and then replace a with A_a in every production rule of length 2. This completes the conversion to Chomsky normal form. As claimed, the total running time of the algorithm is $O(L^2)$, and the total length of the output grammar is also $O(L^2)$.

CNF Conversion Example

As a running example, let's apply these stages one at a time to our first example grammar.

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

0. Add a new starting non-terminal S'.

$$\underline{S' \to S} \qquad S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

1. Decompose the long production rule $C \to \mathtt{0}C\mathtt{1}$.

$$S' \to S \qquad S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad \underline{C \to \varepsilon \mid \mathtt{0}D \qquad D \to C\mathtt{1}}$$

2. Identify C as the only nullable non-terminal. Because S' is not nullable, remove the production rule $S' \to S$.

3. Eliminate the ε-production $C \to \varepsilon$.

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \underline{\mid \mathtt{0}} \qquad B \to B\mathtt{1} \mid C\mathtt{1} \underline{\mid \mathtt{1}} \qquad C \to \mathtt{0}D \qquad D \to C\mathtt{1} \underline{\mid \mathtt{1}}$$

4. No two non-terminals are equivalent, so there's nothing to merge.

5. Remove the unit productions $S' \to S$, $S \to A$, and $S \to B$.

$$S \to \underline{\mathtt{0}A \mid \mathtt{0}C \mid B\mathtt{1} \mid C\mathtt{1} \mid \mathtt{0} \mid \mathtt{1}}$$
$$A \to \mathtt{0}A \mid \mathtt{0}C \mid \mathtt{0} \qquad B \to B\mathtt{1} \mid C\mathtt{1} \mid \mathtt{1} \qquad C \to \mathtt{0}D \qquad D \to C\mathtt{1} \mid \mathtt{1}.$$

6. Finally, protect the terminals $\mathtt{0}$ and $\mathtt{1}$ to obtain the final CNF grammar.

$$\begin{array}{ll} S \to \underline{EA \mid EC \mid BF \mid CF \mid \mathtt{0} \mid \mathtt{1}} & \\ A \to \underline{EA \mid EC} \mid \mathtt{0} & B \to \underline{BF \mid CF} \mid \mathtt{1} \\ C \to \underline{ED} & D \to \underline{CF} \mid \mathtt{1} \\ \underline{E \to \mathtt{0}} & \underline{F \to \mathtt{1}} \end{array}$$

Exercises

1. Describe context-free grammars that generate each of the following languages. The function $\#(x, w)$ returns the number of occurrences of the ***substring*** x in the string w. For example, $\#(\mathtt{0}, \mathtt{101001}) = 3$ and $\#(\mathtt{010}, \mathtt{1010100011}) = 2$.

   (a) All strings in $\{\mathtt{0}, \mathtt{1}\}^*$ whose length is divisible by 5.

   (b) All strings in $\{\mathtt{0}, \mathtt{1}\}^*$ representing a non-negative multiple of 5 in binary.

   (c) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{0}, w) = \#(\mathtt{1}, w)\}$

   (d) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{0}, w) \neq \#(\mathtt{1}, w)\}$

   (e) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{00}, w) = \#(\mathtt{11}, w)\}$

   (f) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{01}, w) = \#(\mathtt{10}, w)\}$

   (g) $\{w \in \{\mathtt{0}, \mathtt{1}\}^* \mid \#(\mathtt{0}, w) = \#(\mathtt{1}, w) \text{ and } |w| \text{ is a multiple of } 3\}$

   (h) $\{\mathtt{0}, \mathtt{1}\}^* \setminus \{\mathtt{0}^n \mathtt{1}^n \mid n \geq 0\}$

   (i) $\{\mathtt{0}^n \mathtt{1}^{2n} \mid n \geq 0\}$

   (j) $\{\mathtt{0}, \mathtt{1}\}^* \setminus \{\mathtt{0}^n \mathtt{1}^{2n} \mid n \geq 0\}$

   (k) $\{\mathtt{0}^n \mathtt{1}^m \mid 0 \leq 2m \leq n < 3m\}$

   (l) $\{\mathtt{0}^i \mathtt{1}^j \mathtt{2}^{i+j} \mid i, j \geq 0\}$

   (m) $\{\mathtt{0}^i \mathtt{1}^j \mathtt{2}^k \mid i = j \text{ or } j = k\}$

   (n) $\{\mathtt{0}^i \mathtt{1}^j \mathtt{2}^k \mid i \neq j \text{ or } j \neq k\}$

   (o) $\{\mathtt{0}^i \mathtt{1}^j \mathtt{0}^j \mathtt{1}^i \mid i, j \geq 0\}$

   (p) $\left\{w\$\mathtt{0}^{\#(\mathtt{0}, w)} \,\middle|\, w \in \{\mathtt{0}, \mathtt{1}\}^*\right\}$

   (q) $\{xy \mid x, y \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } x \neq y \text{ and } |x| = |y|\}$

   (r) $\left\{x\$y^R \,\middle|\, x, y \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } x \neq y\right\}$

   (s) $\{x\$y \mid x, y \in \{\mathtt{0}, \mathtt{1}\}^* \text{ and } \#(\mathtt{0}, x) = \#(\mathtt{1}, y)\}$

   (t) $\{\mathtt{0}, \mathtt{1}\}^* \setminus \{ww \mid w \in \{\mathtt{0}, \mathtt{1}\}^*\}$

   (u) All strings in $\{\mathtt{0}, \mathtt{1}\}^*$ that are *not* palindromes.

   (v) All strings in $\{\mathtt{(}, \mathtt{)}, \diamond\}^*$ in which the parentheses are balanced and the symbol $\diamond$ appears at most four times. For example, `()(())` and `(◇◇(()()◇)()())◇` and `◇◇◇` are strings in this language, but `)(()` and `(◇◇◇)◇◇` are not.

2. Describe recursive automata for each of the languages in problem 1. ("Describe" does not necessarily mean "draw"!)

3. Prove that if L is a context-free language, then L^R is also a context-free language. *[Hint: How do you reverse a context-free grammar?]*

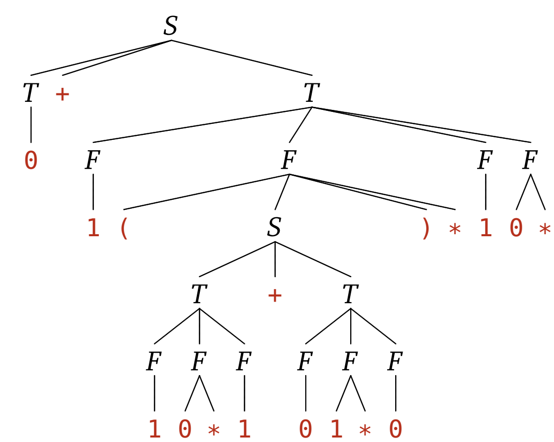
4. Consider a generalization of context-free grammars that allows any *regular expression* over $\Sigma \cup \Gamma$ to appear on the right side of a production rule. Without loss of generality, for each non-terminal $A \in \Gamma$, the generalized grammar contains a single regular expression $R(A)$. To apply a production rule to a string, we replace any non-terminal A with an arbitrary word

in the language described by $R(A)$. As usual, the language of the generalized grammar is the set of all strings that can be derived from its start non-terminal.

For example:, the following generalized context-free grammar describes the language of all regular expressions over the alphabet $\{0, 1\}$:

$$S \to (T+)^*T + \emptyset \qquad \text{(Regular expressions)}$$
$$T \to \varepsilon + F^*F \qquad \text{(Terms = summable expressions)}$$
$$F \to (0 + 1 + (S))(* + \varepsilon) \qquad \text{(Factors = concatenable expressions)}$$

Here is a parse tree for the regular expression `0+1(10*1+01*0)*10*` (which represents the set of all binary numbers divisible by 3):



Prove that every *generalized* context-free grammar describes a context-free language. In other words, show that allowing regular expressions to appear in production rules does not increase the expressive power of context-free grammars.

Think globally, act locally.

— Attributed to Patrick Geddes (c.1915), among many others.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.

— Alan Turing, "Computing Machinery and Intelligence" (1950)

Never worry about theory as long as the machinery does what it's supposed to do.

— Robert Anson Heinlein, *Waldo & Magic, Inc.* (1950)

6 Turing Machines

In 1936, a few months before his 24th birthday, Alan Turing launched computer science as a modern intellectual discipline. In a single remarkable paper, Turing provided the following results:

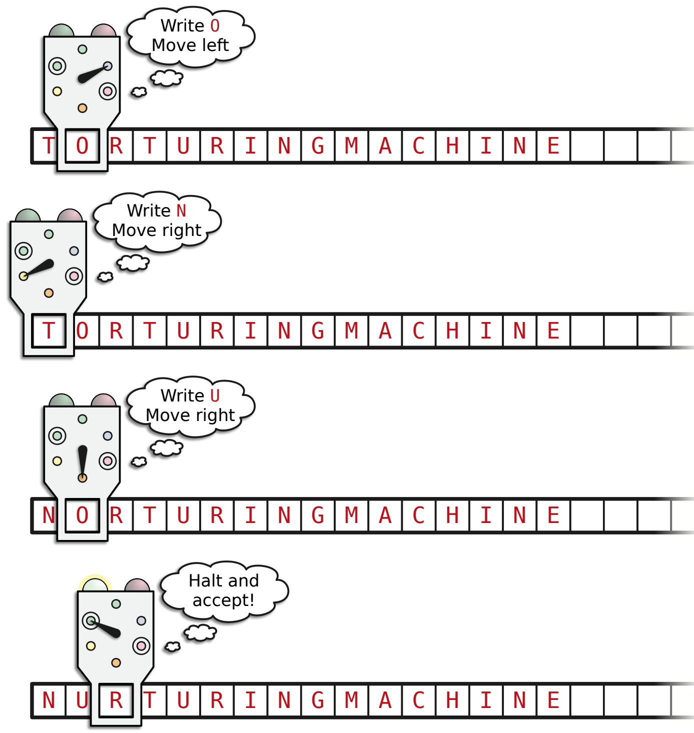
- A simple formal model of mechanical computation now known as *Turing machines*.
- A description of a single *universal* machine that can be used to compute *any* function computable by *any* other Turing machine.
- A proof that no Turing machine can solve the *halting problem*—Given the formal description of an arbitrary Turing machine M, does M halt or run forever?
- A proof that no Turing machine can determine whether an arbitrary given proposition is provable from the axioms of first-order logic. This Hilbert and Ackermann's famous *Entscheidungsproblem* ("decision problem")
- Compelling arguments[1] that his machines can execute *arbitrary* "calculation by finite means".

Turing's paper was not the first to prove that the *Entscheidungsproblem* had no algorithmic solution. Alonzo Church published the first proof just a new months earlier, using a very different model of computation, now called the ***untyped λ-calculus***. Turing and Church developed their results independently; indeed, Turing rushed the submission of his own paper immediately after receiving a copy of Church's paper, pausing only long enough to prove that any function computable via λ-calculus can also be computed by a Turing machine and vice versa. Church was the referee for Turing's paper; between the paper's submission and its acceptance, Turing was admitted to Princeton, where he became Church's PhD student. He finished his PhD two years later.

Informally, Turing described a device with a finite number of ***internal states*** that has access to memory in the form of a ***tape***. The tape consists of a semi-infinite sequence of ***cells***, each

[1]As Turing put it, "All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically." The claim that anything that can be computed can be computing using Turing machines is now known as the ***Church-Turing thesis***.

containing a single symbol from some arbitrary finite alphabet. The Turing machine can access the tape only through its ***head***, which is positioned over a single cell. Initially, the tape contains an arbitrary finite ***input string*** followed by an infinite sequence of ***blanks***, and the head is positioned over the first cell on the tape. In a single iteration, the machine reads the symbol in that cell, possibly write a new symbol into that cell, possibly changes its internal state, possibly moves the head to a neighboring cell, and possibly halts. The precise behavior of the machine at each iteration is entirely determined by its internal state and the symbol that it reads. When the machine halts, it indicates whether it has ***accepted*** or ***rejected*** the original input string.



A few iterations of a six-state Turing machine.

6.1 Why Bother?

Students used to thinking of computation in terms of higher-level operations like random memory accesses, function calls, and recursion may wonder why we should even consider a model as simple and constrained as Turing machines. Admittedly, Turing machines are a terrible model for thinking about ***fast*** computation; simple operations that take constant time in the standard random-access model can require *arbitrarily* many steps on a Turing machine. Worse, seemingly minor variations in the precise definition of "Turing machine" can have significant impact on problem complexity. As a simple example (which will make more sense later), we can reverse a string of n bits in $O(n)$ time using a two-tape Turing machine, but the same task provably requires $\Omega(n^2)$ time on a single-tape machine.

But here we are not interested in finding *fast* algorithms, or indeed in finding algorithms at all, but rather in proving that some problems cannot be solved by *any* computational means. Such a bold claim requires a formal definition of "computation" that is simple enough to support formal argument, but still powerful enough to describe arbitrary algorithms. Turing machines are ideal for this purpose. In particular, Turing machines are powerful enough to *simulate other*

Turing machines, while still simple enough to let us build up this self-simulation from scratch, unlike more complex but efficient models like the standard random-access machine

(Arguably, self-simulation is even simpler in Church's λ-calculus, or in Schönfinkel and Curry's combinator calculus, which is one of many reasons those models are more common in the design and analysis of programming languages than Turing machines. Those models much more abstract; in particular, they are harder to show equivalent to standard iterative models of computation.)

6.2 Formal Definitions

Formally, a Turing machine consists of the following components. (Hang on; it's a long list.)

- An arbitrary finite set Γ with at least two elements, called the ***tape alphabet***.
- An arbitrary symbol $\square \in \Gamma$, called the ***blank symbol*** or just the ***blank***.
- An arbitrary nonempty subset $\Sigma \subseteq (\Gamma \setminus \{\square\})$, called the ***input alphabet***.
- Another arbitrary finite set Q whose elements are called ***states***.
- Three distinct special states $\mathsf{start}, \mathsf{accept}, \mathsf{reject} \in Q$.
- A ***transition*** function $\delta \colon (Q \setminus \{\mathsf{accept}, \mathsf{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.

A ***configuration*** or ***global state*** of a Turing machine is represented by a triple $(q, x, i) \in Q \times \Gamma^* \times \mathbb{N}$, indicating that the machine's internal state is q, the tape contains the string x followed by an infinite sequence of blanks, and the head is located at position i. Trailing blanks in the tape string are ignored; the triples (q, x, i) and $(q, x\square, i)$ describe exactly the same configuration.

The transition function δ describes the evolution of the machine. For example, $\delta(q, a) = (p, b, -1)$ means that when the machine reads symbol a in state q, it changes its internal state to p, writes symbol b onto the tape at its current location (replacing a), and then decreases its position by 1 (or more intuitively, moves one step to the left). If the position of the head becomes negative, no further transitions are possible, and the machine ***crashes***.

We write $\boldsymbol{(p, x, i) \Rightarrow_M (q, y, j)}$ to indicate that Turing machine M transitions from the first configuration to the second in one step. (The symbol $\Rightarrow$ is often pronounced "yields"; I will omit the subscript M if the machine is clear from context.) For example, $\delta(p, a) = (q, b, \pm 1)$ means that

$$(p, xay, i) \;\Rightarrow\; (q, xby, i \pm 1)$$

for any non-negative integer i, any string x of length i, and any string y. The evolution of any Turing machine is ***deterministic***; each configuration C yields a *unique* configuration C'. We write $\boldsymbol{C \Rightarrow^* C'}$ to indicate that there is a (possibly empty) sequence of transitions from configuration C to configuration C'. (The symbol $\Rightarrow^*$ can be pronounced "eventually yields".)

The initial configuration is $(w, \mathsf{start}, 0)$ for some arbitrary (and possibly empty) ***input string*** $w \in \Sigma^*$. If M eventually reaches the accept state—more formally, if $(w, \mathsf{start}, 0) \Rightarrow^* (x, \mathsf{accept}, i)$ for some string $x \in \Gamma^*$ and some integer i—we say that M ***accepts*** the original input string w. Similarly, if M eventually reaches the reject state, we say that M ***rejects*** w. We must emphasize that "rejects" and "does not accept" are *not* synonyms; if M crashes or runs forever, then M neither accepts nor rejects w.

We distinguish between two different senses in which a Turing machine can "accept" a language. Let M be a Turing machine with input alphabet Σ, and let $L \subseteq \Sigma^*$ be an arbitrary language over Σ.

- M ***recognizes*** or ***accepts*** L if and only if M accepts every string in L but nothing else. A language is ***recognizable*** (or *semi-computable* or *recursively enumerable*) if it is recognized by some Turing machine.

- M ***decides*** L if and only if M accepts every string in L and rejects every string in $\Sigma^* \setminus L$. Equivalently, M decides L if and only if M recognizes L and halts (without crashing) on all inputs. A language is ***decidable*** (or *computable* or *recursive*) if it is decided by some Turing machine.

Trivially, every decidable language is recognizable, but (as we will see later), not every recognizable language is decidable.

6.3 A First Example

Consider the language $L = \{\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$. This language is neither regular nor context-free, but it can be decided by the following six-state Turing machine. The alphabets and states of the machine are defined as follows:

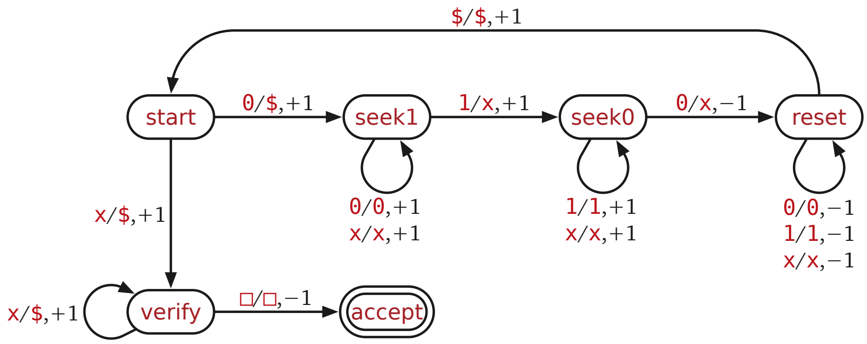
$$\Gamma = \{\mathtt{0}, \mathtt{1}, \mathtt{\$}, \mathtt{x}, \square\}$$
$$\Sigma = \{\mathtt{0}, \mathtt{1}\}$$
$$Q = \{\mathsf{start}, \mathsf{seek1}, \mathsf{seek0}, \mathsf{reset}, \mathsf{verify}, \mathsf{accept}, \mathsf{reject}\}$$

The transition function is described in the following table; all unspecified transitions lead to the reject state. We also give a graphical representation of the same machine, which resembles a drawing of a DFA, but with output symbols and actions specified on each edge. For example, we indicate the transition $\delta(p, \mathtt{0}) = (q, \mathtt{1}, +1)$ by writing $\mathtt{0}/\mathtt{1}, +1$ next to the arrow from state p to state q.

$\delta(\ p\ , a) = (\ q\ , b, \Delta)$	explanation
$\delta($start$, \mathtt{0}) = ($seek1$, \mathtt{\$}, +1)$	mark first 0 and scan right
$\delta($start$, \mathtt{x}) = ($verify$, \mathtt{\$}, +1)$	looks like we're done, but let's make sure
$\delta($seek1$, \mathtt{0}) = ($seek1$, \mathtt{0}, +1)$	scan rightward for 1
$\delta($seek1$, \mathtt{x}) = ($seek1$, \mathtt{x}, +1)$	
$\delta($seek1$, \mathtt{1}) = ($seek0$, \mathtt{x}, +1)$	mark 1 and continue right
$\delta($seek0$, \mathtt{1}) = ($seek0$, \mathtt{1}, +1)$	scan rightward for 0
$\delta($seek0$, \mathtt{x}) = ($seek0$, \mathtt{x}, +1)$	
$\delta($seek0$, \mathtt{0}) = ($reset$, \mathtt{x}, +1)$	mark 0 and scan left
$\delta($reset$, \mathtt{0}) = ($reset$, \mathtt{0}, -1)$	scan leftward for \$
$\delta($reset$, \mathtt{1}) = ($reset$, \mathtt{1}, -1)$	
$\delta($reset$, \mathtt{x}) = ($reset$, \mathtt{x}, -1)$	
$\delta($reset$, \mathtt{\$}) = ($start$, \mathtt{\$}, +1)$	step right and start over
$\delta($verify$, \mathtt{x}) = ($verify$, \mathtt{\$}, +1)$	scan right for any unmarked symbol
$\delta($verify$, \square) = ($accept$, \square, -1)$	success!

The transition function for a Turing machine that decides the language $\{\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$.

Finally, we trace the execution of this machine on two input strings: $\mathtt{001100} \in L$ and $\mathtt{00100} \notin L$. In each configuration, we indicate the position of the head using a small triangle



A graphical representation of the example Turing machine

instead of listing the position explicitly. Notice that we automatically add blanks to the tape string as necessary. Proving that this machine actually decides L—and in particular, that it never crashes or infinite-loops—is a straightforward but tedious exercise in induction.

$$
\begin{aligned}
&(\mathsf{start}, \mathtt{\underset{\blacktriangle}{0}01100}) \Rightarrow (\mathsf{seek1}, \mathtt{\$\underset{\blacktriangle}{0}1100}) \Rightarrow (\mathsf{seek1}, \mathtt{\$0\underset{\blacktriangle}{1}100}) \Rightarrow (\mathsf{seek0}, \mathtt{\$0x\underset{\blacktriangle}{1}00}) \Rightarrow (\mathsf{seek0}, \mathtt{\$0x1\underset{\blacktriangle}{0}0})\\
&\Rightarrow (\mathsf{reset}, \mathtt{\$0x\underset{\blacktriangle}{1}x0}) \Rightarrow (\mathsf{reset}, \mathtt{\$0\underset{\blacktriangle}{x}1x0}) \Rightarrow (\mathsf{reset}, \mathtt{\$\underset{\blacktriangle}{0}x1x0}) \Rightarrow (\mathsf{reset}, \mathtt{\underset{\blacktriangle}{\$}0x1x0})\\
&\Rightarrow (\mathsf{start}, \mathtt{\$\underset{\blacktriangle}{0}x1x0})\\
&\Rightarrow (\mathsf{seek1}, \mathtt{\$\$\underset{\blacktriangle}{x}1x0}) \Rightarrow (\mathsf{seek1}, \mathtt{\$\$x\underset{\blacktriangle}{1}x0}) \Rightarrow (\mathsf{seek0}, \mathtt{\$\$xx\underset{\blacktriangle}{x}0}) \Rightarrow (\mathsf{seek0}, \mathtt{\$\$xxx\underset{\blacktriangle}{0}})\\
&\Rightarrow (\mathsf{reset}, \mathtt{\$\$xx\underset{\blacktriangle}{x}x}) \Rightarrow (\mathsf{reset}, \mathtt{\$\$x\underset{\blacktriangle}{x}xx}) \Rightarrow (\mathsf{reset}, \mathtt{\$\$\underset{\blacktriangle}{x}xxx}) \Rightarrow (\mathsf{reset}, \mathtt{\$\underset{\blacktriangle}{\$}xxxx})\\
&\Rightarrow (\mathsf{verify}, \mathtt{\$\$\underset{\blacktriangle}{x}xxx}) \Rightarrow (\mathsf{verify}, \mathtt{\$\$\$\underset{\blacktriangle}{x}xx}) \Rightarrow (\mathsf{verify}, \mathtt{\$\$\$\$\underset{\blacktriangle}{x}x})\\
&\Rightarrow (\mathsf{verify}, \mathtt{\$\$\$\$\$\underset{\blacktriangle}{x}}) \Rightarrow (\mathsf{verify}, \mathtt{\$\$\$\$\$\$\underset{\blacktriangle}{\square}}) \Rightarrow (\mathsf{accept}, \mathtt{\$\$\$\$\$\underset{\blacktriangle}{\$}}) \Rightarrow \textbf{accept!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string $\mathtt{001100} \in L$

$$
\begin{aligned}
&(\mathsf{start}, \mathtt{\underset{\blacktriangle}{0}0100}) \Rightarrow (\mathsf{seek1}, \mathtt{\$\underset{\blacktriangle}{0}100}) \Rightarrow (\mathsf{seek1}, \mathtt{\$0\underset{\blacktriangle}{1}00}) \Rightarrow (\mathsf{seek0}, \mathtt{\$0x\underset{\blacktriangle}{0}0})\\
&\Rightarrow (\mathsf{reset}, \mathtt{\$0\underset{\blacktriangle}{x}x0}) \Rightarrow (\mathsf{reset}, \mathtt{\$\underset{\blacktriangle}{0}xx0}) \Rightarrow (\mathsf{reset}, \mathtt{\underset{\blacktriangle}{\$}0xx0})\\
&\Rightarrow (\mathsf{start}, \mathtt{\$\underset{\blacktriangle}{0}xx0})\\
&\Rightarrow (\mathsf{seek1}, \mathtt{\$\$\underset{\blacktriangle}{x}x0}) \Rightarrow (\mathsf{seek1}, \mathtt{\$\$x\underset{\blacktriangle}{x}0}) \Rightarrow (\mathsf{seek1}, \mathtt{\$\$xx\underset{\blacktriangle}{0}}) \Rightarrow \textbf{reject!}
\end{aligned}
$$

The evolution of the example Turing machine on the input string $\mathtt{00100} \notin L$

6.4 Variations

There are actually several formal models that all fall under the name "Turing machine", each with small variations on the definition we've given. Although we do need to be explicit about *which* variant we want to use for any particular problem, the differences between the variants are relatively unimportant. For any machine defined in one model, there is an equivalent machine in each of the other models; in particular, all of these variants recognize the same languages and decide the same languages. For example:

- **Halting conditions.** Some models allow multiple accept and reject states, which (depending on the precise model) trigger acceptance or rejection either when the machine enters the state, or when the machine has no valid transitions out of such a state. Others include only explicit accept states, and either equate crashing with rejection or do not define a rejection mechanism at all. Still other models include halting as one of the possible *actions* of the machine, in addition to moving left or moving right; in these models, the machine accepts/rejects its input if and only if it halts in an accepting/non-accepting state.

- **Actions.** Some Turing machine models allow transitions that do not move the head, or that move the head by more than one cell in a single step. Others insist that a single step of the machine *either* writes a new symbol onto the tape *or* moves the head one step. Finally, as mentioned above, some models include halting as one of the available actions.

- **Transition function.** Some models of Turing machines, including Turing's original definition, allow the transition function to be undefined on some state-symbol pairs. In this formulation, the transition function is given by a set $\delta \subset Q \times \Gamma \times Q \times \Gamma \times \{+1, -1\}$, such that for each state q and symbol a, there is at most one transition $(q, a, \cdot, \cdot, \cdot) \in \delta$. If the machine enters a configuration from which there is no transition, it halts and (depending on the precise model) either crashes or rejects. Others define the transition function as $\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{-1, +1\})$, allowing the machine to *either* write a symbol to the tape *or* move the head in each step.

- **Beginning of the tape.** Some models forbid the head to move past the beginning of the tape, either by starting the tape with a special symbol that cannot be overwritten and that forces a rightward transition, or by declaring that a leftward transition at position 0 leaves the head in position 0, or even by pure fiat—declaring any machine that performs a leftward move at position 0 to be invalid.

To prove that any two of these variant "species" of Turing machine are equivalent, we must show how to transform a machine of one species into a machine of the other species that accepts and rejects the same strings. For example, let $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ be a Turing machine with explicit accept and reject states. We can define an equivalent Turing machine M' that halts only when it moves left from position 0, and accepts only by halting while in an accepting state, as follows. We define the set of accepting states for M' as $A = \{\mathsf{accept}\}$ and define a new transition function

$$\delta'(q, a) := \begin{cases} (\mathsf{accept}, a, -1) & \text{if } q = \mathsf{accept} \\ (\mathsf{reject}, a, -1) & \text{if } q = \mathsf{reject} \\ \delta(q, a) & \text{otherwise} \end{cases}$$

Similarly, suppose someone gives us a Turing machine $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ whose transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ allows the machine to transition without moving its head. We can construct an equivalent Turing machine $M' = (\Gamma, \square, \Sigma, Q', s, \mathsf{accept}, \mathsf{reject}, \delta')$ that moves its head at every transition by defining $Q' := Q \times \{0, 1\}$ and

$$\delta'((p, 0), a) := \begin{cases} ((q, 1), b, +1) & \text{if } \delta(p, a) = (q, b, 0), \\ ((q, 0), b, \Delta) & \text{if } \delta(p, a) = (q, b, \Delta) \text{ and } \Delta \neq 0, \end{cases}$$
$$\delta'((p, 1), a) := ((p, 0), a, -1).$$

6.5 Computing Functions

Turing machines can also be used to compute functions from strings to strings, instead of just accepting or rejecting strings. Since we don't care about acceptance or rejection, we replace the explicit accept and reject states with a single halt state, and we define the ***output*** of the Turing machine to be the contents of the tape when the machine halts, after removing the infinite sequence of trailing blanks. More formally, for any Turing machine M, any string $w \in \Sigma^*$, and any string $x \in \Gamma^*$ that does not end with a blank, we write $\boldsymbol{M(w) = x}$ if and only if $(w, s, 0) \Rightarrow_M^* (x, \mathsf{halt}, i)$ for some integer i. If M does not halt on input w, then we write $\boldsymbol{M(w)\nearrow}$, which can be read either "M diverges on w" or "$M(w)$ is undefined." We say that M ***computes*** the function $f : \Sigma^* \to \Sigma^*$ if and only if $M(w) = f(w)$ for every string w.

6.5.1 Shifting

One basic operation that is used in many Turing machine constructions is ***shifting*** the input string a constant number of steps to the right or to the left. For example, given any input string $w \in \{0, 1\}^*$, we can compute the string $0w$ using a Turing machine with tape alphabet $\Gamma = \{0, 1, \square\}$, state set $Q = \{0, 1, \mathsf{halt}\}$, start state 0, and the following transition function:

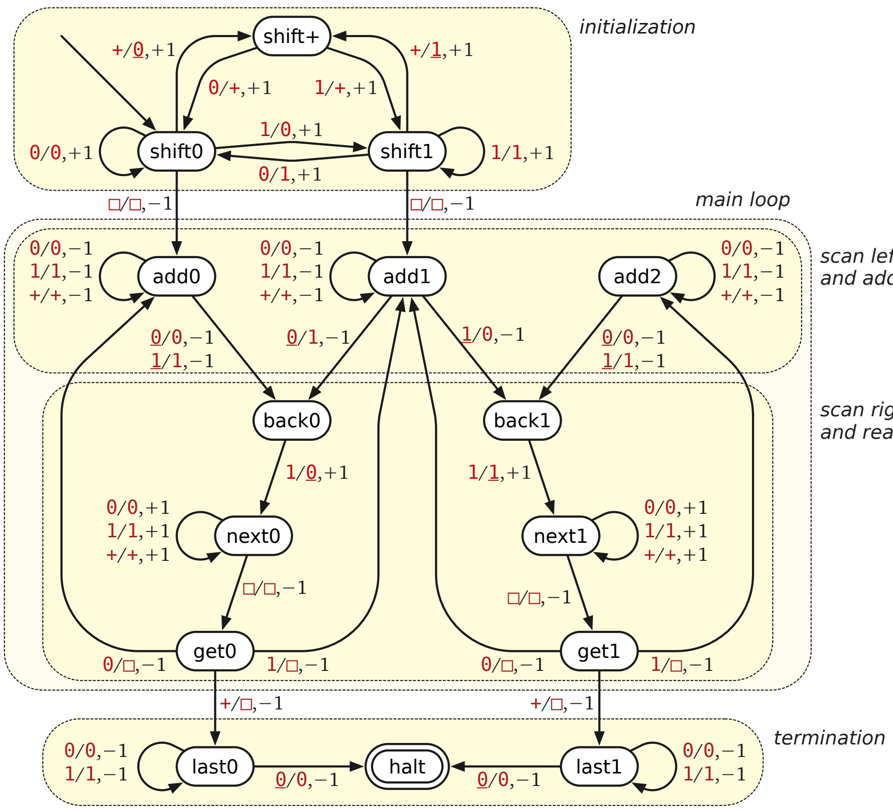
$\delta(p, a)$	$=$	(q, b, Δ)
$\delta(0, 0)$	$=$	$(0, 0, +1)$
$\delta(0, 1)$	$=$	$(1, 0, +1)$
$\delta(0, \square)$	$=$	$(\mathsf{halt}, 0, +1)$
$\delta(1, 0)$	$=$	$(0, 1, +1)$
$\delta(1, 1)$	$=$	$(1, 1, +1)$
$\delta(1, \square)$	$=$	$(\mathsf{halt}, 1, +1)$

By increasing the number of states, we can build a Turing machine that shifts the input string any fixed number of steps in either direction. For example, a machine that shifts its input to the left by five steps might read the string from right to left, storing the five most recently read symbols in its internal state. A typical transition for such a machine would be $\delta(12345, 0) = (01234, 5, -1)$.

6.5.2 Binary Addition

With a more complex Turing machine, we can implement binary addition. The input is a string of the form $w{+}x$, where $w, x \in \{0, 1\}^n$, representing two numbers in binary; the output is the binary representation of $w + x$. To simplify our presentation, we assume that $|w| = |x| > 0$; however, this restrictions can be removed with the addition of a few more states. The following figure shows the entire Turing machine at a glance. The machine uses the tape alphabet $\Gamma = \{\square, 0, 1, +, \underline{0}, \underline{1}\}$; the start state is shift0. All missing transitions go to a fail state, indicating that the input was badly formed.

Execution of this Turing machine proceeds in several phases, each with its own subset of states, as indicated in the figure. The initialization phase scans the entire input, shifting it to the right to make room for the output string, marking the rightmost bit of w, and reading and erasing the last bit of x.



A Turing machine that adds two binary numbers of the same length.

$\delta(p,\ a)$	$=$	$(q,\ b,\ \Delta)$
$\delta(\mathsf{shift0},\ \mathtt{0})$	$=$	$(\mathsf{shift0},\ \mathtt{0},\ +1)$
$\delta(\mathsf{shift0},\ \mathtt{1})$	$=$	$(\mathsf{shift1},\ \mathtt{0},\ +1)$
$\delta(\mathsf{shift0},\ \mathtt{+})$	$=$	$(\mathsf{shift+},\ \underline{\mathtt{0}},\ +1)$
$\delta(\mathsf{shift0},\ \square)$	$=$	$(\mathsf{add0},\ \square,\ -1)$
$\delta(\mathsf{shift1},\ \mathtt{0})$	$=$	$(\mathsf{shift0},\ \mathtt{1},\ +1)$
$\delta(\mathsf{shift1},\ \mathtt{1})$	$=$	$(\mathsf{shift1},\ \mathtt{1},\ +1)$
$\delta(\mathsf{shift1},\ \mathtt{+})$	$=$	$(\mathsf{shift+},\ \underline{\mathtt{1}},\ +1)$
$\delta(\mathsf{shift1},\ \square)$	$=$	$(\mathsf{add1},\ \square,\ -1)$
$\delta(\mathsf{shift+},\ \mathtt{0})$	$=$	$(\mathsf{shift0},\ \mathtt{+},\ +1)$
$\delta(\mathsf{shift+},\ \mathtt{1})$	$=$	$(\mathsf{shift1},\ \mathtt{+},\ +1)$

The first part of the main loop scans left to the marked bit of w, adds the bit of x that was just erased plus the carry bit from the previous iteration, and records the carry bit for the next iteration in the machines internal state.

$\delta(p,\ a)$	$=$	$(q,\ b,\ \Delta)$
$\delta(\mathsf{add0},\ \mathtt{0})$	$=$	$(\mathsf{add0},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add0},\ \mathtt{1})$	$=$	$(\mathsf{add0},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add0},\ \mathtt{+})$	$=$	$(\mathsf{add0},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add0},\ \underline{\mathtt{0}})$	$=$	$(\mathsf{back0},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add0},\ \underline{\mathtt{1}})$	$=$	$(\mathsf{back0},\ \mathtt{1},\ -1)$

$\delta(p,\ a)$	$=$	$(q,\ b,\ \Delta)$
$\delta(\mathsf{add1},\ \mathtt{0})$	$=$	$(\mathsf{add1},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add1},\ \mathtt{1})$	$=$	$(\mathsf{add1},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add1},\ \mathtt{+})$	$=$	$(\mathsf{add1},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add1},\ \underline{\mathtt{0}})$	$=$	$(\mathsf{back0},\ \mathtt{1},\ -1)$
$\delta(\mathsf{add1},\ \underline{\mathtt{1}})$	$=$	$(\mathsf{back1},\ \mathtt{0},\ -1)$

$\delta(p,\ a)$	$=$	$(q,\ b,\ \Delta)$
$\delta(\mathsf{add2},\ \mathtt{0})$	$=$	$(\mathsf{add2},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add2},\ \mathtt{1})$	$=$	$(\mathsf{add2},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add2},\ \mathtt{+})$	$=$	$(\mathsf{add2},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add2},\ \underline{\mathtt{0}})$	$=$	$(\mathsf{back1},\ \mathtt{0},\ -1)$
$\delta(\mathsf{add2},\ \underline{\mathtt{1}})$	$=$	$(\mathsf{back1},\ \mathtt{1},\ -1)$

The second part of the main loop marks the previous bit of w, scans right to the end of x, and then reads and erases the last bit of x, all while maintaining the carry bit.

$\delta(p, a)$	$= (q, b, \Delta)$
$\delta(\text{back0}, 0)$	$= (\text{next0}, \underline{0}, +1)$
$\delta(\text{back0}, 1)$	$= (\text{next0}, \underline{1}, +1)$
$\delta(\text{next0}, 0)$	$= (\text{next0}, 0, +1)$
$\delta(\text{next0}, 1)$	$= (\text{next0}, 0, +1)$
$\delta(\text{next0}, +)$	$= (\text{next0}, 0, +1)$
$\delta(\text{next0}, \square)$	$= (\text{get0}, \square, -1)$
$\delta(\text{get0}, 0)$	$= (\text{add0}, \square, -1)$
$\delta(\text{get0}, 1)$	$= (\text{add1}, \square, -1)$
$\delta(\text{get0}, +)$	$= (\text{last0}, \square, -1)$

$\delta(p, a)$	$= (q, b, \Delta)$
$\delta(\text{back1}, 0)$	$= (\text{next1}, \underline{0}, +1)$
$\delta(\text{back1}, 1)$	$= (\text{next1}, \underline{1}, +1)$
$\delta(\text{next1}, 0)$	$= (\text{next1}, 0, +1)$
$\delta(\text{next1}, 1)$	$= (\text{next1}, 0, +1)$
$\delta(\text{next1}, +)$	$= (\text{next1}, 0, +1)$
$\delta(\text{next1}, \square)$	$= (\text{get1}, \square, -1)$
$\delta(\text{get1}, 0)$	$= (\text{add1}, \square, -1)$
$\delta(\text{get1}, 1)$	$= (\text{add2}, \square, -1)$
$\delta(\text{get1}, +)$	$= (\text{last1}, \square, -1)$

Finally, after erasing the + in the last iteration of the main loop, the termination phase adds the last carry bit to the leftmost output bit and halts.

$\delta(p, a)$	$= (q, b, \Delta)$
$\delta(\text{last0}, 0)$	$= (\text{last0}, 0, -1)$
$\delta(\text{last0}, 1)$	$= (\text{last0}, 0, -1)$
$\delta(\text{last0}, \underline{0})$	$= (\text{halt}, 0, \quad)$
$\delta(\text{last1}, 0)$	$= (\text{last1}, 0, -1)$
$\delta(\text{last1}, 1)$	$= (\text{last1}, 0, -1)$
$\delta(\text{last1}, \underline{0})$	$= (\text{halt}, 1, \quad)$

6.6 Variations on Tracks, Heads, and Tapes

Multiple Tracks

It is sometimes convenient to endow the Turing machine tape with multiple ***tracks***, each with its own tape alphabet, and allow the machine to read from and write to the same position on all tracks simultaneously. For example, to define a Turing machine with three tracks, we need three tape alphabets Γ_1, Γ_2, and Γ_3, each with its own blank symbol, where (say) Γ_1 contains the input alphabet Σ as a subset; we also need a transition function of the form

$$\delta\colon Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \to Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \times \{-1, +1\}$$

Describing a configuration of this machine requires a quintuple (q, x_1, x_2, x_3, i), indicating that each track i contains the string x_i followed by an infinite sequence of blanks. The initial configuration is $(\text{start}, w, \varepsilon, \varepsilon, 0)$, with the input string written on the first track, and the other two tracks completely blank.

But any such machine is equivalent (if not *identical*) to a single-track Turing machine with the (still finite!) tape alphabet $\Gamma := \Gamma_1 \times \Gamma_2 \times \Gamma_3$. Instead of thinking of the tape as three infinite sequences of symbols, we think of it as a single infinite sequence of "records", each containing three symbols. Moreover, there's nothing special about the number 3 in this construction; a Turing machine with *any* constant number of tracks is equivalent to a single-track machine.

Doubly-Infinite Tape

It is also sometimes convenient to allow the tape to be infinite in both directions, for example, to avoid boundary conditions. There are several ways to simulate a doubly-infinite tape on a machine with only a semi-infinite tape. Perhaps the simplest method is to use a semi-infinite tape with two tracks, one containing the cells with positive index and the other containing the cells with negative index in reverse order, with a special marker symbol at position zero to indicate the transition.

0	+1	+2	+3	+4	⋯
►	−1	−2	−3	−4	⋯

Another method is to shuffle the positive-index and negative-index cells onto a single track, and add additional states to allow the Turing machine to move two steps in a single transition. Again, we need a special symbol at the left end of the tape to indicate the transition:

►	0	−1	+1	−2	+2	−3	+3	⋯

A third method maintains two sentinel symbols ► and ◄ that surround all other non-blank symbols on the tape. Whenever the machine reads the right sentinel ◄, we write a blank, move right, write ◄, move left, and then proceed as if we had just read a blank. On the other hand, when the machine reads the left sentinel ►, we shift the entire contents of the tape (up to and including the right sentinel) one step to the right, then move back to the left sentinel, move right, write a blank, and finally proceed as if we had just read a blank. Since the Turing machine does not actually have access to the position of the head *as an integer*, shifting the head and the tape contents one step right has no effect on its future evolution.

►	−3	−2	−1	0	+1	+2	+3	+4	+5	◄	⋯

Using either of the first two methods, we can simulate t steps of an arbitrary Turing machine with a doubly-infinite tape using only $O(t)$ steps on a standard Turing machine. The third method, unfortunately, requires $\Theta(t^2)$ steps in the worst case.

Insertion and Deletion

We can also allow Turing machines to insert and delete cells on the tape, in addition to simply overwriting existing symbols. We've already seen how to insert a new cell: Leave a special mark on the tape (perhaps in a second track), shift everything to the right of this mark one cell to the right, scan left to the mark, erase the mark, and finally write the correct character into the new cell. Deletion is similar: Mark the cell to be deleted, shift everything to the right of the mark one step to the left, scan left to the mark, and erase the mark. We may also need to maintain a mark in some cell to the right every non-blank symbol, indicating that all cells further to the right are blank, so that we know when to stop shifting left or right.

Multiple Heads

Another convenient extension is to allow machines simultaneous access to more than one position on the tape. For example, to define a Turing machine with *three* heads, we need a transition function of the form

$$\delta : Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{-1, +1\}^3.$$

Describing a configuration of such a machine requires a quintuple (q, x, i, j, k), indicating that the machine is in state q, the tape contains string x, and the three heads are at positions i, j, k. The transition function tells us, given q and the three symbols $x[i], x[j], x[k]$, which three symbols to write on the tape and which direction to move each of the heads.

We can simulate this behavior with a single head by adding additional tracks to the tape that record the positions of each head. To simulate a machine M with three heads, we use a tape with four tracks: track 0 is the actual work tape; each of the remaining tracks has a single non-blank symbol recording the position of one of the heads. We also insert a special marker symbols at the left end of the tape.

▶	M	Y	W	O	R	K	T	A	P	E	···
▶									▲		···
▶		▲									···
▶						▲					···

We can simulate any single transition of M, starting with our single head at the left end of the tape, as follows. Throughout the simulation, we maintain the internal state of M as one of the components of our current state. First, for each i, we read the symbol under the ith head of M as follows:

> Scan to the right to find the mark on track i, read the corresponding symbol from track 0 into our internal state, and then return to the left end of the tape.

At this point, our internal state records M's current internal state and the three symbols under M's heads. After one more transition (using M's transition function), our internal state records M's *next* state, the symbol to be written by each head, and the direction to move each head. Then, for each i, we write with and move the ith head of M as follows:

> Scan to the right to find the mark on track i, write the correct symbol onto on track 0, move the mark on track i one step left or right, and then return to the left end of the tape.

Again, there is nothing special about the number 3 here; we can simulate machines with *any* fixed number of heads.

Careful analysis of this technique implies that for any integer k, we can simulate t steps of an arbitrary Turing machine with k independent heads in $\Theta(t^2)$ time on a standard Turing machine with only one head. Unfortunately, this quadratic blowup is unavoidable. It is relatively easy to recognize the language of *marked palindromes* $\{w \bullet w^R \mid w \in \{0,1\}^*\}$ in $O(n)$ time using a Turing machine with two heads, but recognizing this language provably requires $\Omega(n^2)$ time on a standard machine with only one head. On the other hand, with much more sophisticated techniques, it is possible to simulate t steps of a Turing machine with k head, for any fixed integer k, using only $O(t \log t)$ steps on a Turing machine with just *two* heads.

Multiple Tapes

We can also allow machines with multiple independent tapes, each with its own head. To simulate such a machine with a single tape, we simply maintain each tape as an independent track with its own head. Equivalently, we can simulate a machine with k tapes using a single tape with $2k$ tracks, half storing the contents of the k tapes and half storing the positions of the k heads.

▶	T	A	P	E	#	O	N	E			···
▶									▲		···
▶	T	A	P	E	#	T	W	O			···
▶			▲								···
▶	T	A	P	E	#	T	H	R	E	E	···
▶						▲					···

Just as for multiple tracks, for any constant k, we can simulate t steps of an arbitrary Turing machine with k independent tapes in $\Theta(t^2)$ steps on a standard Turing machine with one tape, and this quadratic blowup is unavoidable. Moreover, it is possible to simulate t steps on a k-tape Turing machine using only $O(t \log t)$ steps on a *two*-tape Turing machine using more sophisticated techniques. (This faster simulation is easier to obtain for multiple independent tapes than for multiple heads on the same tape.)

By combining these tricks, we can simulate a Turing machine with any fixed number of tapes, each of which may be infinite in one or both directions, each with any fixed number of heads and any fixed number of tracks, with at most a quadratic blowup in the running time.

6.7 Simulating a Real Computer

6.7.1 Subroutines and Recursion

★★★

Use a second tape/track as a "call stack". Add save and restore actions. In the simplest formulation, subroutines do not have local memory. To call a subroutine, save the current state onto the call stack and jump to the first state of the subroutine. To return, restore (and remove) the return state from the call stack. We can simulate t steps of any recursive Turing machine with $O(t)$ steps on a multitape standard Turing machine, or in $O(t^2)$ steps on a standard Turing machine.

More complex versions of this simulation can adapt to

6.7.2 Random-Access Memory

★★★

Keep [address•data] pairs on a separate "memory" tape. Write address to an "address" tape; read data from or write data to a "data" tape. Add new or changed [address•data] pairs at the end of the memory tape. (Semantics of reading from an address that has never been written to?)

Suppose all memory accesses require at most ℓ address and data bits. Then we can simulate the kth memory access in $O(k\ell)$ steps on a multitape Turing machine or in $O(k^2\ell^2)$ steps on a single-tape machine. Thus, simulating t memory accesses in a random-access machine with ℓ-bit words requires $O(t^2\ell)$ time on a multitape Turing machine, or $O(t^3\ell^2)$ time on a single-tape machine.

6.8 Universal Turing Machines

With all these tools in hand, we can now describe the pinnacle of Turing machine constructions: the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine U is a string $\langle M, w\rangle$ that encodes both an arbitrary Turing machine M and a string w in the input alphabet of M. Given these encodings, U simulates the execution of M on input w; in particular,

- U accepts $\langle M, w\rangle$ if and only if M accepts w.
- U rejects $\langle M, w\rangle$ if and only if M rejects w.

In the next few pages, I will sketch a universal Turing machine U that uses the input alphabet {0, 1, [,], •, |} and a somewhat larger tape alphabet (via marks on additional tracks). However, I do *not* require that the Turing machines that U simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red states and tape symbols of U.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|) \rceil$ bits. Thus, if $\Gamma = \{\mathit{0}, \mathit{1}, \$, x, \square\}$, we might use the following encoding:

$$\langle \mathit{0} \rangle = \texttt{001}, \qquad \langle \mathit{1} \rangle = \texttt{010}, \qquad \langle \$ \rangle = \texttt{011}, \qquad \langle x \rangle = \texttt{100}, \qquad \langle \square \rangle = \texttt{000}.$$

The input string w is encoded by its sequence of symbol encodings, with separators • between every pair of symbols and with brackets **[** and **]** around the whole string. For example, with this encoding, the input string *001100* would be encoded on the input tape as

$$\langle \mathit{001100} \rangle = \texttt{[001•001•010•010•001•001]}$$

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q \rangle$ of $\lceil \lg |Q| \rceil$ bits. Without loss of generality, we encode the start state with all `1`s and the reject state with all `0`s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\langle \mathit{start} \rangle = \texttt{111} \qquad \langle \mathit{seek1} \rangle = \texttt{010} \qquad \langle \mathit{seek0} \rangle = \texttt{011} \qquad \langle \mathit{reset} \rangle = \texttt{100}$$
$$\langle \mathit{verify} \rangle = \texttt{101} \qquad \langle \mathit{accept} \rangle = \texttt{110} \qquad \langle \mathit{reject} \rangle = \texttt{000}$$

We encode the machine M itself as the string $\langle M \rangle = \texttt{[}\langle \mathit{reject} \rangle\texttt{•}\langle \square \rangle\texttt{]}\langle \delta \rangle$, where $\langle \delta \rangle$ is the concatenation of substrings $\texttt{[}\langle p \rangle\texttt{•}\langle a \rangle\texttt{|}\langle q \rangle\texttt{•}\langle b \rangle\texttt{•}\langle \Delta \rangle\texttt{]}$ encoding each transition $\delta(p, a) = (q, b, \Delta)$ such that $q \neq \text{reject}$. We encode the actions $\Delta = \pm 1$ by defining $\langle -1 \rangle := \texttt{0}$ and $\langle +1 \rangle := \texttt{1}$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \$) = (\mathit{start}, \$, +1)$ would be encoded as

$$\texttt{[100•011|001•011•1]}.$$

Our first example Turing machine for recognizing $\{\mathit{0}^n \mathit{1}^n \mathit{0}^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
          [010•001|010•001•1][010•100|010•100•1]
          [010•010|011•100•1][011•010|011•010•1]
          [011•100|011•100•1][011•001|100•100•1]
          [100•001|100•001•0][100•010|100•010•0]
          [100•100|100•100•0][100•011|001•011•1]
          [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of M on U's work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $\texttt{[}\langle q \rangle\texttt{•}\langle a \rangle\texttt{]}$ indicating that (1) the cell contains symbol a; (2) if $q \neq \mathit{reject}$, then M's head is located at this cell, and M is in state q; and (3) if $q = \mathit{reject}$, then M's head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets **[** and **]**.

For example, with the encodings described above, the initial configuration $(\mathit{start}, \underset{\blacktriangle}{\mathit{0}}\mathit{01100}, 0)$ for our first example Turing machine would be encoded on U's tape as follows.

$$\langle \mathit{start}, \underset{\blacktriangle}{\mathit{0}}\mathit{01100}, 0 \rangle = \texttt{[}\underbrace{\texttt{[111•001]}}_{\textbf{start } \mathit{0}}\underbrace{\texttt{[000•001]}}_{\mathit{reject}\ \mathit{0}}\underbrace{\texttt{[000•010]}}_{\mathit{reject}\ \mathit{1}}\underbrace{\texttt{[000•010]}}_{\mathit{reject}\ \mathit{1}}\underbrace{\texttt{[000•001]}}_{\mathit{reject}\ \mathit{0}}\underbrace{\texttt{[000•001]}}_{\mathit{reject}\ \mathit{0}}\texttt{]}$$

Similarly, the intermediate configuration ($\textit{reset}, \texttt{\$0x}\underset{\blacktriangle}{\texttt{1}}\texttt{x0}, 3$) would be encoded as follows:

$$\langle \textit{reset}, \texttt{\$\$x}\underset{\blacktriangle}{\texttt{1}}\texttt{x0}, 3\rangle = \texttt{[}\underbrace{\texttt{[000}\bullet\texttt{011]}}_{\textit{reject \$}}\underbrace{\texttt{[000}\bullet\texttt{011]}}_{\textit{reject 0}}\underbrace{\texttt{[000}\bullet\texttt{100]}}_{\textit{reject x}}\underbrace{\texttt{[010}\bullet\texttt{010]}}_{\textbf{\textit{reset}}\textit{ 1}}\underbrace{\texttt{[000}\bullet\texttt{100]}}_{\textit{reject x}}\underbrace{\texttt{[000}\bullet\texttt{001]}}_{\textit{reject 0}}\texttt{]}$$

Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine U is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine M followed by an encoding of its input string x. Notice the substrings **[[** and **]]** each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. U also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle \textit{start}, x, 0\rangle$ of the initial configuration of M with input x. First, we write $\texttt{[[}\langle \textit{start}\rangle\bullet$. Then we copy the encoded input string $\langle x\rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket **[**, write $\texttt{[[}\langle \textit{start}\rangle\bullet$.
- Instead of copying each separator $\bullet$, write $\texttt{][}\langle \textit{reject}\rangle\bullet$
- Instead of copying the right bracket **]**, write two right brackets **]]**.

The state encodings $\langle \textit{start}\rangle$ and $\langle \textit{reject}\rangle$ can be copied directly from the beginning of $\langle M\rangle$ (replacing `0`s for `1`s for $\langle \textit{start}\rangle$). Finally, we move the head back to the start of U's tape.

At the start of each step of the simulation, U's head is located at the start of the work tape. We scan through the work tape to the unique encoded cell $\texttt{[}\langle p\rangle\bullet\langle a\rangle\texttt{]}$ such that $p \neq \textit{reject}$. Then we scan through the encoded transition function $\langle\delta\rangle$ to find the unique encoded tuple $\texttt{[}\langle p\rangle\bullet\langle a\rangle\texttt{|}\langle q\rangle\bullet\langle b\rangle\bullet\langle\Delta\rangle\texttt{]}$ whose left half matches our the encoded tape cell. If there is no such tuple, then U immediately halts and rejects. Otherwise, we copy the right half $\langle q\rangle\bullet\langle b\rangle$ of the tuple to the work tape. Now if $q = \textit{accept}$, then U immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \neq \textit{reject}$.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by M's transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets **]]**, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with $\texttt{[}\langle \textit{reject}\rangle\bullet\langle\square\rangle\texttt{]]}$ (mostly copied from the beginning of the machine encoding $\langle M\rangle$) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks $\texttt{[}\langle \textit{reject}\rangle\bullet\langle\square\rangle\texttt{]}$ instead of *actual* blanks $\square$.

Example

As an illustrative example, suppose U is simulating our first example Turing machine M on the input string `001100`. The execution of M on input w eventually reaches the configuration ($\textit{seek1}, \texttt{\$\$x}\underset{\blacktriangle}{\texttt{1}}\texttt{x0}, 3$). At the start of the corresponding step in U's simulation, U is in the following configuration:

$$\underset{\blacktriangle}{\texttt{[}}\texttt{[000}\bullet\texttt{011][000}\bullet\texttt{011][000}\bullet\texttt{100][010}\bullet\texttt{010][000}\bullet\texttt{100][000}\bullet\texttt{001]]}$$

First U scans for the first encoded tape cell whose state is not *reject*. That is, U repeatedly compares the first half of each encoded state cell on the work tape with the prefix $\texttt{[}\langle \textit{reject}\rangle\bullet$ of the machine encoding $\langle M\rangle$ on the input tape. U finds a match in the fourth encoded cell.

$$\texttt{[[000}\bullet\texttt{011][000}\bullet\texttt{011][000}\bullet\texttt{100][010}\underset{\blacktriangle}{\bullet}\texttt{010][000}\bullet\texttt{100][000}\bullet\texttt{001]]}$$

Next, U scans the machine encoding $\langle M\rangle$ for the substring `[010•010` matching the current encoded cell. U eventually finds a match in the left size of the the encoded transition `[010•010|011•100•1]`. U copies the state-symbol pair `011•100` from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

`[[000•011][000•011][000•100][011•100][000•100][000•001]]`

The encoded transition instructs U to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

`[[000•011][000•011][000•100][000•100][011•100][000•001]]`

Finally, U scans left until it reads two left brackets `[[`; this returns the head to the left end of the work tape to start the next step in the simulation. U's tape now holds the encoding of M's configuration ($seek0$, $\$\$xxx0$, 4), as required.

`[[000•011][000•011][000•100][000•100][011•100][000•001]]`

Exercises

1. Describe Turing machines that decide each of the following languages:

   (a) Palindromes over the alphabet $\{0, 1\}$

   (b) $\{ww \mid w \in \{0,1\}^*\}$

   (c) $\left\{0^a 1^b 0^{ab} \mid a, b \in \mathbb{N}\right\}$

2. Let $\langle n\rangle_2$ denote the binary representation of the non-negative integer n. For example, $\langle 17\rangle_2 = 10001$ and $\langle 42\rangle_2 = 101010$. Describe Turing machines that compute the following functions from $\{0,1\}^*$ to $\{0,1\}^*$:

   (a) $w \mapsto www$

   (b) $1^n01^m \mapsto 1^{mn}$

   (c) $1^n \mapsto 1^{2^n}$

   (d) $1^n \mapsto \langle n\rangle_2$

   (e) $0^*\langle n\rangle_2 \mapsto 1^n$

   (f) $\langle n\rangle_2 \mapsto \langle n^2\rangle_2$

3. Describe Turing machines that write each of the following infinite streams of bits onto their tape. Specifically, for each integer n, there must be a finite time after which the first n symbols on the tape always match the first n symbols in the target stream.

   (a) An infinite stream of 1s

   (b) 0101101110111101111101111110..., where the nth block of 1s has length n.

   (c) The stream of bits whose nth bit is 1 if and only if n is prime.

(d) The ***Thue-Morse sequence*** $T_0 \bullet T_1 \bullet T_2 \bullet T_3 \cdots$, where

$$T_n := \begin{cases} \texttt{0} & \text{if } n = 0 \\ \texttt{1} & \text{if } n = 1 \\ T_{n-1} \bullet \overline{T_{n-1}} & \text{otherwise} \end{cases}$$

where $\overline{w}$ indicates the binary string obtained from w by flipping every bit. Equivalently, the nth bit of the Thue Morse sequence if `0` if the binary representation of n has an even number of `1`s and `1` otherwise.

`0 1 10 1001 10010110 1001011001101001 10010110011010010110100110010110`...

(e) The ***Fibonacci*** sequence $F_0 \bullet F_1 \bullet F_2 \bullet F_3 \cdots$, where

$$F_n := \begin{cases} \texttt{0} & \text{if } n = 0 \\ \texttt{1} & \text{if } n = 1 \\ F_{n-2} \bullet F_{n-1} & \text{otherwise} \end{cases}$$

`0 1 01 101 01101 10101101 0110110101101 101011010110110101101`...

4. A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

   Prove formally that any standard Turing machine can be simulated by a two-stack machine. That is, given any standard Turing machine M, describe a two-stack machine M' that accepts and rejects exactly the same input strings as M.

★★★

Counter machines. Configuration consists of k rational numbers and an internal state (from some finite set Q). Transition function $\delta: Q \times \{=0, >0\}^k \to Q \times \{-1, 0, +1\}^k$ takes internal state and signs of counters as input, and produces new internal state and changes to counters as output.

- Prove that any Turing machine can be simulated by a three-counter machine. One counter holds the binary representation of the tape after the head; another counter holds the reversed binary representation of the tape before the head. Implement transitions via halving, doubling, and parity, using the third counter for scratch work.
- Prove that two counters can simulate three. Store $2^a 3^b 5^c$ in one counter, use the other for scratch work.
- Prove that a three-counter machine can compute any computable function: Given input $(n, 0, 0)$, we can compute $(f(n), 0, 0)$ for *any* computable function f. First transform $(n, 0, 0)$ to $(2^n, 0, 0)$ using all three counters; then run two- (or three-)counter TM simulation to obtain $(2^{f(n)}, 0, 0)$; and finally transform $(2^{f(n)}, 0, 0)$ to $(f(n), 0, 0)$ using all three counters.
- **HARD:** Prove that a two-counter machine cannot transform $(n, 0)$ to $(2^n, 0)$. [Barzdin' 1963, Yao 1971, Schröpel 1972, Ibarra+Trân 1993]

★★★

FRACTRAN [Conway 1987]: One-counter machine whose "program" is a sequence of rational numbers. The counter is initially 1. At each iteration, multiply the counter by the first rational number that yields an integer; if there is no such number, halt.

- Prove that for any computable function $f : \mathbb{N} \to \mathbb{N}$, there is a FRACTRAN program that transforms 2^{n+1} into $3^{f(n)+1}$, for all natural numbers n.
- Prove that every FRACTRAN program, given the integer 1 as input, either outputs 1 or loops forever. It follows that there is no FRACTRAN program for the increment function $n \mapsto n+1$.

5. A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, describe a tag-Turing machine M' that accepts and rejects exactly the same input strings as M.

6. ⋆(a) Prove that any standard Turing machine can be simulated by a Turing machine with only three states. *[Hint: Use the tape to store an encoding of the state of the machine yours is simulating.]*

   ★(b) Prove that any standard Turing machine can be simulated by a Turing machine with only *two* states.

7. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$, where the arrows indicate which direction the head should move.

   (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

   (b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

   - Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
   - Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
   - Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
   - Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

   Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

8. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current

cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

9. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

   Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem 4!)

10. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

    Show that any tape-stack Turing machine can be simulated by a standard Turing machine.

★★★

- Tape alphabet = $\mathbb{N}$.
  - Read: zero or positive. Write: $+1$, -1
  - Read: even or odd. Write: $+1$, -1, $\times 2$, $\div 2$
  - Read: positive, negative, or zero. Write: $x + y$ (merge), $x - y$ (merge), 1, 0
- Never three times in a row in the same direction
- Hole-punch TM: tape alphabet $\{\square, \blacksquare\}$, and only $\square \mapsto \blacksquare$ transitions allowed.

Caveat lector: This note is not even a first draft, but more of a rough sketch, with many topics still to be written and/or unwritten. But the semester is over, so it's time to put it down. Please send bug reports and suggestions to jeffe@illinois.edu.

Any sufficiently advanced technology is indistinguishable from magic.
— Arthur C. Clarke, "Hazards of Prophecy: The Failure of Imagination" (1962)

Any technology that is distinguishable from magic is insufficiently advanced.
— Barry Gehm, quoted by Stan Schmidt in *ANALOG* magazine (1991)

7 Universal Models of Computation

★★★

Remind about the Church-Turing thesis.

There is some confusion here between **universal models of computation** and the somewhat wider class of **undecidable problems**.

7.1 Universal Turing Machines

The pinnacle of Turing machine constructions is the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine U is a string $\langle M, w\rangle$ that encodes both an arbitrary Turing machine M and a string w in the input alphabet of M. Given these encodings, U simulates the execution of M on input w; in particular,

- U accepts $\langle M, w\rangle$ if and only if M accepts w.
- U rejects $\langle M, w\rangle$ if and only if M rejects w.

In the next few pages, I will sketch a universal Turing machine U that uses the input alphabet $\{\mathtt{0}, \mathtt{1}, \mathtt{[}, \mathtt{]}, \bullet, \mathtt{|}\}$ and a somewhat larger tape alphabet. However, I do *not* require that the Turing machines that U simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red states and tape symbols of U.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|) \rceil$ bits. Thus, if $\Gamma = \{\mathit{0}, \mathit{1}, \mathit{\$}, \mathit{x}, \square\}$, we might use the following encoding:

$$\langle \mathit{0} \rangle = \mathtt{001}, \qquad \langle \mathit{1} \rangle = \mathtt{010}, \qquad \langle \mathit{\$} \rangle = \mathtt{011}, \qquad \langle \mathit{x} \rangle = \mathtt{100}, \qquad \langle \square \rangle = \mathtt{000}.$$

The input string w is encoded by its sequence of symbol encodings, with separators $\bullet$ between every pair of symbols and with brackets $\mathtt{[}$ and $\mathtt{]}$ around the whole string. For example, with this encoding, the input string $\mathit{001100}$ would be encoded on the input tape as

$$\langle \mathit{001100} \rangle = \mathtt{[001\bullet001\bullet010\bullet010\bullet001\bullet001]}$$

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q \rangle$ of $\lceil \lg |Q| \rceil$ bits. Without loss of generality, we encode the start state with all `1`s and the reject state with all `0`s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\langle \mathit{start} \rangle = \mathtt{111} \qquad \langle \mathit{seek1} \rangle = \mathtt{010} \qquad \langle \mathit{seek0} \rangle = \mathtt{011} \qquad \langle \mathit{reset} \rangle = \mathtt{100}$$
$$\langle \mathit{verify} \rangle = \mathtt{101} \qquad \langle \mathit{accept} \rangle = \mathtt{110} \qquad \langle \mathit{reject} \rangle = \mathtt{000}$$

We encode the machine M itself as the string $\langle M \rangle = \mathtt{[}\langle \mathit{reject} \rangle \bullet \langle \square \rangle \mathtt{]} \langle \delta \rangle$, where $\langle \delta \rangle$ is the concatenation of substrings $\mathtt{[}\langle p \rangle \bullet \langle a \rangle \mathtt{|} \langle q \rangle \bullet \langle b \rangle \bullet \langle \Delta \rangle \mathtt{]}$ encoding each transition $\delta(p,a) = (q,b,\Delta)$ such that $q \neq$ reject. We encode the actions $\Delta = \pm 1$ by defining $\langle -1 \rangle := \mathtt{0}$ and $\langle +1 \rangle := \mathtt{1}$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \mathtt{\$}) = (\mathit{start}, \mathtt{\$}, +1)$ would be encoded as

$$\mathtt{[100{\bullet}011|001{\bullet}011{\bullet}1]}.$$

Our first example Turing machine for recognizing $\{\mathtt{0}^n\mathtt{1}^n\mathtt{0}^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
         [010•001|010•001•1][010•100|010•100•1]
         [010•010|011•100•1][011•010|011•010•1]
         [011•100|011•100•1][011•001|100•100•1]
         [100•001|100•001•0][100•010|100•010•0]
         [100•100|100•100•0][100•011|001•011•1]
         [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of M on U's work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $\mathtt{[}\langle q \rangle \bullet \langle a \rangle \mathtt{]}$ indicating that (1) the cell contains symbol a; (2) if $q \neq \mathit{reject}$, then M's head is located at this cell, and M is in state q; and (3) if $q = \mathit{reject}$, then M's head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets `[` and `]`.

For example, with the encodings described above, the initial configuration $(\mathit{start}, \underset{\blacktriangle}{\mathtt{0}}\mathtt{01100}, 0)$ for our first example Turing machine would be encoded on U's tape as follows.

$$\langle \mathit{start}, \underset{\blacktriangle}{\mathtt{0}}\mathtt{01100}, 0 \rangle = \mathtt{[}\underbrace{\mathtt{[111{\bullet}001]}}_{\textbf{start } \mathtt{0}}\underbrace{\mathtt{[000{\bullet}001]}}_{\text{reject } \mathtt{0}}\underbrace{\mathtt{[000{\bullet}010]}}_{\text{reject } \mathtt{1}}\underbrace{\mathtt{[000{\bullet}010]}}_{\text{reject } \mathtt{1}}\underbrace{\mathtt{[000{\bullet}001]}}_{\text{reject } \mathtt{0}}\underbrace{\mathtt{[000{\bullet}001]}}_{\text{reject } \mathtt{0}}\mathtt{]}$$

Similarly, the intermediate configuration $(\mathit{reset}, \mathtt{\$0x}\underset{\blacktriangle}{\mathtt{1}}\mathtt{x0}, 3)$ would be encoded as follows:

$$\langle \mathit{reset}, \mathtt{\$\$x}\underset{\blacktriangle}{\mathtt{1}}\mathtt{x0}, 3 \rangle = \mathtt{[}\underbrace{\mathtt{[000{\bullet}011]}}_{\text{reject } \mathtt{\$}}\underbrace{\mathtt{[000{\bullet}011]}}_{\text{reject } \mathtt{0}}\underbrace{\mathtt{[000{\bullet}100]}}_{\text{reject } \mathtt{x}}\underbrace{\mathtt{[010{\bullet}010]}}_{\textbf{reset } \mathtt{1}}\underbrace{\mathtt{[000{\bullet}100]}}_{\text{reject } \mathtt{x}}\underbrace{\mathtt{[000{\bullet}001]}}_{\text{reject } \mathtt{0}}\mathtt{]}$$

Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine U is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine M followed by an encoding of its input string x. Notice the substrings `[[` and `]]` each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. U also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle \textit{start}, x, 0\rangle$ of the initial configuration of M with input x. First, we write `[[`$\langle \textit{start}\rangle$`•`. Then we copy the encoded input string $\langle x\rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket `[`, write `[[`$\langle \textit{start}\rangle$`•`.
- Instead of copying each separator `•`, write `][`$\langle \textit{reject}\rangle$`•`
- Instead of copying the right bracket `]`, write two right brackets `]]`.

The state encodings $\langle \textit{start}\rangle$ and $\langle \textit{reject}\rangle$ can be copied directly from the beginning of $\langle M\rangle$ (replacing `0`s for `1`s for $\langle \textit{start}\rangle$). Finally, we move the head back to the start of U's tape.

At the start of each step of the simulation, U's head is located at the start of the work tape. We scan through the work tape to the unique encoded cell `[`$\langle p\rangle$`•`$\langle a\rangle$`]` such that $p \neq \textit{reject}$. Then we scan through the encoded transition function $\langle \delta\rangle$ to find the unique encoded tuple `[`$\langle p\rangle$`•`$\langle a\rangle$`|`$\langle q\rangle$`•`$\langle b\rangle$`•`$\langle \Delta\rangle$`]` whose left half matches our the encoded tape cell. If there is no such tuple, then U immediately halts and rejects. Otherwise, we copy the right half $\langle q\rangle$`•`$\langle b\rangle$ of the tuple to the work tape. Now if $q = \textit{accept}$, then U immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \neq \textit{reject}$.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by M's transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets `]]`, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with `[`$\langle \textit{reject}\rangle$`•`$\langle \square\rangle$`]]` (mostly copied from the beginning of the machine encoding $\langle M\rangle$) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks `[`$\langle \textit{reject}\rangle$`•`$\langle \square\rangle$`]` instead of *actual* blanks $\square$.

Example

As an illustrative example, suppose U is simulating our first example Turing machine M on the input string `001100`. The execution of M on input w eventually reaches the configuration (*seek1*, `$$x1x0`, 3). At the start of the corresponding step in U's simulation, U is in the following configuration:

```
[[000•011][000•011][000•100][010•010][000•100][000•001]]
▲
```

First U scans for the first encoded tape cell whose state is not *reject*. That is, U repeatedly compares the first half of each encoded state cell on the work tape with the prefix `[`$\langle \textit{reject}\rangle$`•` of the machine encoding $\langle M\rangle$ on the input tape. U finds a match in the fourth encoded cell.

```
[[000•011][000•011][000•100][010•010][000•100][000•001]]
                                ▲
```

Next, U scans the machine encoding $\langle M\rangle$ for the substring `[010•010` matching the current encoded cell. U eventually finds a match in the left size of the the encoded transition `[010•010|011•100•1]`. U copies the state-symbol pair `011•100` from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

$$\texttt{[[000•011][000•011][000•100][}\underline{\texttt{011•100}}\texttt{][000•100][000•001]]}$$

The encoded transition instructs U to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

$$\texttt{[[000•011][000•011][000•100][}\underline{\texttt{000}}\texttt{•100][}\underline{\texttt{011}}\texttt{•100][000•001]]}$$

Finally, U scans left until it reads two left brackets `[[`; this returns the head to the left end of the work tape to start the next step in the simulation. U's tape now holds the encoding of M's configuration (*seek0*, *$$xxx0*, 4), as required.

`[[000•011][000•011][000•100][000•100][011•100][000•001]]`

7.2 Two-Stack Machines

A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

★★★

Simulate a doubly-infinite tape with two stacks, one holding the tape contents to the left of the head, the other holding the tape contents to the right of the head. For each transition of a standard Turing machine M, the stack machine pops the top symbol off the (say) left stack, changes its internal state according to the transition δ, and then either pushes a new symbol onto the right stack, or pushes a new symbol onto the left stack and then moves the top symbol from the right stack to the left stack.

7.3 Counter Machines

★★★

A configuration of a k-counter machine consists of k non-negative integers and an internal state from some finite set Q. The transition function $\delta : Q \times \{0, +1\}^k \to Q \times \{-1, 0, +1\}^k$ takes an internal state and the signs of the counters as input, and produces a new internal state and changes to counters as output.

- Prove that any Turing machine can be simulated by a three-counter machine. One counter holds the binary representation of the tape after the head; another counter holds the reversed binary representation of the tape before the head. Implement transitions via halving, doubling, and parity, using the third counter for scratch work.
- Prove that two counters can simulate three. Store $2^a 3^b 5^c$ in one counter, use the other for scratch work.
- Prove that a three-counter machine can compute any computable function: Given input $(n, 0, 0)$, we can compute $(f(n), 0, 0)$ for *any* computable function f. First transform $(n, 0, 0)$ to $(2^n, 0, 0)$ using all three counters; then run two- (or three-)counter TM simulation to obtain $(2^{f(n)}, 0, 0)$; and finally transform $(2^{f(n)}, 0, 0)$ to $(f(n), 0, 0)$ using all three counters.
- **HARD:** Prove that a two-counter machine cannot transform $(n, 0)$ to $(2^n, 0)$. [Barzhdin 1963, Yao 1971, Schröpel 1972]

7.4 FRACTRAN

★★★

FRACTRAN [Conway 1987]: A one-counter machine whose "program" is a sequence of rational numbers. The counter is initially 1. At each iteration, multiply the counter by the first rational number that yields an integer; if there is no such number, halt.

- Prove that for any computable function $f : \mathbb{N} \to \mathbb{N}$, there is a FRACTRAN program that transforms 2^{n+1} into $3^{f(n)+1}$, for all natural numbers n.
- Prove that every FRACTRAN program, given the integer 1 as input, either outputs 1 or loops forever. It follows that there is no FRACTRAN program for the increment function $n \mapsto n+1$.

7.5 Post Correspondence Problem

Given n of pairs of strings $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, is there a finite sequence of integers $(i_1, i_2, \ldots, i_k)$ such that $x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_{i_k}$? For notation convenience, we write each pair vertically as $\begin{bmatrix} x \\ y \end{bmatrix}$ instead of horizontally as (x, y). For example, given the string pairs

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{100} \end{bmatrix}, \; b = \begin{bmatrix} \texttt{01} \\ \texttt{00} \end{bmatrix}, \; c = \begin{bmatrix} \texttt{110} \\ \texttt{11} \end{bmatrix},$$

we should answer TRUE, because

$$cbca = \begin{bmatrix} \texttt{110} \\ \texttt{11} \end{bmatrix} \begin{bmatrix} \texttt{01} \\ \texttt{00} \end{bmatrix} \begin{bmatrix} \texttt{110} \\ \texttt{11} \end{bmatrix} \begin{bmatrix} \texttt{0} \\ \texttt{100} \end{bmatrix}$$

gives us `110110100` for both concatenations. As more extreme examples, the shortest solutions for the input

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{001} \end{bmatrix}, \; b = \begin{bmatrix} \texttt{001} \\ \texttt{1} \end{bmatrix}, \; c = \begin{bmatrix} \texttt{1} \\ \texttt{0} \end{bmatrix}$$

have length 75; one such solution is $aacaacabbabccaaccaaaacbaabbaacbacbbccbbacbaccbcb$ $acbbacbaccbacbbbacccbabbccbaacaacaaacbabbaacacbccbbabacbcaaccbacabbbbabcccc$ $bcaababaaccbcbbbacccbabbccb$. The shortest solution for the instance

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{000} \end{bmatrix}, \; b = \begin{bmatrix} \texttt{0} \\ \texttt{0101} \end{bmatrix}, \; c = \begin{bmatrix} \texttt{01} \\ \texttt{1} \end{bmatrix}, \; d = \begin{bmatrix} \texttt{1111} \\ \texttt{10} \end{bmatrix}$$

is the unbelievable $a^2 b^8 a^4 c^{16} a b^4 a^2 b^4 a d^4 b^3 c^8 a^6 c^8 b^2 c^4 b c^6 d^2 a^{18} d^2 c^4 d c a d^2 c b^{54} c^3 d c a^2 c^{111} d c$ $a^6 d^{28} c b^{17} c^{63} d^{16} c^{16} d^4 c^4 d c$, which has total length 451. Finally, the shortest solution for the instance

$$a = \begin{bmatrix} \texttt{0} \\ \texttt{00010} \end{bmatrix}, \; b = \begin{bmatrix} \texttt{010} \\ \texttt{01} \end{bmatrix}, \; c = \begin{bmatrix} \texttt{100} \\ \texttt{0} \end{bmatrix},$$

has length 528.

★★★

The simplest universality proof simulates a tag-Turing machine.

7.6 Matrix Mortality

★★★

Given a set of integer matrices $A_1, \ldots, A_k$, is the product of any sequence of these matrices (with repetition) equal to 0? Undecidable by reduction from PCP, even for two 15×15 matrices or six 3×3 matrices [Cassaigne, Halava, Harju, Nicolas 2014]

7.7 Dynamical Systems

★★★

Ray Tracing [Reif, Tygar, and Yoshida 1994] The configuration of a Turing machine is encoded as the (x, y) coordinates of a light path crossing the unit square $[0,1] \times [0,1]$, where the x-(resp. y-)coordinate encodes the tape contents to the left (resp. right) of the head. Need either quadratic-surface mirrors or refraction to simulate transitions.

N-body problem [Smith 2006]: Similar idea

Skolem-Pisot reachability: Given an integer vector x and an integer matrix A, does $A^n x = (0, \dots)$ for any integer n? [Halava, Harju, Hirvensalo, Karhumäki 2005] It's surprising that this problem is undecidable; the similar mortality problem for one matrix is not.

7.8 Wang Tiles

★★★

Turing machine simulation is straightforward. *Small* Turing-complete tile sets via affine maps (via two-stack machines) are a little harder.

7.9 Combinator Calculus

In the 1920s, Moses Schönfinkel developed what can now be interpreted as a model of computation now called *combinator calculus* or *combinatory logic*. Combinator calculus operates on ***terms***, where every term is either one of a finite number of ***combinators*** (represented here by upper case letters) or an ordered pair of terms. For notational convenience, we omit commas between components of every pair and parentheses around the *left* term in every pair. Thus, $\mathtt{SKK(IS)}$ is shorthand for the term $(((\mathtt{S}, \mathtt{K}), \mathtt{K}), (\mathtt{I}, \mathtt{S}))$.

We can "evaluate" any term by a sequence of rewriting rules that depend on its first primitive combinator. Schönfinkel defined three primitive combinators with the following evaluation rules:

- Identity: $\mathtt{I}x \mapsto x$
- Constant: $\mathtt{K}xy \mapsto x$
- Substitution: $\mathtt{S}xyz \mapsto xz(yz)$

Here, x, y, and z are variables representing unknown but arbitrary terms. "Computation" in the combinator calculus is performed by repeatedly evaluating arbitrary (sub)terms with one of these three structures, until all such (sub)terms are gone.

For example, the term $\mathtt{S(K(SI))K}xy$ (for any terms x and y) evaluates as follows:

$$
\begin{aligned}
\underline{\mathtt{S(K(SI))K}xy} &\mapsto \underline{\mathtt{K(SI)}x}(\mathtt{K}x)y && \text{Substitution}\\
&\mapsto \underline{\mathtt{SI}(\mathtt{K}x)y} && \text{Constant}\\
&\mapsto \underline{\mathtt{I}y}(\mathtt{K}xy) && \text{Substitution}\\
&\mapsto y(\underline{\mathtt{K}xy}) && \text{Identity}\\
&\mapsto yx && \text{Constant}
\end{aligned}
$$

Thus, we can define a new combinator $\mathtt{R} := \mathtt{S(K(SI))K}$ that upon evaluation reverses the next two terms: $\mathtt{R}xy \mapsto yx$.

On the other hand, evaluating $\mathtt{SII(S(KI)(SII))}$ leads to an infinite loop:

$$\begin{aligned}
\underline{\mathtt{SII(S(KI)(SII))}} &\mapsto \underline{\mathtt{I(S(KI)(SII))}}\mathtt{(I(S(KI)(SII)))} && \text{Substitution}\\
&\mapsto \mathtt{S(KI)(SII)(}\underline{\mathtt{I(S(KI)(SII))}}\mathtt{)} && \text{Identity}\\
&\mapsto \underline{\mathtt{S(KI)(SII)(S(KI)(SII))}} && \text{Identity}\\
&\mapsto \underline{\mathtt{KI(S(KI)(SII))}}\mathtt{(SII(S(KI)(SII)))} && \text{Substitution}\\
&\mapsto \underline{\mathtt{I(SII(S(KI)(SII)))}} && \text{Constant}\\
&\mapsto \mathtt{SII(S(KI)(SII))} && \text{Identity}
\end{aligned}$$

Wikipedia sketches a direct *undecidability* proof. Is there a Turing-completeness proof that avoids λ-calculus?

Exercises

1. A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, describe a tag-Turing machine M' that accepts and rejects exactly the same input strings as M.

2. *(a) Prove that any standard Turing machine can be simulated by a Turing machine with only three states. *[Hint: Use the tape to store an encoding of the state of the machine yours is simulating.]*

   $^\bigstar$(b) Prove that any standard Turing machine can be simulated by a Turing machine with only *two* states.

3. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta: Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$, where the arrows indicate which direction the head should move.

   (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

   (b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

   - Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
   - Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
   - Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
   - Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

4. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

   Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

5. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

   Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem **??**!)

6. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

   Show that any tape-stack Turing machine can be simulated by a standard Turing machine.

★★★

- Tape alphabet = $\mathbb{N}$.
  - Read: zero or positive. Write: $+1$, -1
  - Read: even or odd. Write: $+1$, -1, $\times 2$, $\div 2$
  - Read: positive, negative, or zero. Write: $x + y$ (merge), $x - y$ (merge), 1, 0
- Never three times in a row in the same direction
- Hole-punch TM: tape alphabet $\{\square, \blacksquare\}$, and only $\square \mapsto \blacksquare$ transitions allowed.

Caveat lector: This is the zeroth (draft) edition of this lecture note. Please send bug reports and suggestions to jeffe@illinois.edu.

I said in my haste, All men are liars.

— Psalms 116:11 (King James Version)

yields falsehood when preceded by its quotation.

— William V. Quine, "Paradox", *Scientific American* (1962)

Some problems are so complex that you have to be highly intelligent and well informed just to be undecided about them.

— Laurence Johnston Peter, *Peter's Almanac* (September 24, 1982)

"Proving or disproving a formula—once you've encrypted the formula into numbers, that is—is just a calculation on that number. So it means that the answer to the question is, no! Some formulas cannot be proved or disproved by any mechanical process! So I guess there's some point in being human after all!"

Alan looked pleased until Lawrence said this last thing, and then his face collapsed. "Now there you go making unwarranted assumptions."

— Neal Stephenson, *Cryptonomicon* (1999)

No matter how P might perform, Q will scoop it:
Q uses P's output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it's wrong, and is false when it's true!

— Geoffrey S. Pullum, "Scooping the Loop Sniffer" (2000)

This castle is in unacceptable condition! ***UNACCEPTABLE!!***

— Earl of Lemongrab [Justin Roiland], "Too Young"
Adventure Time (August 8, 2011)

8 Undecidability

Perhaps the single most important result in Turing's remarkable 1936 paper is his solution to Hilbert's *Entscheidungsproblem*, which asked for a general automatic procedure to determine whether a given statement of first-order logic is *provable*. Turing proved that no such procedure exists; there is no systematic way to distinguish between statements that cannot be proved even in principle and statements whose proofs we just haven't found yet.

8.1 Acceptable versus Decidable

Recall that there are three possible outcomes for a Turing machine M running on any particular input string w: acceptance, rejection, and divergence. Every Turing machine M immediately defines four different languages (over the input alphabet Σ of M):

- The *accepting* language $\textsc{Accept}(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$
- The *rejecting* language $\textsc{Reject}(M) := \{w \in \Sigma^* \mid M \text{ rejects } w\}$
- The *halting* language $\textsc{Halt}(M) := \textsc{Accept}(M) \cup \textsc{Reject}(M)$
- The *diverging* language $\textsc{Diverge}(M) := \Sigma^* \setminus \textsc{Halt}(M)$

For any language L, the sentence "***M accepts L***" means $\textsc{Accept}(M) = L$, and the sentence "***M decides L***" means $\textsc{Accept}(M) = L$ and $\textsc{Diverge}(M) = \varnothing$.

Now let L be an arbitrary language. We say that L is ***acceptable*** (or *semi-computable*, or *semi-decidable*, or *recognizable*, or *listable*, or *recursively enumerable*) if some Turing machine accepts L, and ***unacceptable*** otherwise. Similarly, L is ***decidable*** (or *computable*, or *recursive*) if some Turing machine decides L, and ***undecidable*** otherwise.

8.2 Lo, I Have Become Death, Stealer of Pie

There is a subtlety in the definitions of "acceptable" and "decidable" that many beginners miss: A language can be decidable even if we can't exhibit a specific Turing machine decides it. As a canonical example, consider the language $\Pi = \{w \mid \mathtt{1}^{|w|} \text{ appears in the binary expansion of } \pi\}$. Despite appearances, this language *is* decidable! There are only two cases to consider:

- Suppose there is an integer N such that the binary expansion of π contains the substring $\mathtt{1}^N$ but does not contain the substring $\mathtt{1}^{N+1}$. Let M_N be the Turing machine with $N+3$ states $\{0, 1, \dots, N, \mathsf{accept}, \mathsf{reject}\}$, start state 0, and the following transition function:

$$\delta(q, a) = \begin{cases} \mathsf{accept} & \text{if } a = \square \\ \mathsf{reject} & \text{if } a \neq \square \text{ and } q = n \\ (q+1, a, +1) & \text{otherwise} \end{cases}$$

  This machine correctly decides Π.

- Suppose the binary expansion of π contains arbitrarily long substrings of $\mathtt{1}$s. Then any Turing machine that accepts all inputs correctly decides Π.

We have no idea which of these machines correctly decides Π, but one of them does, and that's enough!

8.3 Useful Lemmas

This subsection lists several simple but useful properties of (un)decidable and (un)acceptable languages. For almost all of these properties, the proofs are straightforward; readers are strongly encouraged to try to prove each lemma themselves before reading ahead.

One might reasonably ask why we don't also define "rejectable" and "haltable" languages. The following lemma, whose proof is an easy exercise (hint, hint), implies that these are both identical to the acceptable languages.

Lemma 1. *Let M be an arbitrary Turing machine.*
(a) There is a Turing machine M^R such that $\textsc{Accept}(M^R) = \textsc{Reject}(M)$ and $\textsc{Reject}(M^R) = \textsc{Accept}(M)$.
(b) There is a Turing machine M^A such that $\textsc{Accept}(M^A) = \textsc{Accept}(M)$ and $\textsc{Reject}(M^A) = \varnothing$.
(c) There is a Turing machine M^H such that $\textsc{Accept}(M^H) = \textsc{Halt}(M)$ and $\textsc{Reject}(M^H) = \varnothing$.

The decidable languages have several fairly obvious useful properties.

Lemma 2. *If L and L' are decidable, then $L \cup L'$, $L \cap L'$, $L \setminus L'$, and $L' \setminus L$ are also decidable.*

Proof: Let M and M' be Turing machines that decide L and L', respectively. We can build a Turing machine $M_\cup$ that decides $L \cup L'$ as follows. First, $M_\cup$ copies its input string w onto a second tape. Then $M_\cup$ runs M on input w (on the first tape), and then runs M' on input w (on the second tape). If either M or M' accepts, then $M_\cup$ accepts; if both M and M' reject, then $M_\cup$ rejects.

The other three languages are similar. □

Corollary 3. *The following hold for all languages L and L'.*
(a) If $L \cap L'$ is undecidable and L' is decidable, then L is undecidable.
(b) If $L \cup L'$ is undecidable and L' is decidable, then L is undecidable.
(c) If $L \setminus L'$ is undecidable and L' is decidable, then L is undecidable.
(d) If $L' \setminus L$ is undecidable and L' is decidable, then L is undecidable.

The asymmetry between acceptance and rejection implies that merely acceptable languages are not quite as well-behaved as decidable languages.

Lemma 4. *For all acceptable languages L and L', the languages $L \cup L'$ and $L \cap L'$ are also acceptable.*

Proof: Let M and M' be Turing machines that decide L and L', respectively. We can build a Turing machine $M_\cap$ that decides $L \cap L'$ as follows. First, $M_\cap$ copies its input string w onto a second tape. Then $M_\cap$ runs M on input w using the first tape, and then runs M' on input w using the second tape. If both M and M' accept, then $M_\cap$ accepts; if either M or M' reject, then $M_\cap$ rejects; if either M or M' diverge, then $M_\cap$ diverges (automatically).

The construction for $L \cup L'$ is more subtle; instead of running M and M' in series, we must run them in parallel. Like $M_\cap$, the new machine $M_\cup$ starts by copying its input string w onto a second tape. But then $M_\cup$ runs M and M' simultaneously; with each step of $M_\cup$ simulating both one step of M on the first tape and one step of M' on the second. Ignoring the states and transitions needed for initialization, the state set of $M_\cup$ is the product of the state sets of M and M', and the transition function is

$$\delta_\cup(q,a,q',a') = \begin{cases} \mathsf{accept}_\cup & \text{if } q = \mathsf{accept} \text{ or } q' = \mathsf{accept}' \\ \mathsf{reject}_\cup & \text{if } q = \mathsf{reject} \text{ and } q' = \mathsf{reject}' \\ (\delta(q,a), \delta'(q',a')) & \text{otherwise} \end{cases}$$

Thus, $M_\cup$ accepts as soon as either M or M' accepts, and rejects only after both M or M' reject. □

Lemma 5. *An acceptable language L is decidable if and only if $\Sigma^* \setminus L$ is also acceptable.*

Proof: Let M and $\overline{M}$ be Turing machines that accept L and $\Sigma^* \setminus L$, respectively. Following the previous proof, we construct a new Turing machine M^* that copies its input onto a second tape, and then simulates M and M' in parallel on the two tapes. If M accepts, then M^* accepts; if $\overline{M}$ accepts, then M^* rejects. Since every string is accepted by either M or $\overline{M}$, we conclude that M^* decides L.

The other direction follows immediately from Lemma 1. □

8.4 Self-Haters Gonna Self-Hate

Let U be an arbitrary fixed universal Turing machine. Any Turing machine M can be encoded as a string $\langle M\rangle$ of symbols from U's input alphabet, so that U can simulate the execution of M on any suitably encoded input string. Different universal Turing machines require different encodings.[1]

A Turing machine encoding is just a string, and any string (over the correct alphabet) can be used as the input to a Turing machine. Thus, we can use the encoding $\langle M\rangle$ of any Turing machine M as the input to another Turing machine. We've already seen an example of this ability in our universal Turing machine U, but more significantly, we can use $\langle M\rangle$ as the input to *the same Turing machine M*. Thus, each of the following languages is well-defined:

$$\begin{aligned}
\textsc{SelfAccept} &:= \Big\{\langle M\rangle \;\Big|\; M \text{ accepts } \langle M\rangle\Big\}\\
\textsc{SelfReject} &:= \Big\{\langle M\rangle \;\Big|\; M \text{ rejects } \langle M\rangle\Big\}\\
\textsc{SelfHalt} &:= \Big\{\langle M\rangle \;\Big|\; M \text{ halts on } \langle M\rangle\Big\}\\
\textsc{SelfDiverge} &:= \Big\{\langle M\rangle \;\Big|\; M \text{ diverges on } \langle M\rangle\Big\}
\end{aligned}$$

One of Turing's key observations is that ***SelfReject* is undecidable**; Turing proved this theorem by contradiction as follows:

Suppose to the contrary that there is a Turing machine SR such that $\textsc{Accept}(SR) = \textsc{SelfReject}$ and $\textsc{Diverge}(SR) = \varnothing$. More explicitly, for ***any*** Turing machine M,

- SR accepts $\langle M\rangle \iff M$ rejects $\langle M\rangle$, and
- SR rejects $\langle M\rangle \iff M$ does not reject $\langle M\rangle$.

In particular, these equivalences must hold when M is equal to SR. Thus,

- SR accepts $\langle SR\rangle \iff SR$ rejects $\langle SR\rangle$, and
- SR rejects $\langle SR\rangle \iff SR$ does not reject $\langle SR\rangle$.

In short, SR accepts $\langle SR\rangle$ if and only if SR rejects $\langle SR\rangle$, which is impossible! The only logical conclusion is that the Turing machine SR does not exist!

8.5 Aside: Uncountable Barbers

Turing's proof by contradiction is nearly identical to the famous ***diagonalization argument*** that uncountable sets exist, published by Georg Cantor in 1891. Indeed, $\textsc{SelfReject}$ is sometimes called "the diagonal language". Recall that a function $f : A \to B$ is a ***surjection***[2] if $f(A) = \{f(a) \mid a \in A\} = B$.

Cantor's Theorem. *Let $f : X \to 2^X$ be an **arbitrary** function from an **arbitrary** set X to its power set. This function f is not a surjection.*

[1]In fact, these undecidability proofs never actually use the universal Turing machine; all we really need is an encoding function that associates a unique string $\langle M\rangle$ with every Turing machine M. However, we *do* need the encoding to be compatible with a universal Turing machine for the results in Section **??**.

[2]more commonly, flouting all reasonable standards of grammatical English, "an onto function"

Proof: Fix an arbitrary function $f : X \to 2^X$. Call an element $x \in X$ ***happy*** if $x \in f(x)$ and ***sad*** if $x \notin f(x)$. Let Y be the set of all sad elements of X; that is, for *every* element $x \in X$, we have

$$x \in Y \iff x \notin f(x).$$

For the sake of argument, suppose f is a surjection. Then (by definition of surjection) there must be an element $y \in X$ such that $f(y) = Y$. Then for *every* element $x \in X$, we have

$$x \in f(y) \iff x \notin f(x).$$

In particular, the previous equivalence must hold when $x = y$:

$$y \in f(y) \iff y \notin f(y).$$

We have a contradiction! We conclude that f is not a surjection after all. □

Now let $X = \Sigma^*$, and define the function $f : X \to 2^X$ as follows:

$$f(w) := \begin{cases} \textsc{Accept}(M) & \text{if } w = \langle M \rangle \text{ for some Turing machine } M \\ \varnothing & \text{if } w \text{ is not the encoding of a Turing machine} \end{cases}$$

Cantor's theorem immediately implies that not all languages are acceptable.

Alternatively, let X be the set of all Turing machines that halt on all inputs. For any Turing machine $M \in X$, let $f(M)$ be the set of all Turing machines $N \in X$ such that M accepts the encoding $\langle N \rangle$. Then a Turing machine M is *sad* if it rejects its own encoding $\langle M \rangle$; thus, Y is essentially the set SelfReject. Cantor's argument now immediately implies that no Turing machine decides the language SelfReject.

The core of Cantor's diagonalization argument also appears in the "barber paradox" popularized by Bertrand Russell in the 1910s. In a certain small town, every resident has a haircut on Haircut Day. Some residents cut their own hair; others have their hair cut by another resident of the same town. To obtain an official barber's license, a resident must cut the hair of all residents who don't cut their own hair, and no one else. Given these assumptions, we can immediately conclude that there are no licensed barbers. After all, who would cut the barber's hair?

To map Russell's barber paradox back to Cantor's theorem, let X be the set of residents, and let $f(x)$ be the set of residents who have their hair cut by x; then a resident is *sad* if they do not cut their own hair. To prove that SelfReject is undecidable, replace "resident" with "a Turing machine that halts on all inputs", and replace "A cuts B's hair" with "A accepts $\langle B \rangle$".

8.6 Just Don't Know What to Do with Myself

Similar diagonal arguments imply that the other three languages SelfAccept, SelfHalt, and Self-Diverge are also undecidable. The proofs are not quite as direct for these three languages as the proof for SelfReject; each fictional deciding machine requires a small modification to create the contradiction.

Theorem 6. *SelfAccept is undecidable.*

Proof: For the sake of argument, suppose there is a Turing machine SA such that $\textsc{Accept}(SA) = \textsc{SelfAccept}$ and $\textsc{Diverge}(M) = \varnothing$. Let SA^R be the Turing machine obtained from SA by swapping its accept and reject states (as in the proof of Lemma 1). Then $\textsc{Reject}(SA^R) = \textsc{SelfAccept}$ and $\textsc{Diverge}(SA^R) = \varnothing$. It follows that SA^R rejects $\langle SA^R \rangle$ if and only if SA^R accepts $\langle SA^R \rangle$, which is impossible. □

Theorem 7. *SelfHalt is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine SH such that $\textsc{Accept}(SH) = \textsc{SelfHalt}$ and $\textsc{Diverge}(SH) = \varnothing$. Let SH^X be the Turing machine obtained from SH by redirecting every transition to accept to a new hanging state hang, and then redirecting every transition to reject to accept. Then $\textsc{Accept}(SH^X) = \Sigma^* \setminus \textsc{SelfHalt}$ and $\textsc{Reject}(SH^X) = \varnothing$. It follows that SH^X accepts $\langle SH^X \rangle$ if and only if SH^X does not halt on $\langle SH^X \rangle$, and we have a contradiction. □

Theorem 8. *SelfDiverge is **unacceptable and therefore** undecidable.*

Proof: Suppose to the contrary that there is a Turing machine SD such that $\textsc{Accept}(M) = \textsc{SelfDiverge}$. Let SD^A be the Turing machine obtained from M by redirecting every transition to reject to a new hanging state hang such that $\delta(\text{hang}, a) = (\text{hang}, a, +1)$ for every symbol a. Then $\textsc{Accept}(SD^A) = \textsc{SelfDiverge}$ and $\textsc{Reject}(SD^A) = \varnothing$. It follows that SD^A accepts $\langle SD^A \rangle$ if and only if SD^A does not halt on $\langle SD^A \rangle$, which is impossible. □

*8.7 Nevertheless, Acceptable

Our undecidability argument for SelfDiverge actually implies the stronger result that SelfDiverge is unacceptable; we never assumed that the hypothetical accepting machine SD halts on all inputs. However, we can use or modify our universal Turing machine to accept the other three languages.

Theorem 9. *SelfAccept is acceptable.*

Proof: We describe a Turing machine SA that accepts the language SelfAccept. Given any string w as input, SA first verifies that w is the encoding of a Turing machine. If w is not the encoding of a Turing machine, then SA diverges. Otherwise, $w = \langle M \rangle$ for some Turing machine M; in this case, SA writes the string $ww = \langle M \rangle\langle M \rangle$ onto its tape and passes control to the universal Turing machine U. U then simulates M (the machine encoded by the first half of its input) on the string $\langle M \rangle$ (the second half of its input).[3] In particular, U accepts $\langle M, M \rangle$ if and only if M accepts $\langle M \rangle$. We conclude that SR accepts $\langle M \rangle$ if and only if M accepts $\langle M \rangle$. □

Theorem 10. *SelfReject is acceptable.*

Proof: Let U^R be the Turing machine obtained from our universal machine U by swapping the accept and reject states. We describe a Turing machine SR that accepts the language SelfReject as follows. SR first verifies that its input string w is the encoding of a Turing machine and diverges if not. Otherwise, SR writes the string $ww = \langle M, M \rangle$ onto its tape and passes control to the reversed universal Turing machine U^R. Then U^R accepts $\langle M, M \rangle$ if and only if M rejects $\langle M \rangle$. We conclude that SR accepts $\langle M \rangle$ if and only if M rejects $\langle M \rangle$. □

Finally, because SelfHalt is the union of two acceptable languages, SelfHalt is also acceptable.

[3]To simplify the presentation, I am implicitly assuming here that $\langle M \rangle = \langle\langle M \rangle\rangle$. Without this assumption, we need a Turing machine that transforms an arbitrary string $w \in \Sigma_M^*$ into its encoding $\langle w \rangle$ for U; building such a Turing machine is straightforward.

8.8 The Halting Problem via Reduction

Consider the following related languages:[4]

$$\begin{aligned}
\text{ACCEPT} &:= \{\langle M, w\rangle \mid M \text{ accepts } w\}\\
\text{REJECT} &:= \{\langle M, w\rangle \mid M \text{ rejects } w\}\\
\text{HALT} &:= \{\langle M, w\rangle \mid M \text{ halts on } w\}\\
\text{DIVERGE} &:= \{\langle M, w\rangle \mid M \text{ diverges on } w\}
\end{aligned}$$

Deciding the language HALT is what is usually meant by the *halting problem*: Given a program M and an input w to that program, does the program halt? This problem may seem trivial; why not just run the program and see? More formally, why not just pass the input string $\langle M, x\rangle$ to our universal Turing machine U? That strategy works perfectly if we just want to *accept* HALT, but we actually want to *decide* HALT; if M is not going to halt on w, we still want an answer in a finite amount of time. Sadly, we can't always get what we want.

Theorem 11. *HALT is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine H that decides HALT. Then we can use H to build another Turing machine SH that decides the language SELFHALT. Given any string w, the machine SH first verifies that $w = \langle M\rangle$ for some Turing machine M (rejecting if not), then writes the string $ww = \langle M, M\rangle$ onto the tape, and finally passes control to H. But SELFHALT is undecidable, so no such machine SH exists. We conclude that H does not exist either. □

Nearly identical arguments imply that the languages ACCEPT, REJECT, and DIVERGE are undecidable.

Here we have our first example of an undecidability proof by ***reduction***. Specifically, we ***reduced*** the language SELFHALT to the language HALT. More generally, to reduce one language X to another language Y, we assume (for the sake of argument) that there is a program P_Y that decides Y, and we write another program that decides X, using P_Y as a black-box subroutine. If later we discover that Y is decidable, we can immediately conclude that X is decidable. Equivalently, if we later discover that X is undecidable, we can immediately conclude that Y is undecidable.

To prove that a language L is undecidable,
reduce a known undecidable language to L.

Perhaps the most confusing aspect of reduction arguments is that the *languages* we want to prove undecidable nearly (but not quite) always involve encodings of Turing machines, while at the same time, the *programs* that we build to prove them undecidable are also Turing machines. Our proof that HALT is undecidable involved three different machines:

- The hypothetical Turing machine H that decides HALT.
- The new Turing machine SH that decides SELFHALT, using H as a subroutine.

[4]Sipser uses the shorter name A_{TM} instead of ACCEPT, but uses $HALT_{TM}$ instead of HALT. I have no idea why he thought four-letter names are okay, but six-letter names are not. His subscript TM is just a reminder that these are languages of *Turing machine* encodings, as opposed to encodings of DFAs or some other machine model.

- The Turing machine M whose encoding is the input to H.

It is *incredibly* easy to get confused about which machines are playing each in the proof. Therefore, it is absolutely vital that we give each machine in a reduction proof a unique and mnemonic name, and then ***always*** refer to each machine ***by name***. Never write, say, or even *think* "the machine" or "that machine" or (gods forbid) "it". You also may find it useful to think of the working ***programs*** we are trying to construct (H and SH in this proof) as being written in a different language than the arbitrary ***source code*** that we want those programs to analyze ($\langle M \rangle$ in this proof).

8.9 One Million Years Dungeon!

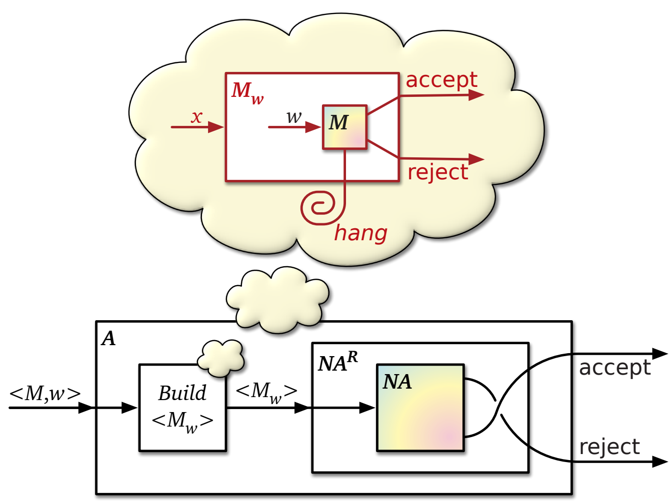
As a more complex set of examples, consider the following languages:

$$\begin{aligned}
\text{NeverAccept} &:= \{\langle M \rangle \mid \text{Accept}(M) = \varnothing\} \\
\text{NeverReject} &:= \{\langle M \rangle \mid \text{Reject}(M) = \varnothing\} \\
\text{NeverHalt} &:= \{\langle M \rangle \mid \text{Halt}(M) = \varnothing\} \\
\text{NeverDiverge} &:= \{\langle M \rangle \mid \text{Diverge}(M) = \varnothing\}
\end{aligned}$$

Theorem 12. *NeverAccept is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine NA that decides NeverAccept. Then by swapping the accept and reject states, we obtain a Turing machine NA^R that decides the complementary language $\Sigma^* \setminus$ NeverAccept.

To reach a contradiction, we construct a Turing machine A that decides Accept as follows. Given the encoding $\langle M, w \rangle$ of an arbitrary machine M and an arbitrary string w as input, A writes the encoding $\langle M_w \rangle$ of a new Turing machine M_w that ignores its input, writes w onto the tape, and then passes control to M. Finally, A passes the new encoding $\langle M_w \rangle$ as input to NA^R. The following cartoon tries to illustrate the overall construction.



A reduction from from Accept to NeverAccept, which proves NeverAccept undecidable.

Before going any further, it may be helpful to list the various Turing machines that appear in this construction.

- The hypothetical Turing machine NA that decides NeverAccept.
- The Turing machine NA^R that decides $\Sigma^* \setminus$ NeverAccept, which we constructed by modifying NA.
- The Turing machine A that we are building, which decides Accept using NA^R as a black-box subroutine.
- The Turing machine M, whose encoding is part of the input to A.
- The Turing machine M_w whose encoding A constructs from $\langle M, w\rangle$ and then passes to NA^R as input.

Now let M be an arbitrary Turing machine and w be an arbitrary string, and suppose we run our new Turing machine A on the encoding $\langle M, w\rangle$. To complete the proof, we need to consider two cases: Either M accepts w or M does not accept w.

- First, suppose M accepts w.
  - Then for all strings x, the machine M_w accepts x.
  - So Accept$(M_w) = \Sigma^*$, by the definition of Accept(M_w).
  - So $\langle M_w\rangle \notin$ NeverAccept, by definition of NeverAccept.
  - So NA rejects $\langle M_w\rangle$, because NA decides NeverAccept.
  - So NA^R accepts $\langle M_w\rangle$, buy construction of NA^R.
  - We conclude that A accepts $\langle M, w\rangle$, by construction of A.
- On the other hand, suppose M does not accept w, either rejecting or diverging instead.
  - Then for all strings x, the machine M_w does not accept x.
  - So Accept$(M_w) = \varnothing$, by the definition of Accept(M_w).
  - So $\langle M_w\rangle \in$ NeverAccept, by definition of NeverAccept.
  - So NA accepts $\langle M_w\rangle$, because NA decides NeverAccept.
  - So NA^R rejects $\langle M_w\rangle$, buy construction of NA^R.
  - We conclude that A rejects $\langle M, w\rangle$, by construction of A.

In short, A decides the language Accept, which is impossible. We conclude that NA does not exist. □

Again, similar arguments imply that the languages NeverReject, NeverHalt, and Never-Diverge are undecidable. In each case, the core of the argument is describing how to transform the incoming machine-and-input encoding $\langle M, w\rangle$ into the encoding of an appropriate new Turing machine $\langle M_w\rangle$.

Now that we know that NeverAccept and its relatives are undecidable, we can use them as the basis of further reduction proofs. Here is a typical example:

Theorem 13. *The language DivergeSame* $:= \left\{ \langle M_1\rangle \langle M_2\rangle \;\middle|\; \textit{Diverge}(M_1) = \textit{Diverge}(M_2) \right\}$ *is undecidable.*

Proof: Suppose for the sake of argument that there is a Turing machine DS that decides DivergeSame. Then we can build a Turing machine ND that decides NeverDiverge as follows. Fix a Turing machine Y that accepts Σ^* (for example, by defining $\delta(\text{start}, a) = (\text{accept}, \cdot, \cdot)$ for all $a \in \Gamma$). Given an arbitrary Turing machine encoding $\langle M\rangle$ as input, ND writes the string $\langle M\rangle\langle Y\rangle$ onto the tape and then passes control to DS. There are two cases to consider:

- If DS accepts $\langle M\rangle\langle Y\rangle$, then $\textsc{Diverge}(M) = \textsc{Diverge}(Y) = \varnothing$, so $\langle M\rangle \in \textsc{NeverDiverge}$.

- If DS rejects $\langle M\rangle\langle Y\rangle$, then $\textsc{Diverge}(M) \neq \textsc{Diverge}(Y) = \varnothing$, so $\langle M\rangle \notin \textsc{NeverDiverge}$.

In short, ND accepts $\langle M\rangle$ if and only if $\langle M\rangle \in \textsc{NeverDiverge}$, which is impossible. We conclude that DS does not exist. □

8.10 Rice's Theorem

In 1953, Henry Rice proved the following extremely powerful theorem, which essentially states that ***every*** interesting question about the language accepted by a Turing machine is undecidable.

Rice's Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*
- *There is a Turing machine Y such that $\textsc{Accept}(Y) \in \mathcal{L}$.*
- *There is a Turing machine N such that $\textsc{Accept}(N) \notin \mathcal{L}$.*

The language $\textsc{AcceptIn}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) \in \mathcal{L}\right\}$ is undecidable.

Proof: Without loss of generality, suppose $\varnothing \notin \mathcal{L}$. (A symmetric argument establishes the theorem in the opposite case $\varnothing \in \mathcal{L}$.) Fix an arbitrary Turing machine Y such that $\textsc{Accept}(Y) \in \mathcal{L}$.

Suppose to the contrary that there is a Turing machine $A_{\mathcal{L}}$ that decides $\textsc{AcceptIn}(\mathcal{L})$. To derive a contradiction, we describe a Turing machine H that decides the halting language $\textsc{Halt}$, using $A_{\mathcal{L}}$ as a black-box subroutine. Given the encoding $\langle M, w\rangle$ of an arbitrary Turing machine M and an arbitrary string w as input, H writes the encoding $\langle WTF\rangle$ of a new Turing machine WTF that executes the following algorithm:

```
WTF(x):
  run M on input w (and discard the result)
  run Y on input x
```

H then passes the new encoding $\langle WTF\rangle$ to $A_{\mathcal{L}}$.

Now let M be an arbitrary Turing machine and w be an arbitrary string, and suppose we run our new Turing machine H on the encoding $\langle M, w\rangle$. There are two cases to consider.

- Suppose M halts on input w.
  - Then for all strings x, the machine WTF accepts x if and only if Y accepts x.
  - So $\textsc{Accept}(WTF) = \textsc{Accept}(Y)$, by definition of $\textsc{Accept}(\,\cdot\,)$.
  - So $\textsc{Accept}(WTF) \in \mathcal{L}$, by definition of Y.
  - So $A_{\mathcal{L}}$ accepts $\langle WTF\rangle$, because $A_{\mathcal{L}}$ decides $\textsc{AcceptIn}(\mathcal{L})$.
  - So H accepts $\langle M, w\rangle$, by definition of H.

- Suppose M does not halt on input w.
  - Then for all strings x, the machine WTF does not halt on input x, and therefore does not accept x.
  - So $\textsc{Accept}(WTF) = \varnothing$, by definition of $\textsc{Accept}(WTF)$.
  - So $\textsc{Accept}(WTF) \notin \mathcal{L}$, by our assumption that $\varnothing \notin \mathcal{L}$.
  - So $A_{\mathcal{L}}$ rejects $\langle WTF\rangle$, because $A_{\mathcal{L}}$ decides $\textsc{AcceptIn}(\mathcal{L})$.
  - So H rejects $\langle M, w\rangle$, by definition of H.

In short, H decides the language $\textsc{Halt}$, which is impossible. We conclude that $A_{\mathcal{L}}$ does not exist. $\square$

The set $\mathcal{L}$ in the statement of Rice's Theorem is often called a ***property*** of languages, rather than a *set*, to avoid the inevitable confusion about sets of sets. We can also think of $\mathcal{L}$ as a ***decision problem*** about languages, where the languages are represented by Turing machines that accept or decide them. Rice's theorem states that the ***only*** properties of languages that are decidable are the trivial properties "Does this Turing machine accept an acceptable language?" (Answer: Yes, by definition.) and "Does this Turing machine accept Discover?" (Answer: No, because Discover is a credit card, not a language.)

Rice's Theorem makes it incredibly easy to prove that language properties are undecidable; we only need to exhibit one acceptable language that has the property and another acceptable language that does not. In fact, most proofs using Rice's theorem can use at least one of the following Turing machines:

- $M_{\textsc{Accept}}$ accepts every string, by defining $\delta(\text{start}, a) = \text{accept}$ for every tape symbol a.
- $M_{\textsc{Reject}}$ rejects every string, by defining $\delta(\text{start}, a) = \text{reject}$ for every tape symbol a.
- $M_{\textsc{Diverge}}$ diverges on every string, by defining $\delta(\text{start}, a) = (\text{start}, a, +1)$ for every tape symbol a.

Corollary 14. *Each of the following languages is undecidable.*
(a) $\{\langle M\rangle \mid M$ *accepts given an empty initial tape*$\}$
(b) $\{\langle M\rangle \mid M$ *accepts the string* `UIUC`$\}$
(c) $\{\langle M\rangle \mid M$ *accepts exactly three strings*$\}$
(d) $\{\langle M\rangle \mid M$ *accepts all palindromes*$\}$
(e) $\{\langle M\rangle \mid \textsc{Accept}(M)$ *is regular*$\}$
(f) $\{\langle M\rangle \mid \textsc{Accept}(M)$ *is not regular*$\}$
(g) $\{\langle M\rangle \mid \textsc{Accept}(M)$ *is undecidable*$\}$
(h) $\{\langle M\rangle \mid \textsc{Accept}(M) = \textsc{Accept}(N)\}$, *for some arbitrary fixed Turing machine* N.

Proof: In all cases, undecidability follows from Rice's theorem.

(a) Let $\mathcal{L}$ be the set of all languages that contain the empty string. Then $\textsc{AcceptIn}(\mathcal{L}) = \{\langle M\rangle \mid M$ accepts given an empty initial tape$\}$.

  - Given an empty initial tape, $M_{\textsc{Accept}}$ accepts, so $\textsc{Halt}(M_{\textsc{Accept}}) \in \mathcal{L}$.
  - Given an empty initial tape, $M_{\textsc{Diverge}}$ does not accept, so $\textsc{Halt}(M_{\textsc{Diverge}}) \notin \mathcal{L}$.

  Therefore, Rice's Theorem implies that $\textsc{AcceptIn}(\mathcal{L})$ is undecidable.

(b) Let $\mathcal{L}$ be the set of all languages that contain the string `UIUC`.

  - $M_{\textsc{Accept}}$ accepts `UIUC`, so $\textsc{Halt}(M_{\textsc{Accept}}) \in \mathcal{L}$.
  - $M_{\textsc{Diverge}}$ does not accept `UIUC`, so $\textsc{Halt}(M_{\textsc{Diverge}}) \notin \mathcal{L}$.

  Therefore, $\textsc{AcceptIn}(\mathcal{L}) = \{\langle M\rangle \mid M$ accepts the string `UIUC`$\}$ is undecidable by Rice's Theorem.

(c) There is a Turing machine that accepts the language $\{$`larry`, `curly`, `moe`$\}$. On the other hand, $M_{\textsc{Reject}}$ does not accept exactly three strings.

(d) M_{ACCEPT} accepts all palindromes, and M_{REJECT} does not accept all palindromes.

(e) M_{REJECT} accepts the regular language $\varnothing$, and there is a Turing machine $M_{0^n1^n}$ that accepts the non-regular language $\{0^n1^n \mid n \geq 0\}$.

(f) M_{REJECT} accepts the regular language $\varnothing$, and there is a Turing machine $M_{0^n1^n}$ that accepts the non-regular language $\{0^n1^n \mid n \geq 0\}$.[5]

(g) M_{REJECT} accepts the decidable language $\varnothing$, and there is a Turing machine that accepts the undecidable language SELFREJECT.

(h) The Turing machine N accepts $\text{ACCEPT}(N)$ by definition. The Turing machine N^R, obtained by swapping the accept and reject states of N, accepts the language $\text{HALT}(L) \setminus \text{ACCEPT}(N) \neq \text{ACCEPT}(N)$. □

We can also use Rice's theorem as a component in more complex undecidability proofs, where the target language consists of more than just a single Turing machine encoding.

Theorem 15. *The language $L := \left\{\langle M, w\rangle \;\middle|\; M \text{ accepts } w^k \text{ for every integer } k \geq 0\right\}$ is undecidable.*

Proof: Fix an arbitrary string w, and let $\mathcal{L}$ be the set of all languages that contain w^k for all k. Then $\text{ACCEPT}(M_{\text{ACCEPT}}) = \Sigma^* \in \mathcal{L}$ and $\text{ACCEPT}(M_{\text{REJECT}}) = \varnothing \notin \mathcal{L}$. Thus, *even if the string w is fixed in advance*, no Turing machine can decide L. □

Nearly identical reduction arguments imply the following variants of Rice's theorem. (The names of these theorems are not standard.)

Rice's Rejection Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\text{REJECT}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\text{REJECT}(N) \notin \mathcal{L}$.*

The language $\text{REJECTIN}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \text{REJECT}(M) \in \mathcal{L}\right\}$ is undecidable.

Rice's Halting Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\text{HALT}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\text{HALT}(N) \notin \mathcal{L}$.*

The language $\text{HALTIN}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \text{HALT}(M) \in \mathcal{L}\right\}$ is undecidable.

Rice's Divergence Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\text{DIVERGE}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\text{DIVERGE}(N) \notin \mathcal{L}$.*

The language $\text{DIVERGEIN}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \text{DIVERGE}(M) \in \mathcal{L}\right\}$ is undecidable.

Rice's Decision Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that* ***decides*** *an language in $\mathcal{L}$.*
- *There is a Turing machine N such that* ***decides*** *an language not in $\mathcal{L}$.*

The language $\text{DECIDEIN}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; M \textbf{ decides} \text{ a language in } \mathcal{L}\right\}$ is undecidable.

[5]Yes, parts (e) and (f) have exactly the same proof.

As a final sanity check, always be careful to distinguish the following objects:

- The string ε
- The language $\varnothing$
- The language $\{\varepsilon\}$
- The language property $\varnothing$
- The language property $\{\varnothing\}$
- The language property $\{\{\varepsilon\}\}$
- The Turing machine M_{REJECT} that rejects every string and therefore ***decides*** the language $\varnothing$.
- The Turing machine M_{DIVERGE} that diverges on every string and therefore ***accepts*** the language $\varnothing$.

*8.11 The Rice-McNaughton-Myhill-Shapiro Theorem

The following subtle generalization of Rice's theorem precisely characterizes which properties of acceptable languages are *acceptable*. This result was partially proved by Henry Rice in 1953, in the same paper that proved Rice's Theorem; Robert McNaughton, John Myhill, and Norman Shapiro completed the proof a few years later, each independently from the other two.[6]

The Rice-McNaughton-Myhill-Shapiro Theorem. *Let $\mathcal{L}$ be an arbitrary set of acceptable languages. The language* $\text{AcceptIn}(\mathcal{L}) := \{\langle M\rangle \mid \text{Accept}(M) \in \mathcal{L}\}$ *is **acceptable** if and only if $\mathcal{L}$ satisfies the following conditions:*

*(a) $\mathcal{L}$ is **monotone**: For any language $L \in \mathcal{L}$, every superset of L is also in $\mathcal{L}$.*

*(b) $\mathcal{L}$ is **compact**: Every language in $\mathcal{L}$ has a finite subset that is also in $\mathcal{L}$.*

*(c) $\mathcal{L}$ is **finitely acceptable**: The language $\big\{\langle L\rangle \mid L \in \mathcal{L}$ and L is finite$\big\}$ is acceptable.*[7]

I won't give a complete proof of this theorem (in part because it requires techniques I haven't introduced), but the following lemma is arguably the most interesting component:

Lemma 16. *Let $\mathcal{L}$ be a set of acceptable languages. If $\mathcal{L}$ is not monotone, then $\text{AcceptIn}(\mathcal{L})$ is unacceptable.*

Proof: Suppose to the contrary that there is a Turing machine $AI_{\mathcal{L}}$ that accepts $\text{AcceptIn}(\mathcal{L})$. Using this Turing machine as a black box, we describe a Turing machine SD that accepts the unacceptable language SelfDiverge. Fix two Turing machines Y and N such that

$$\begin{aligned} \text{Accept}(Y) &\in \mathcal{L}, \\ \text{Accept}(N) &\notin \mathcal{L}, \\ \text{and} \quad \text{Accept}(Y) &\subseteq \text{Accept}(N). \end{aligned}$$

Let w be the input to SD. After verifying that $w = \langle M\rangle$ for some Turing machine M (and rejecting otherwise), SD writes the encoding $\langle WTF\rangle$ or a new Turing machine WTF that implements the following algorithm:

[6]McNaughton never published his proof (although he did announce the result); consequently, this theorem is sometimes called "The Rice-Myhill-Shapiro Theorem". Even more confusingly, Myhill published his proof twice, once in a paper with John Shepherdson and again in a later paper with Jacob Dekker. So maybe it should be called the Rice–Dekker-Myhill–McNaughton–Myhill-Shepherdson–Shapiro Theorem.

[7]Here the encoding $\langle L\rangle$ of a finite language $L \subseteq \Sigma^*$ is exactly the string that you would write down to explicitly describe L. Formally, $\langle L\rangle$ is the unique string over the alphabet $\Sigma \cup \{\texttt{\{}, \texttt{,}, \texttt{\}}, \texttt{ε}\}$ that contains the strings in L in lexicographic order, separated by commas `,` and surrounded by braces `{}`, with `ε` representing the empty string. For example, $\big\langle\{\varepsilon, \texttt{0}, \texttt{01}, \texttt{0110}, \texttt{01101001}\}\big\rangle = $ `{ε,0,01,0110,01101001}`.

```
WTF(x):
  write x to second tape
  write ⟨M⟩ to third tape
  in parallel:
      run Y on the first tape
      run N on the second tape
      run M on the third tape
  if Y accepts x
      accept
  if N accepts x and M halts on ⟨M⟩
      accept
```

Finally, SD passes the new encoding $\langle WTF\rangle$ to $AI_{\mathcal{L}}$. There are two cases to consider:

- If M halts on $\langle M\rangle$, then $\text{Accept}(WTF) = \text{Accept}(N) \notin \mathcal{L}$, and therefore $AI_{\mathcal{L}}$ does not accept $\langle WTF\rangle$.
- If M does not halt on $\langle M\rangle$, then $\text{Accept}(WTF) = \text{Accept}(Y) \in \mathcal{L}$, and therefore $AI_{\mathcal{L}}$ accepts $\langle WTF\rangle$.

In short, SD accepts SelfDiverge, which is impossible. We conclude that SD does not exist. □

Corollary 17. *Each of the following languages is unacceptable.*
(a) $\{\langle M\rangle \mid \text{Accept}(M) \textit{ is finite}\}$
(b) $\{\langle M\rangle \mid \text{Accept}(M) \textit{ is infinite}\}$
(c) $\{\langle M\rangle \mid \text{Accept}(M) \textit{ is regular}\}$
(d) $\{\langle M\rangle \mid \text{Accept}(M) \textit{ is not regular}\}$
(e) $\{\langle M\rangle \mid \text{Accept}(M) \textit{ is decidable}\}$
(f) $\{\langle M\rangle \mid \text{Accept}(M) \textit{ is undecidable}\}$
(g) $\{\langle M\rangle \mid M \textit{ accepts at least one string in } \text{SelfDiverge}\}$
(h) $\{\langle M\rangle \mid \text{Accept}(M) = \text{Accept}(N)\}$*, for some arbitrary fixed Turing machine N.*

Proof: (a) The set of finite languages is not monotone: $\varnothing$ is finite; Σ^* is not finite; both $\varnothing$ and Σ^* are acceptable (in fact decidable); and $\varnothing \subset \Sigma^*$.

(b) The set of infinite acceptable languages is not compact: No finite subset of the infinite acceptable language Σ^* is infinite!

(c) The set of regular languages is not monotone: Consider the languages $\varnothing$ and $\{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$.

(d) The set of non-regular acceptable languages is not monotone: Consider the languages $\{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$ and Σ^*.

(e) The set of decidable languages is not monotone: Consider the languages $\varnothing$ and SelfReject.

(f) The set of undecidable acceptable languages is not monotone: Consider the languages SelfReject and Σ^*.

(g) The set $\mathcal{L} = \{L \mid L \cap \text{SelfDiverge} \neq \varnothing\}$ is not finitely acceptable. For any string w, deciding whether $\{w\} \in \mathcal{L}$ is equivalent to deciding whether $w \in \text{SelfDiverge}$, which is impossible.

(h) If $\text{Accept}(N) \neq \Sigma^*$, then the set $\{\text{Accept}(N)\}$ is not monotone. On the other hand, if $\text{Accept}(N) = \Sigma^*$, then the set $\{\text{Accept}(N)\}$ is not compact: No finite subset of Σ^* is equal to Σ^*! □

8.12 Turing Machine Behavior: It's Complicated

Rice's theorems imply that every interesting question about the language that a Turing machine accepts—or more generally, the function that a program computes—is undecidable. A more subtle question is whether we can recognize Turing machines that exhibit certain *internal behavior*. Some behaviors we can recognize; others we can't.

Theorem 18. *The language* NEVERLEFT $:= \{\langle M, w\rangle \mid$ *Given* w *as input,* M *never moves left*$\}$ *is decidable.*

Proof: Given the encoding $\langle M, w\rangle$, we simulate M with input w using our universal Turing machine U, but with the following termination conditions. If M ever moves its head to the left, then we reject. If M halts without moving its head to the left, then we accept. Finally, if M reads more than $|Q|$ blanks, where Q is the state set of M, then we accept. If the first two cases do not apply, M only moves to the right; moreover, after reading the entire input string, M only reads blanks. Thus, after reading $|Q|$ blanks, it must repeat some state, and therefore loop forever without moving to the left. The three cases are exhaustive. □

Theorem 19. *The language* LEFTTHREE $:= \{\langle M, w\rangle \mid$ *Given* w *as input,* M *eventually moves left three times in a row*$\}$ *is undecidable.*

Proof: Given $\langle M\rangle$, we build a new Turing machine M' that accepts the same language as M and moves left three times in a row if and only if it accepts, as follows. For each non-accepting state p of M, the new machine M' has three states p_1, p_2, p_3, with the following transitions:

$$\begin{aligned}
\delta'(p_1, a) &= (q_2, b, \Delta), && \text{where } (q, b, \Delta) = \delta(p, a) \text{ and } q \neq \text{accept} \\
\delta'(p_2, a) &= (p_3, a, +1) \\
\delta'(p_3, a) &= (p_1, a, -1)
\end{aligned}$$

In other words, after each non-accepting transition, M' moves once to the right and then once to the left. For each transition to accept, M' has a sequence of seven transitions: three steps to the right, then three steps to the left, and then finally accept', all without modifying the tape. (The three steps to the right ensure that M' does not fall off the left end of the tape.)

Finally, M' moves left three times in a row if and only if M accepts w. Thus, if we could decide LEFTTHREE, we could also decide ACCEPT, which is impossible. □

There is no hard and fast rule like Rice's theorem to distinguish decidable behaviors from undecidable behaviors, but I can offer two rules of thumb.

- If it is possible to simulate an arbitrary Turing machine while avoiding the target behavior, then the behavior is not decidable. For example: there is no algorithm to determine whether a given Turing machine reenters its start state, or revisits the left end of the tape, or writes a blank.
- If a Turing machine with the target behavior is limited to a finite number of configurations, or is guaranteed to force an infinite loop after a finite number of transitions, then the behavior is likely to be decidable. For example, there *are* algorithms to determine whether a given Turing machine ever leaves its start state, or reads its entire input string, or writes a non-blank symbol over a blank.

Exercises

1. Let M be an arbitrary Turing machine.
   (a) Describe a Turing machine M^R such that
   $$\text{Accept}(M^R) = \text{Reject}(M) \quad \text{and} \quad \text{Reject}(M^R) = \text{Accept}(M).$$
   (b) Describe a Turing machine M^A such that
   $$\text{Accept}(M^A) = \text{Accept}(M) \quad \text{and} \quad \text{Reject}(M^A) = \varnothing.$$
   (c) Describe a Turing machine M^H such that
   $$\text{Accept}(M^H) = \text{Halt}(M) \quad \text{and} \quad \text{Reject}(M^H) = \varnothing.$$

2. (a) Prove that Accept is undecidable.
   (b) Prove that Reject is undecidable.
   (c) Prove that Diverge is undecidable.

3. (a) Prove that NeverReject is undecidable.
   (b) Prove that NeverHalt is undecidable.
   (c) Prove that NeverDiverge is undecidable.

4. Prove that each of the following languages is undecidable.
   (a) $\text{AlwaysAccept} := \{\langle M\rangle \mid \text{Accept}(M) = \Sigma^*\}$
   (b) $\text{AlwaysReject} := \{\langle M\rangle \mid \text{Reject}(M) = \Sigma^*\}$
   (c) $\text{AlwaysHalt} := \{\langle M\rangle \mid \text{Halt}(M) = \Sigma^*\}$
   (d) $\text{AlwaysDiverge} := \{\langle M\rangle \mid \text{Diverge}(M) = \Sigma^*\}$

5. Let $\mathcal{L}$ be a non-empty proper subset of the set of acceptable languages. Prove that the following languages are undecidable:
   (a) $\text{RejectIn}(\mathcal{L}) := \big\{\langle M\rangle \;\big|\; \text{Reject}(M) \in \mathcal{L}\big\}$
   (b) $\text{HaltIn}(\mathcal{L}) := \big\{\langle M\rangle \;\big|\; \text{Halt}(M) \in \mathcal{L}\big\}$
   (c) $\text{DivergeIn}(\mathcal{L}) := \big\{\langle M\rangle \;\big|\; \text{Diverge}(M) \in \mathcal{L}\big\}$

6. For each of the following decision problems, either *sketch* an algorithm or prove that the problem is undecidable. Recall that w^R denotes the reversal of string w. For each problem, the input is the encoding $\langle M\rangle$ of a Turing machine M.
   (a) Does M accept $\langle M\rangle^R$?
   (b) Does M reject any palindrome?
   (c) Does M accept all palindromes?

(d) Does M diverge only on palindromes?

(e) Is there an input string that forces M to move left?

(f) Is there an input string that forces M to move left three times in a row?

(g) Does M accept the encoding of any Turing machine N such that $\textsc{Accept}(N) = \textsc{SelfDiverge}$?

7. For each of the following decision problems, either *sketch* an algorithm or prove that the problem is undecidable. Recall that w^R denotes the reversal of string w. For each problem, the input is an encoding $\langle M, w\rangle$ of a Turing machine M and its input string w.

   (a) Does M accept the string ww^R?

   (b) Does M accept either w or w^R?

   (c) Does M either accept w or reject w^R?

   (d) Does M accept the string w^k for some integer k?

   (e) Does M accept w in at most $2^{|w|}$ steps?

   (f) If we run M on input w, does M ever change a symbol on its tape?

   (g) If we run M on input w, does M ever move to the right?

   (h) If we run M on input w, does M ever move to the right twice in a row?

   (i) If we run M on input w, does M move its head to the right more than $2^{|w|}$ times (not necessarily consecutively)?

   (j) If we run M with input w, does M ever change a $\square$ on the tape to any other symbol?

   (k) If we run M with input w, does M ever change a $\square$ on the tape to 1?

   (l) If we run M with input w, does M ever write a $\square$?

   (m) If we run M with input w, does M ever leave its start state?

   (n) If we run M with input w, does M ever reenter its start state?

   (o) If we run M with input w, does M ever reenter a state that it previously left? That is, are there states $p \neq q$ such that M moves from state p to state q and then later moves back to state p?

8. Let M be a Turing machine, let w be an arbitrary input string, and let s and t be positive integers integer. We say that M accepts w ***in space s*** if M accepts w after accessing at most the first s cells on the tape, and M accepts w ***in time t*** if M accepts w after at most t transitions.

   (a) Prove that the following languages are decidable:

      i. $\{\langle M, w\rangle \mid M \text{ accepts } w \text{ in time } |w|^2\}$

      ii. $\{\langle M, w\rangle \mid M \text{ accepts } w \text{ in space } |w|^2\}$

   (b) Prove that the following languages are undecidable:

      i. $\{\langle M\rangle \mid M \text{ accepts at least one string } w \text{ in time } |w|^2\}$

      ii. $\{\langle M\rangle \mid M \text{ accepts at least one string } w \text{ in space } |w|^2\}$

9. Let L_0 be an arbitrary language. For any integer $i > 0$, define the language

$$L_i := \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_{i-1} \right\}.$$

For which integers $i > 0$ is L_i decidable? Obviously the answer depends on the initial language L_0; give a complete characterization of all possible cases. Prove your answer is correct. *[Hint: This question is a lot easier than it looks!]*

10. Argue that each of the following decision problems about programs in your favorite programming language are undecidable.

    (a) Does this program correctly compute Fibonacci numbers?

    (b) Can this program fall into an infinite loop?

    (c) Will the value of this variable ever change?

    (d) Will this program every attempt to deference a null pointer?

    (e) Does this program free every block of memory that it dynamically allocates?

    (f) Is any statement in this program unreachable?

    (g) Do these two programs compute the same function?

*11. Call a Turing machine ***conservative*** if it never writes over its input string. More formally, a Turing machine is conservative if for every transition $\delta(p, a) = (q, b, \Delta)$ where $a \in \Sigma$, we have $b = a$; and for every transition $\delta(p, a) = (q, b, \Delta)$ where $a \notin \Sigma$, we have $b \neq \Sigma$.

    (a) Prove that if M is a conservative Turing machine, then $\textsc{Accept}(M)$ is a regular language.

    (b) Prove that the language $\{\langle M \rangle \mid M \text{ is conservative and } M \text{ accepts } \varepsilon\}$ is undecidable.

    Together, these two results imply that every conservative Turing machine accepts the same language as some DFA, but it is impossible to determine *which* DFA.

★12. (a) Prove that it is undecidable whether a given C++ program is syntactically correct. *[Hint: Use templates!]*

    (b) Prove that it is undecidable whether a given ANSI C program is syntactically correct. *[Hint: Use the preprocessor!]*

    (c) Prove that it is undecidable whether a given Perl program is syntactically correct. *[Hint: Does that slash character / delimit a regular expression or represent division?]*

Caveat lector: This is the zeroth (draft) edition of this lecture note. In particular, some topics still need to be written. Please send bug reports and suggestions to jeffe@illinois.edu.

If first you don't succeed, then try and try again.
And if you don't succeed again, just try and try and try.

— Marc Blitzstein, "Useless Song", *The Three Penny Opera* (1954)
Adaptation of Bertold Brecht, "Das Lied von der Unzulänglichkeit menschlichen Strebens" *Die Dreigroschenoper* (1928)

Children need encouragement.
If a kid gets an answer right, tell him it was a lucky guess.
That way he develops a good, lucky feeling.

— Jack Handey, "Deep Thoughts", *Saturday Night Live* (March 21, 1992)

9 Nondeterministic Turing Machines

9.1 Definitions

In his seminal 1936 paper, Turing also defined an extension of his "automatic machines" that he called ***choice machines***, which are now more commonly known as ***nondeterministic Turing machines***. The execution of a nondeterministic Turing machine is *not determined* entirely by its input and its transition function; rather, at each step of its execution, the machine can *choose* from a set of possible transitions. The distinction between deterministic and nondeterministic Turing machines exactly parallels the distinction between deterministic and nondeterministic finite-state automata.

Formally, a nondeterministic Turing machine has all the components of a standard deterministic Turing machine—a finite tape alphabet Γ that contains the input alphabet Σ and a blank symbol $\square$; a finite set Q of internal states with special start, accept, and reject states; and a transition function δ. However, the transition function now has the signature

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{-1,+1\}}.$$

That is, for each state p and tape symbol a, the output $\delta(p, a)$ of the transition function is a *set* of triples of the form $(q, b, \Delta) \in Q \times \Gamma \times \{-1, +1\}$. Whenever the machine finds itself in state p reading symbol a, the machine *chooses* an arbitrary triple $(q, b, \Delta) \in \delta(p, a)$, and then changes its state to q, writes b to the tape, and moves the head by Δ. If the set $\delta(p, a)$ is empty, the machine moves to the reject state and halts.

The set of all possible transition sequences of a nondeterministic Turing machine N on a given input string w define a rooted tree, called a ***computation tree***. The initial configuration $(\text{start}, w, 0)$ is the root of the computation tree, and the children of any configuration (q, x, i) are the configurations that can be reached from (q, x, i) in one transition. In particular, any configuration whose state is accept or reject is a leaf. For deterministic Turing machines, this computation tree is just a single path, since there is at most one valid transition from every configuration.

9.2 Acceptance and Rejection

Unlike deterministic Turing machines, there is a fundamental asymmetry between the acceptance and rejection criteria for nondeterministic Turing machines. Let N be any nondeterministic Turing machine, and let w be any string.

- ***N accepts w*** if and only if there is *at least one* sequence of valid transitions from the initial configuration $(\text{start}, w, 0)$ that leads to the accept state. Equivalently, N accepts w if the computation tree contains at least one accept leaf.
- ***N rejects w*** if and only if *every* sequence of valid transitions from the initial configuration $(\text{start}, w, 0)$ leads to the reject state. Equivalently, N rejects w if every path through the computation tree ends with a reject leaf.

In particular, N can accept w even when there are choices that allow the machine to run forever, but rejection requires N to halt after only a finite number of transitions, no matter what choices it makes along the way. Just as for deterministic Turing machines, it is possible that N neither accepts nor rejects w.

Acceptance and rejection of *languages* are defined exactly as they are for deterministic machines. A non-deterministic Turing machine N ***accepts*** a language $L \subseteq \Sigma^*$ if M accepts all strings in L and nothing else; N ***rejects*** L if M rejects every string in L and nothing else; and finally, N ***decides*** L if M accepts L and rejects $\Sigma^* \setminus L$.

9.3 Time and Space Complexity

★★★

- Define "time" and "space".
- TIME($f(n)$) is the class of languages that can be decided by a deterministic multi-tape Turing machine in $O(f(n))$ time.
- NTIME($f(n)$) is the class of languages that can be decided by a nondeterministic multi-tape Turing machine in $O(f(n))$ time.
- SPACE($f(n)$) is the class of languages that can be decided by deterministic multi-tape Turing machine in $O(f(n))$ space.
- NSPACE($f(n)$) is the class of languages that can be decided by a nondeterministic multi-tape Turing machine in $O(f(n))$ space.
- Why multi-tape TMs? Because t steps on any k-tape Turing machine can be simulated in $O(t \log t)$ steps on a two-tape machine [Hennie and Stearns 1966, essentially using lazy counters and amortization], and in $O(t^2)$ steps on a single-tape machine [Hartmanis and Stearns 1965; realign multiple tracks at every simulation step]. Moreover, the latter quadratic bound is tight [Hennie 1965 (palindromes, via communication complexity)].

9.4 Deterministic Simulation

Theorem 1. *For any nondeterministic Turing machine N, there is a deterministic Turing machine M that accepts exactly the same strings and N and rejects exactly the same strings as N. Moreover, if every computation path of N on input x halts after at most t steps, then M halts on input x after at most $O(t^2 r^{2t})$ steps, where r is the maximum size of any transition set in N.*

Proof: I'll describe a deterministic machine M that performs a breadth-first search of the computation tree of N. (The depth-first search performed by a standard recursive backtracking algorithm won't work here. If N's computation tree contains an infinite path, a depth-first search would get stuck in that path without exploring the rest of the tree.)

At the beginning of each simulation round, M's tape contains a string of the form

$$\Box\Box\cdots\Box\bullet\bullet y_1q_1z_1\bullet y_2q_2z_2\bullet\cdots\bullet y_kq_kz_k\bullet\bullet$$

where each substring $y_iq_iz_i$ encodes a configuration $(q_i, y_iz_i, |y_i|)$ of some computation path of N, and $\bullet$ is a new symbol not in the tape alphabet of N. The machine M interprets this sequence of encoded configurations as a queue, with new configurations inserted on the right and old configurations removed from the left. The double-separators $\bullet\bullet$ uniquely identify the start and end of this queue; outside this queue, the tape is entirely blank.

Specifically, in each round, first M appends the encodings of all configurations than N can reach in one transition from the first encoded configuration $(q_1, y_1z_1, |y_1|)$; then M erases the first encoded configuration.

$$\cdots\Box\Box\underline{\bullet\bullet y_1q_1z_1}\bullet y_2q_2z_2\bullet\cdots\bullet y_rq_rz_r\bullet\underline{\bullet\Box\Box\cdots}$$

$$\Downarrow \qquad\qquad\qquad\qquad \Downarrow$$

$$\cdots\Box\Box\underline{\Box\Box\cdots\Box\bullet}\bullet y_2q_2z_2\bullet\cdots\bullet y_kq_kz_k\bullet\underline{\tilde{y}_1\tilde{q}_1\tilde{z}_1\bullet\tilde{y}_2\tilde{q}_2\tilde{z}_2\bullet\cdots\bullet\tilde{y}_r\tilde{q}_r\tilde{z}_r\bullet\bullet}\Box\Box\cdots$$

Suppose each transition set $\delta_N(q,a)$ has size at most r. Then after simulating t steps of N, the tape string of M encoding $O(r^t)$ different configurations of N and therefore has length $L = O(tr^t)$ (not counting the initial blanks). If M begins each simulation phase by moving the initial configuration from the beginning to the end of the tape string, which takes $O(t^2r^t)$ time, the time for the rest of the the simulation phase is negligible. Altogether, simulating all r^t possibilities for the the tth step of N requires $O(t^2r^{2t})$ time. We conclude that M can simulate the first t steps of every computation path of N in $\boldsymbol{O(t^2r^{2t})}$ ***time***, as claimed. $\Box$

The running time of this simulation is dominated by the time spent reading from one end of the tape string and writing to the other. It is fairly easy to reduce the running time to $O(tr^t)$ by using either two tapes (a "read tape" containing N-configurations at time t and a "write tape" containing N-configurations at time $t+1$) or two independent heads on the same tape (one at each end of the queue).

9.5 Nondeterminism as Advice

Any nondeterministic Turing machine N can also be simulated by a *deterministic* machine M with *two* inputs: the user input string $w \in \Sigma^*$, and a so-called ***advice*** string $x \in \Omega^*$, where Ω is another finite alphabet. Only the first input string w is actually given by the user. At least for now, we assume that the advice string x is given on a separate read-only tape.

The deterministic machine M simulates N step-by-step, but whenever N has a choice of how to transition, M reads a new symbol from the advice string, and that symbol determines the choice. In fact, without loss of generality, we can assume that M reads a new symbol from the advice string and moves the advice-tape's head to the right on *every* transition. Thus, M's transition function has the form $\delta_M : Q \times \Gamma \times \Omega \to Q \times \Gamma \times \{-1, +1\}$, and we require that

$$\delta_N(q,a) = \{\delta_M(q,a,\omega) \mid \omega \in \Omega\}$$

For example, if N has a binary choice

$$\delta_N(\mathsf{branch}, \mathsf{?}) = \big\{(\mathsf{left}, \mathsf{L}, -1), (\mathsf{right}, \mathsf{R}, +1)\big\},$$

then M might determine this choice by defining

$$\delta_M(\mathtt{branch}, \mathtt{?}, \mathtt{0}) = (\mathtt{left}, \mathtt{L}, -1) \qquad \text{and} \qquad \delta_M(\mathtt{branch}, \mathtt{?}, \mathtt{1}) = (\mathtt{right}, \mathtt{R}, +1)$$

More generally, if every set $\delta_N(p, a)$ has size r, then we let $\Omega = \{1, 2, \ldots, r\}$ and define $\delta_M(q, a, i)$ to be the ith element of $\delta_N(q, a)$ in some canonical order.

Now observe that N accepts a string w if and only if M accepts the pair (w, x) for *some* string $x \in \Omega^*$, and N rejects w if and only if M rejects the pair (w, x) for *all* strings $x \in \Omega^*$.

The "advice" formulation of nondeterminism allows a different strategy for simulation by a standard deterministic Turing machine, which is often called ***dovetailing***. Consider all possible advice strings x, in increasing order of length; listing these advice strings is equivalent to repeatedly incrementing a base-r counter. For each advice string x, simulate M on input (w, x) for exactly $|x|$ transitions.

```
DOVETAIL_M(w):
  for t ← 1 to ∞
      done ← TRUE
      for all strings x ∈ Ω^t
          if M accepts (w, x) in at most t steps
              accept
          if M(w, x) does not halt in at most t steps
              done ← FALSE
      if done
          reject
```

The most straightforward Turing-machine implementation of this algorithm requires three tapes: A read-only input tape containing w, an advice tape containing x (which is also used as a timer for the simulation), and the work tape. This simulation requires $O(tr^t)$ time to simulate all possibilities for t steps of the original non-deterministic machine N.

If we insist on using a standard Turing machine with a single tape and a single head, the simulation becomes slightly more complex, but (unlike our earlier queue-based strategy) not significantly slower. This standard machine S maintains a string of the form $\bullet w \bullet x \bullet z$, where z is the current work-tape string of M (or equivalently, of N), with marks (on a second track) indicating the current positions of the heads on M's work tape and M's advice tape. Simulating a single transition of M now requires $O(|x|)$ steps, because S needs to shuttle its single head between these two marks. Thus, S requires $\boldsymbol{O(t^2 r^t)}$ ***time*** to simulate all possibilities for t steps of the original non-deterministic machine N. This is significantly faster than the queue-based simulation, because we don't record (and therefore don't have to repeatedly scan over) intermediate configurations; recomputing everything from scratch is actually cheaper!

9.6 The Cook-Levin Theorem

★★★ Define SAT and CIRCUITSAT. Non-determinism is fundamentally different from other Turing machine extensions, in that it seems to provide an exponential speedup for some problems, just like NFAs can use exponentially fewer states than DFAs for the same language.

The Cook-Levin Theorem. *If SAT $\in$ P, then P=NP.*

Proof: Let $L \subseteq \Sigma^*$ be an arbitrary language in NP, over some fixed alphabet Σ. There must be an integer k and Turing machine M that satisfies the following conditions:

- For all strings $w \in L$, there is at least one string $x \in \Sigma^*$ such that M accepts the string $w\square x$.
- For all strings $w \notin L$ and $x \in \Sigma^*$, M rejects the string $w\square x$.
- For all strings $w, x \in \Sigma^*$, M halts on input $w\square x$ after at most $\max\{1, |w|^k\}$ steps.

Now suppose we are given a string $w \in \Sigma^*$. Let $n = |w|$ and let $N = \max\{1, |w|^k\}$. We construct a boolean formula Φ_w that is satisfiable if and only if $w \in L$, by following the execution of M on input $w\square x$ for some unknown advice string x. Without loss of generality, we can assume that $|x| = N - n - 1$ (since we can extend any shorter string x with blanks.) Our formula Φ_w uses the following boolean variables for all symbols $a \in \Gamma$, all states $q \in Q$, and all integers $0 \leq t \leq N$ and $0 \leq i \leq N + 1$.

- $Q_{t,i,q}$ — M is in state q with its head at position i after t transitions.
- $T_{t,i,a}$ — The kth cell of M's work tape contains a after t transitions.

The formula Φ_w is the conjunction of the following constraints:

- **Boundaries:** To simplify later constraints, we include artificial boundary variables just past both ends of the tape:

$$Q_{t,i,q} = Q_{t,N+1,q} = \text{False} \qquad \text{for all } 0 \leq t \leq N \text{ and } q \in Q$$
$$T_{t,0,a} = T_{t,N+1,a} = \text{False} \qquad \text{for all } 0 \leq t \leq N \text{ and } a \in \Gamma$$

- **Initialization:** We have the following values for variables with $t = 0$:

$$\begin{aligned} Q_{0,1,\text{start}} &= \text{True} & & \\ Q_{0,1,q} &= \text{False} & & \text{for all } q \neq \text{start} \\ H_{0,i,q} &= \text{False} & & \text{for all } i \neq 1 \text{ and } q \in Q \\ T_{0,i,w_i} &= \text{True} & & \text{for all } 1 \leq i \leq n \\ T_{0,i,a} &= \text{False} & & \text{for all } 1 \leq i \leq n \text{ and } a \neq w_i \\ T_{0,n+1,\square} &= \text{True} & & \\ T_{0,i,a} &= \text{False} & & \text{for all } a \neq \square \end{aligned}$$

- **Uniqueness:** The variables $T_{0,i,a}$ with $n + 2 \leq i \leq N$ represent the unknown advice string x; these are the "inputs" to Φ_w. We need some additional constraints ensure that for each i, *exactly one* of these variables is True:

$$\left(\bigvee_{a\in\Gamma} T_{0,j,a}\right) \wedge \bigwedge_{a\neq b} \left(\overline{T_{0,j,a}} \vee \overline{T_{0,j,b}}\right)$$

- **Transitions:** For all $1 \leq t \leq N$ and $1 \leq i \leq N$, the following constraints simulate the transition from time $t - 1$ to time t.

$$Q_{t,i,q} = \bigvee_{\delta(p,a)=(q,\cdot,+1)} \left(Q_{t-1,i-1,p} \wedge T_{t,i-1,a}\right) \vee \bigvee_{\delta(p,a)=(q,\cdot,-1)} \left(Q_{t-i,i+1,p} \wedge T_{t,i+1,a}\right)$$

$$T_{t,i,b} = \bigvee_{\delta(p,a)=(\cdot,b,\cdot)} \left(Q_{t-1,i,p} \wedge T_{t-1,i,a}\right) \vee \left(\bigwedge_{q\in Q} \overline{Q_{t-1,i,q}} \wedge T_{t-1,i,b}\right)$$

- **Output:** We have one final constraint that indicates acceptance.

$$z = \bigvee_{t=0}^{N} \bigvee_{i=1}^{N} Q_{t,i,\text{accept}}$$

A straightforward induction argument implies that *without the acceptance constraint*, any assignment of values to the unknown variables $T_{0,i,a}$ that satisfies the uniqueness constraints determines *unique* values for the other variables in Φ_w, which consistently describe the execution of M. Thus, Φ_w is satisfiable if and only if for some input values $T_{0,i,a}$, the resulting , including acceptance. In other words, Φ_w is satisfiable if and only if there is a string $x \in \Gamma^*$ such that M accepts the input $w\Box x$. We conclude that Φ_w is satisfiable if and only if $w \in L$.

For any any string w of length n, the formula Φ_w has $O(N^2)$ variables and $O(N^2)$ constraints (where the hidden constants depend on the machine M). Every constraint except acceptance has constant length, so altogether Φ_w has length $O(N^2)$. Moreover, we can construct Φ_w in $O(N^2) = O(n^{2k})$ time.

In conclusion, if we could decide SAT for formulas of size n in (say) $O(n^c)$ time, then we could decide membership in L in $O(n^{2kc})$ time, which implies that $L \in$ P. □

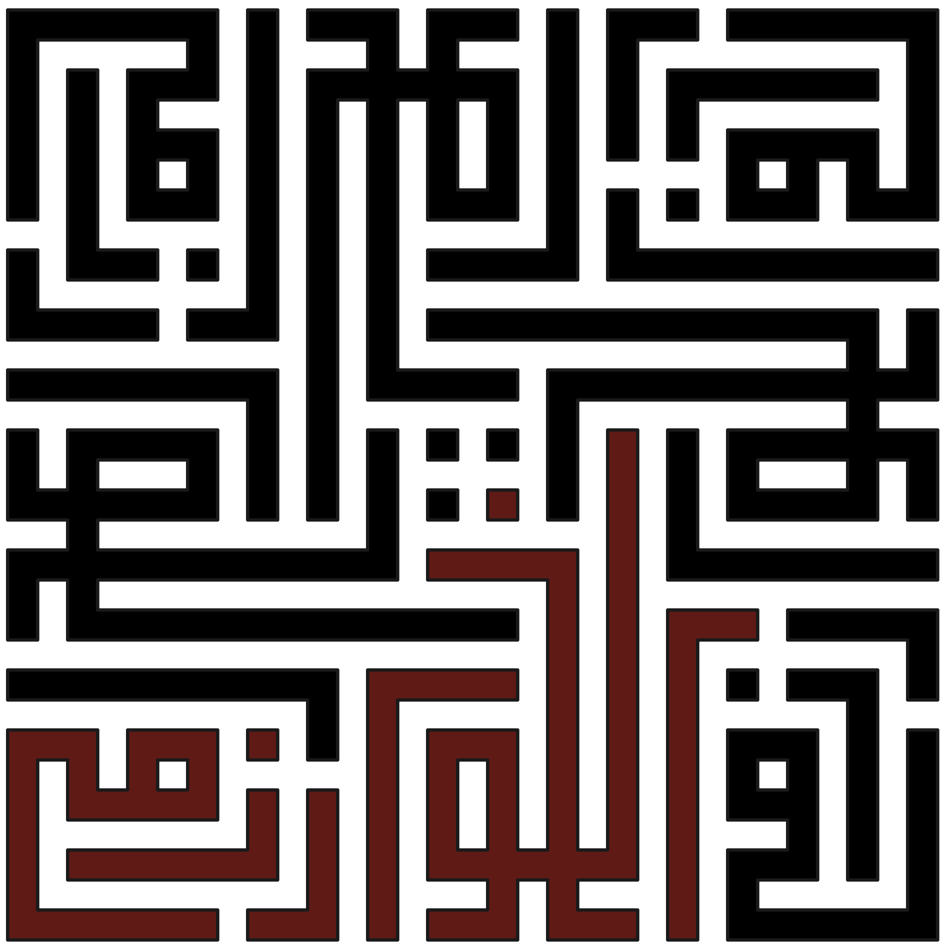
Exercises

1. Prove that the following problem is NP-hard, *without* using the Cook-Levin Theorem. Given a string $\langle M, w\rangle$ that encodes a non-deterministic Turing machine M and a string w, does M accept w in at most $|w|$ transitions?

★★★ More excersises!

Algorithms

Jeff Erickson



January 4, 2015

http://www.cs.illinois.edu/~jeffe/teaching/algorithms/

Shall I tell you, my friend, how you will come to understand it?
Go and write a book on it.

— Henry Home, Lord Kames (1696–1782), to Sir Gilbert Elliot

The individual is always mistaken. He designed many things, and drew in other persons as coadjutors, quarrelled with some or all, blundered much, and something is done; all are a little advanced, but the individual is always mistaken. It turns out somewhat new and very unlike what he promised himself.

— Ralph Waldo Emerson, "Experience", *Essays, Second Series* (1844)

Theoretical lectures should neither be a reproduction of
nor a comment upon any text-book, however satisfactory.
The student's notebook should be his principal text-book.

— André Weil, "Mathematical Teaching in Universities" (1954)

About These Notes

These are lecture notes that I wrote for various algorithms classes at the University of Illinois at Urbana-Champaign, which I have taught on average once a year since January 1999. The most recent revision of these notes (or nearly so) is available online at http://www.cs.illinois.edu/~jeffe/teaching/algorithms/, along with a near-complete archive of all my past homeworks and exams. Whenever I teach an algorithms class, I revise, update, and sometimes cull these notes as the course progresses, so you may find more recent versions on the web page of whatever course I am currently teaching.

With few exceptions, each of these "lecture notes" contains far too much material to cover in a single lecture. In a typical 75-minute class period, I cover about 4 or 5 pages of material—a bit more if I'm teaching graduate students than undergraduates. Moreover, I can only cover at most two-thirds of these notes in *any* capacity in a single 15-week semester. Your mileage may vary! (Arguably, that means that as I continue to add material, the label "lecture notes" becomes less and less accurate.) I teach algorithms at multiple leaves; different courses cover different but overlapping subsets of this material. The ordering of the notes is mostly consistent with my lower-level classes, with more advanced material (indicated by *stars) inserted near the more basic material it builds on. The actual material doesn't permit a strict linear ordering, but I've tried to keep forward references to a minimum.

About the Exercises

Each note ends with several exercises, most of which have been used at least once in a homework assignment, discussion section, or exam. *Stars indicate more challenging problems; many of these starred problems appeared on qualifying exams for the algorithms PhD students at UIUC. A small number of *really* hard problems are marked with a $^\bigstar$larger star; one or two *open* problems are indicated by $\bigstar$enormous stars. Many of these exercises were contributed by my amazing teaching assistants:

> Aditya Ramani, Akash Gautam, Alex Steiger, Alina Ene, Amir Nayyeri, Asha Seetharam, Ashish Vulimiri, Ben Moseley, Brad Sturt, Brian Ensink, Chao Xu, Chris Neihengen, Connor Clark, Dan Bullok, Dan Cranston, Daniel Khashabi, David Morrison, Johnathon Fischer, Junqing Deng, Ekta Manaktala, Erin Wolf Chambers,

Gail Steitz, Gio Kao, Grant Czajkowski, Hsien-Chih Chang, Igor Gammer, John Lee, Kent Quanrud, Kevin Milans, Kevin Small, Kyle Fox, Kyle Jao, Lan Chen, Michael Bond, Mitch Harris, Naveen Arivazhagen, Nick Bachmair, Nick Hurlburt, Nirman Kumar, Nitish Korula, Rachit Agarwal, Reza Zamani-Nasab, Rishi Talreja, Rob McCann, Shripad Thite, Subhro Roy, Tana Wattanawaroon, and Yasu Furakawa.

Please do not ask me for solutions to the exercises. If you are a student, seeing the solution will rob you of the experience of solving the problem yourself, which is the only way to learn the material. If you are an instructor, you shouldn't assign problems that you can't solve yourself! (Because I don't always follow my own advice, I sometimes assign buggy problems, but I've tried to keep these out of the lecture notes themselves.)

"Johnny's" multi-colored crayon homework was found under the TA office door among the other Fall 2000 Homework 1 submissions.

Acknowledgments

All of this material draws heavily on the creativity, wisdom, and experience of thousands of algorithms students, teachers, and researchers. In particular, I am immensely grateful to the more than 2000 Illinois students who have used these notes as a primary reference, offered useful (if sometimes painful) criticism, and suffered through some truly awful first drafts. I'm also grateful for the contributions and feedback from the teaching assistants listed above. Finally, thanks to many colleagues at Illinois and elsewhere who have used these notes in their own classes and have sent helpful feedback and bug reports.

Naturally, these notes owe a great deal to the people who taught me this algorithms stuff in the first place: Bob Bixby and Michael Pearlman at Rice; David Eppstein, Dan Hirshberg, and George Lueker at UC Irvine; and Abhiram Ranade, Dick Karp, Manuel Blum, Mike Luby, and Raimund Seidel at UC Berkeley. I've also been helped tremendously by many discussions with faculty colleagues at Illinois—Cinda Heeren, Edgar Ramos, Herbert Edelsbrunner, Jason Zych, Lenny Pitt, Madhu Parasarathy, Mahesh Viswanathan, Margaret Fleck, Shang-Hua Teng, Steve LaValle, and especially Chandra Chekuri, Ed Reingold, and Sariel Har-Peled. I stole the first iteration of the overall course structure, and the idea to write up my own lecture notes, from Herbert Edelsbrunner.

The picture of the Spirit of Arithmetic from *Margarita Philosophica* at the end of the introductory notes was copied from Wikimedia Commons; the original 1508 woodcut is in the public domain. The map on the first page of the maxflow/mincut notes was copied from Lex Schrijver's excellent survey "On the history of combinatorial optimization (till 1960)"; the original map is from a US Government work in the public domain. Several of Randall Munroe's xkcd comic strips are reproduced under a Creative Commons License. One well-known frame from Allie Brosh's comic strip *Hyperbole and a Half* appears twice ***without*** permission. (Hire *all* the lawyers?)

I drew all other figures in the notes myself using OmniGraffle, except for a few older figures that I drew with (shudder) xfig. In particular, the square-Kufi rendition of the name "al-Khwārizmī" on the cover is my own.

Prerequisites

These notes assume the reader has mastered the material covered in the first two years of a strong undergraduate computer science curriculum, and that they have the intellectual maturity to recognize and repair any remaining gaps in their mastery. In particular, for most students, these notes are *not* suitable for a *first* course in data structures and algorithms. Specific prerequisites include the following:

- **Discrete mathematics:** High-school algebra, logarithm identities, naive set theory, Boolean algebra, first-order predicate logic, sets, functions, equivalences, partial orders, modular arithmetic, recursive definitions, trees (as abstract objects, not data structures), graphs.
- **Proof techniques:** direct, indirect, contradiction, exhaustive case analysis, and induction (especially "strong" and "structural" induction). Lecture 0 requires induction, and whenever Lecture $n-1$ requires induction, so does Lecture n.
- **Elementary discrete probability:** uniform vs non-uniform distributions, expectation, conditional probability, linearity of expectation, independence.
- **Iterative programming concepts:** variables, conditionals, loops, indirection (addresses/pointers/references), subroutines, recursion. I do not assume fluency in any particular programming language, but I do assume experience with at least one language that supports indirection and recursion.
- **Fundamental abstract data types:** scalars, sequences, vectors, sets, stacks, queues, priority queues, dictionaries.
- **Fundamental data structures:** arrays, linked lists (single and double, linear and circular), binary search trees, at least one *balanced* binary search tree (AVL trees, red-black trees, treaps, skip lists, splay trees, etc.), binary heaps, hash tables, and most importantly, the difference between this list and the previous list.
- **Fundamental algorithmic problems:** sorting, searching, enumeration.
- **Fundamental algorithms:** elementary arithmetic, sequential search, binary search, comparison-based sorting (selection, insertion, merge-, heap-, quick-), radix sort, pre-/post-/inorder tree traversal, breadth- and depth-first search (at least in trees), and most importantly, the difference between this list and the previous list.
- **Basic algorithm analysis:** Asymptotic notation (o, O, Θ, Ω, ω), translating loops into sums and recursive calls into recurrences, evaluating simple sums and recurrences.
- **Mathematical maturity:** facility with abstraction, formal (especially recursive) definitions, and (especially inductive) proofs; writing and following mathematical arguments; recognizing and avoiding syntactic, semantic, and/or logical nonsense.

Two notes on prerequisite material appear as an appendix to the main lecture notes: one on proofs by induction, and one on solving recurrences. The main lecture notes also *briefly* cover some prerequisite material, but more as a reminder than a good introduction. For a more thorough overview, I strongly recommend the following:

- Margaret M. Fleck. *Building Blocks for Theoretical Computer Science*, unpublished textbook, most recently revised January 2013.
- Eric Lehman, F. Thomson Leighton, and Albert R. Meyer. *Mathematics for Computer Science*, unpublished lecture notes, most recent (public) revision January 2013.

- Pat Morin. *Open Data Structures*, most recently revised June 2014 (edition 0.1G). A permanently free open-source textbook, which Pat maintains and regularly updates.

Additional References

I strongly encourage students (and other readers) *not* to restrict themselves to my notes or any other single textual reference. Authors and readers bring their own perspectives to the material; no instructor "clicks" with every student, or even every very strong student. Finding the author that most effectively gets their intuition into *your* head take some effort, but that effort pays off handsomely in the long run. The following references have been particularly valuable to me as sources of inspiration, intuition, examples, and problems.

- Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. (I used this textbook as an undergraduate at Rice and again as a masters student at UC Irvine.)
- Thomas Cormen, Charles Leiserson, Ron Rivest, and Cliff Stein. *Introduction to Algorithms*, third edition. MIT Press/McGraw-Hill, 2009. (I used the first edition as a teaching assistant at Berkeley.)
- Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2006.
- Jeff Edmonds. *How to Think about Algorithms*. Cambridge University Press, 2008.
- Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- Michael T. Goodrich and Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*. John Wiley & Sons, 2002.
- John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, first edition. Addison-Wesley, 1979. (I used this textbook as an undergraduate at Rice. Don't bother with the later editions.)
- Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2005.
- Donald Knuth. *The Art of Computer Programming*, volumes 1–4A. Addison-Wesley, 1997 and 2011. (My parents gave me the first three volumes for Christmas when I was 14, but I didn't actually read them until *much* later.)
- Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989. (I used this textbook as a teaching assistant at Berkeley.)
- Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
- Ian Parberry. *Problems on Algorithms*. Prentice-Hall, 1995 (out of print). Available from http://www.eng.unt.edu/ian/books/free/license.html after promising to make a small charitable donation. Please honor your promise.
- Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003. (Just in case you thought Knuth was the only author who could stun oxen.)
- Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 2011.

- Jeffrey O. Shallit. *A Second Course in Formal Languages and Automata Theory.* Cambridge University Press, 2008.
- Michael Sipser. *Introduction to the Theory of Computation*, third edition. Cengage Learning, 2012. Recommended if and only if you don't have to pay for it.
- Robert Endre Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.
- Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 2001.
- Class notes from my own algorithms classes at Berkeley, especially those taught by Dick Karp and Raimund Seidel.
- Lecture notes, slides, homeworks, exams, video lectures, research papers, blog posts, and full-fledged MOOCs made freely available on the web by innumerable colleagues around the world.

Caveat Lector!

Despite several rounds of revision, these notes still contain mnay mistakes, errors, bugs, gaffes, omissions, snafus, kludges, typos, mathos, grammaros, thinkos, brain farts, nonsense, garbage, cruft, junk, and outright lies, **all of which are entirely Steve Skiena's fault.** I revise and update these notes every time I teach an algorithms class, so please let me know if you find a bug. (Steve is unlikely to care.) I regularly award extra credit to students who post explanations and/or corrections of errors in the lecture notes. If I'm not teaching your class, encourage your instructor to set up a similar extra-credit scheme, and forward the bug reports to ~~Steve~~ me!

Of course, any other feedback is also welcome!

Enjoy!

— *Jeff*

It is traditional for the author to magnanimously accept the blame for whatever deficiencies remain. I don't. Any errors, deficiencies, or problems in this book are somebody else's fault, but I would appreciate knowing about them so as to determine who is to blame.

— Steven S. Skiena, *The Algorithm Design Manual* (1997)

Contents

Hinc incipit algorismus.
Haec algorismus ars praesens dicitur in qua
talibus indorum fruimur bis quinque figuris
0. 9. 8. 7. 6. 5. 4. 3. 2. 1.

— Friar Alexander de Villa Dei, *Carmen de Algorismo*, (c. 1220)

We should explain, before proceeding, that it is not our object to consider this program with reference to the actual arrangement of the data on the Variables of the engine, but simply as an abstract question of the nature *and* number *of the* operations *required to be perfomed during its complete solution.*

— Ada Augusta Byron King, Countess of Lovelace, translator's notes for Luigi F. Menabrea, "Sketch of the Analytical Engine invented by Charles Babbage, Esq." (1843)

You are right to demand that an artist engage his work consciously, but you confuse two different things: solving the problem and correctly posing the question.

— Anton Chekhov, in a letter to A. S. Suvorin (October 27, 1888)

The moment a man begins to talk about technique
that's proof that he is fresh out of ideas.

— Raymond Chandler

0 Introduction

0.1 What is an algorithm?

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions. For example, here is an algorithm for singing that annoying song "99 Bottles of Beer on the Wall", for arbitrary values of 99:

```
BottlesOfBeer(n):
  For i ← n down to 1
      Sing "i bottles of beer on the wall, i bottles of beer,"
      Sing "Take one down, pass it around, i − 1 bottles of beer on the wall."
  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, n bottles of beer on the wall."
```

The word "algorithm" does *not* derive, as algorithmophobic classicists might guess, from the Greek roots *arithmos* ($\alpha\rho\iota\theta\mu o\varsigma$), meaning "number", and *algos* ($\alpha\lambda\gamma o\varsigma$), meaning "pain". Rather, it is a corruption of the name of the 9th century Persian mathematician Abū 'Abd Allāh Muḥammad ibn Mūsā al-Khwārizmī.[1] Al-Khwārizmī is perhaps best known as the writer of the treatise *Al-Kitāb al-mukhtaṣar fīhīsāb al-abr wa'l-muqābala*[2], from which the modern word *algebra* derives. In another treatise, al-Khwārizmī popularized the modern decimal system for writing and manipulating numbers—in particular, the use of a small circle or *ṣifr* to represent a missing quantity—which had originated in India several centuries earlier. This system later became known in Europe as *algorism*, and its figures became known in English as *ciphers*.[3]

[1] 'Mohammad, father of Adbdulla, son of Moses, the Kwārizmian'. Kwārizm is an ancient city, now called Khiva, in the Khorezm Province of Uzbekistan.

[2] "The Compendious Book on Calculation by Completion and Balancing"

[3] The Italians transliterated *ṣifr* as *zefiro*, which later evolved into the modern *zero*.

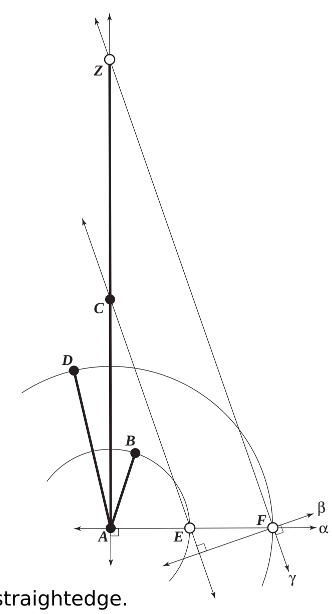
Thanks to the efforts of the medieval Italian mathematician Leonardo of Pisa, better known as Fibonacci, algorism began to replace the abacus as the preferred system of commercial calculation in Europe in the late 12th century. (Indeed, the word *calculate* derives from the Latin word *calculus*, meaning "small rock", referring to the stones on a counting board, or abacus.) Ciphers became truly ubiquitous in Western Europe only after the French revolution 600 years after Fibonacci. The more modern word *algorithm* is a false cognate with the Greek word *arithmos* ($\alpha\rho\iota\theta\mu o\varsigma$), meaning 'number' (and perhaps the previously mentioned $\alpha\lambda\gamma o\varsigma$).[4] Thus, until very recently, the word *algorithm* referred exclusively to pencil-and-paper methods for numerical calculations. People trained in the reliable execution of these methods were called—you guessed it—*computers*.[5]

0.2 A Few Simple Examples

Multiplication by compass and straightedge

Although they have only been an object of formal study for a few decades, algorithms have been with us since the dawn of civilization, for centuries before Al-Khwārizmī and Fibonacci popularized the cypher. Here is an algorithm, popularized (but almost certainly not discovered) by Euclid about 2500 years ago, for multiplying or dividing numbers using a ruler and compass. The Greek geometers represented numbers using line segments of the appropriate length. In the pseudo-code below, Circle(p, q) represents the circle centered at a point p and passing through another point q. Hopefully the other instructions are obvious.[6]

```
⟨⟨Construct the line perpendicular to ℓ and passing through P.⟩⟩
RightAngle(ℓ, P):
  Choose a point A ∈ ℓ
  A, B ← Intersect(Circle(P, A), ℓ)
  C, D ← Intersect(Circle(A, B), Circle(B, A))
  return Line(C, D)

⟨⟨Construct a point Z such that |AZ| = |AC||AD|/|AB|.⟩⟩
MultiplyOrDivide(A, B, C, D):
  α ← RightAngle(Line(A, C), A)
  E ← Intersect(Circle(A, B), α)
  F ← Intersect(Circle(A, D), α)
  β ← RightAngle(Line(E, C), F)
  γ ← RightAngle(β, F)
  return Intersect(γ, Line(A, C))
```



Multiplying or dividing using a compass and straightedge.

[4] In fact, some medieval English sources claim the Greek prefix "algo-" meant "art" or "introduction". Other sources claimed that algorithms was invented by a Greek philosopher, or a king of India, or perhaps a king of Spain, named "Algus" or "Algor" or "Argus". A few, possibly including Dante Alighieri, even identified the inventor with the mythological Greek shipbuilder and eponymous argonaut. I don't think any serious medieval scholars made the connection to the Greek work for pain, although I'm quite certain their students did.

[5] From the Latin verb *putāre*, which variously means "to trim/prune", "to clean", "to arrange", "to value", "to judge", and "to consider/suppose"; also the source of the English words "dispute", "reputation", and "amputate".

[6] Euclid and his students almost certainly drew their constructions on an *abax* ($\alpha\beta\alpha\xi$), a table covered in dust or sand (or perhaps very small rocks). Over the next several centuries, the Greek *abax* evolved into the medieval European *abacus*.

This algorithm breaks down the difficult task of multiplication into a series of simple primitive operations: drawing a line between two points, drawing a circle with a given center and boundary point, and so on. These primitive steps are quite non-trivial to execute on a modern digital computer, but this algorithm wasn't designed for a digital computer; it was designed for the Platonic Ideal Classical Greek Mathematician, wielding the Platonic Ideal Compass and the Platonic Ideal Straightedge. In this example, Euclid first defines a new primitive operation, constructing a right angle, by (as modern programmers would put it) writing a subroutine.

Multiplication by duplation and mediation

Here is an even older algorithm for multiplying large numbers, sometimes called *(Russian) peasant multiplication*. A variant of this method was copied into the Rhind papyrus by the Egyptian scribe Ahmes around 1650 BC, from a document he claimed was (then) about 350 years old. This was the most common method of calculation by Europeans before Fibonacci's introduction of Arabic numerals; it was still taught in elementary schools in Eastern Europe in the late 20th century. This algorithm was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.

```
PeasantMultiply(x, y):
  prod ← 0
  while x > 0
      if x is odd
          prod ← prod + y
      x ← ⌊x/2⌋
      y ← y + y
  return p
```

x	y		$prod$
			0
123	$+456$	$=$	456
61	$+912$	$=$	1368
30	~~1824~~		
15	$+3648$	$=$	5016
7	$+7296$	$=$	12312
3	$+14592$	$=$	26904
1	$+29184$	$=$	**56088**

The peasant multiplication algorithm breaks the difficult task of general multiplication into four simpler operations: (1) determining parity (even or odd), (2) addition, (3) duplation (doubling a number), and (4) mediation (halving a number, rounding down).[7] Of course a full specification of this algorithm requires describing how to perform those four 'primitive' operations. Peasant multiplication requires (a constant factor!) more paperwork to execute by hand, but the necessary operations are easier (for humans) to remember than the 10×10 multiplication table required by the American grade school algorithm.[8]

The correctness of peasant multiplication follows from the following recursive identity, which holds for any non-negative integers x and y:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

[7]The version of this algorithm actually used in ancient Egypt does not use mediation or parity, but it does use comparisons. To avoid halving, the algorithm pre-computes two tables by repeated doubling: one containing all the powers of 2 not exceeding x, the other containing the same powers of 2 multiplied by y. The powers of 2 that sum to x are then found by greedy subtraction, and the corresponding entries in the other table are added together to form the product.

[8]American school kids learn a variant of the *lattice* multiplication algorithm developed by Indian mathematicians and described by Fibonacci in *Liber Abaci*. The two algorithms are equivalent if the input numbers are represented in binary.

Congressional Apportionment

Here is another good example of an algorithm that comes from outside the world of computing. Article I, Section 2 of the United States Constitution requires that

> Representatives and direct Taxes shall be apportioned among the several States which may be included within this Union, according to their respective Numbers. . . . The Number of Representatives shall not exceed one for every thirty Thousand, but each State shall have at Least one Representative. . . .

Since there are a limited number of seats available in the House of Representatives, exact proportional representation is impossible without either shared or fractional representatives, neither of which are legal. As a result, several different apportionment algorithms have been proposed and used to round the fractional solution fairly. The algorithm actually used today, called *the Huntington-Hill method* or *the method of equal proportions*, was first suggested by Census Bureau statistician Joseph Hill in 1911, refined by Harvard mathematician Edward Huntington in 1920, adopted into Federal law (2 U.S.C. §§2a and 2b) in 1941, and survived a Supreme Court challenge in 1992.[9] The input array $Pop[1..n]$ stores the populations of the n states, and R is the total number of representatives. Currently, $n = 50$ and $R = 435$. The output array $Rep[1..n]$ stores the number of representatives assigned to each state.

```
ApportionCongress(Pop[1..n], R):
  PQ ← NewPriorityQueue
  for i ← 1 to n
      Rep[i] ← 1
      Insert(PQ, i, Pop[i]/√2)
      R ← R − 1

  while R > 0
      s ← ExtractMax(PQ)
      Rep[s] ← Rep[s] + 1
      Insert(PQ, s, Pop[s] / √(Rep[s](Rep[s] + 1)))
      R ← R − 1

  return Rep[1..n]
```

This pseudocode description assumes that you know how to implement a priority queue that supports the operations NewPriorityQueue, Insert, and ExtractMax. (The actual law doesn't assume that, of course.) The output of the algorithm, and therefore its correctness, does not depend *at all* on how the priority queue is implemented. The Census Bureau uses an unsorted array, stored in a column of an Excel spreadsheet; you should have learned a more efficient solution in your undergraduate data structures class.

[9] Overruling an earlier ruling by a federal district court, the Supreme Court unanimously held that ***any*** apportionment method adopted in good faith by Congress is constitutional (*United States Department of Commerce v. Montana*). The current congressional apportionment algorithm is described in gruesome detail at the U.S. Census Department web site http://www.census.gov/population/www/censusdata/apportionment/computing.html. A good history of the apportionment problem can be found at http://www.thirty-thousand.org/pages/Apportionment.htm. A report by the Congressional Research Service describing various apportionment methods is available at http://www.rules.house.gov/archives/RL31074.pdf.

A bad example

As a prototypical example of a sequence of instructions that is not actually an algorithm, consider "Martin's algorithm":[10]

```
BECOMEAMILLIONAIREANDNEVERPAYTAXES:
  Get a million dollars.
  If the tax man comes to the door and says, "You have never paid taxes!"
      Say "I forgot."
```

Pretty simple, except for that first step; it's a doozy. A group of billionaire CEOs might consider this an algorithm, since for them the first step is both unambiguous and trivial, but for the rest of us poor slobs, Martin's procedure is too vague to be considered an actual algorithm. On the other hand, this is a perfect example of a ***reduction***—it *reduces* the problem of being a millionaire and never paying taxes to the 'easier' problem of acquiring a million dollars. We'll see reductions over and over again in this class. As hundreds of businessmen and politicians have demonstrated, if you know how to solve the easier problem, a reduction tells you how to solve the harder one.

Martin's algorithm, like many of our previous examples, is not the kind of algorithm that computer scientists are used to thinking about, because it is phrased in terms of operations that are difficult for computers to perform. In this class, we'll focus (almost!) exclusively on algorithms that can be reasonably implemented on a standard digital computer. In other words, each step in the algorithm must be something that either is directly supported by common programming languages (such as arithmetic, assignments, loops, or recursion) or is something that you've already learned how to do in an earlier class (like sorting, binary search, or depth first search).

0.3 Writing down algorithms

Computer programs are concrete representations of algorithms, but algorithms are *not* programs; they should not be described in a particular programming language. The whole *point* of this course is to develop computational techniques that can be used in *any* programming language. The idiosyncratic syntactic details of C, C++, C#, Java, Python, Ruby, Erlang, Haskell, OcaML, Scheme, Scala, Clojure, Visual Basic, Smalltalk, Javascript, Processing, Squeak, Forth, TeX, Fortran, COBOL, INTERCAL, MMIX, LOLCODE, Befunge, Parseltongue, Whitespace, or Brainfuck are of little or no importance in algorithm design, and focusing on them will only distract you from what's *really* going on.[11] What we really want is closer to what you'd write in the *comments* of a real program than the code itself.

On the other hand, a plain English prose description is usually not a good idea either. Algorithms have lots of structure—especially conditionals, loops, and recursion—that are far too easily hidden by unstructured prose. Natural languages like English are full of ambiguities, subtleties,

[10]Steve Martin, "You Can Be A Millionaire", Saturday Night Live, January 21, 1978. Also appears on *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

[11]This is, of course, a matter of religious conviction. Linguists argue incessantly over the *Sapir-Whorf hypothesis*, which states (more or less) that people think only in the categories imposed by their languages. According to an extreme formulation of this principle, some concepts in one language simply cannot be understood by speakers of other languages, not just because of technological advancement—How would you translate 'jump the shark' or 'blog' into Aramaic?—but because of inherent structural differences between languages and cultures. For a more skeptical view, see Steven Pinker's *The Language Instinct*. There is admittedly some strength to this idea when applied to different programming paradigms. (What's the Y combinator, again? How do templates work? What's an Abstract Factory?) Fortunately, those differences are generally too subtle to have much impact in *this* class. For a compelling counterexample, see Chris Okasaki's thesis/monograph *Functional Data Structures* and its more recent descendants.

and shades of meaning, but algorithms must be described as precisely and unambiguously as possible. Finally and more seriously, in non-technical writing, there is natural tendency to describe repeated operations informally: "Do this first, then do this second, and so on." But as anyone who has taken one of those 'What comes next in this sequence?' tests already knows, specifying what happens in the first few iterations of a loop says very little, of anything, about what happens later iterations. To make the description unambiguous, we must explicitly specify the behavior of *every* iteration. The stupid joke about the programmer dying in the shower has a grain of truth—"Lather, rinse, repeat" *is* ambiguous; what exactly do we repeat, and until when?

In my opinion, the clearest way to present an algorithm is using pseudocode. Pseudocode uses the *structure* of formal programming languages and mathematics to break algorithms into primitive steps; but the primitive steps themselves may be written using mathematics, pure English, or an appropriate mixture of the two. Well-written pseudocode reveals the internal structure of the algorithm but hides irrelevant implementation details, making the algorithm much easier to understand, analyze, debug, and implement.

The precise syntax of pseudocode is a personal choice, but the overriding goal should be clarity and precision. Ideally, pseudocode should allow any competent programmer to implement the underlying algorithm, quickly and correctly, in *their* favorite programming language, *without understanding why the algorithm works*. Here are the guidelines I follow and strongly recommend:

- Be consistent!
- Use standard imperative programming keywords (if/then/else, while, for, repeat/until, case, return) and notation ($variable \leftarrow value$, $Array[index]$, $function(argument)$, $bigger > smaller$, etc.). Keywords should be standard English words: write 'else if' instead of 'elif'.
- Indent everything carefully and consistently; the block structure should be visible from across the room. This rule is especially important for nested loops and conditionals. *Don't* add unnecessary syntactic sugar like braces or begin/end tags; careful indentation is almost always enough.
- Use mnemonic algorithm and variable names. Short variable names are good, but readability is more important than concision; except for idioms like loop indices, short but complete words are better than single letters. Absolutely *never* use pronouns!
- Use standard mathematical notation for standard mathematical things. For example, write $x \cdot y$ instead of $x * y$ for multiplication; write $x \bmod y$ instead of $x \% y$ for remainder; write $\sqrt{x}$ instead of $sqrt(x)$ for square roots; write a^b instead of $power(a, b)$ for exponentiation; and write ϕ instead of *phi* for the golden ratio.
- Avoid mathematical notation if English is clearer. For example, 'Insert a into X' may be preferable to $\textsc{Insert}(X, a)$ or $X \leftarrow X \cup \{a\}$.
- Each statement should fit on one line, and each line should contain either exactly one statement or exactly one structuring element (for, while, if). (I sometimes make an exception for short and similar statements like $i \leftarrow i + 1$; $j \leftarrow j - 1$; $k \leftarrow 0$.)
- *Don't* use a fixed-width typeface to typeset pseudocode; it's much harder to read than normal typeset text. Similarly, *don't* typeset keywords like 'for' or 'while' in a different **style**; the syntactic sugar is not what you want the reader to look at. On the other hand, I do use *italics* for variables (following the standard mathematical typesetting convention), SMALL CAPS for algorithms and constants, and *a different typeface* for literal strings.

0.4 Analyzing algorithms

It's not enough just to write down an algorithm and say 'Behold!' We must also convince our audience (and ourselves!) that the algorithm actually does what it's supposed to do, and that it does so efficiently.

Correctness

In some application settings, it is acceptable for programs to behave correctly most of the time, on all 'reasonable' inputs. Not in this class; we require algorithms that are correct for *all possible* inputs. Moreover, we must *prove* that our algorithms are correct; trusting our instincts, or trying a few test cases, isn't good enough. Sometimes correctness is fairly obvious, especially for algorithms you've seen in earlier courses. On the other hand, 'obvious' is all too often a synonym for 'wrong'. Many of the algorithms we will discuss in this course will require extra work to prove correct. Correctness proofs almost always involve induction. We *like* induction. Induction is our *friend*.[12]

But before we can formally prove that our algorithm does what it's supposed to do, we have to formally *state* what it's supposed to do! Algorithmic problems are usually presented using standard English, in terms of real-world objects, not in terms of formal mathematical objects. It's up to us, the algorithm designers, to restate these problems in terms of mathematical objects that we can prove things about—numbers, arrays, lists, graphs, trees, and so on. We must also determine if the problem statement carries any hidden assumptions, and state those assumptions explicitly. (For example, in the song "n Bottles of Beer on the Wall", n is always a positive integer.) Restating the problem formally is not only required for proofs; it is also one of the best ways to really understand what a problem is asking for. The hardest part of answering any question is figuring out the right way to ask it!

It is important to remember the distinction between a problem and an algorithm. A problem is a task to perform, like "Compute the square root of x" or "Sort these n numbers" or "Keep n algorithms students awake for t minutes". An algorithm is a set of instructions for accomplishing such a task. The same problem may have hundreds of different algorithms; the same algorithm may solve hundreds of different problems.

Running time

The most common way of ranking different algorithms for the same problem is by how quickly they run. Ideally, we want the fastest possible algorithm for any particular problem. In many application settings, it is acceptable for programs to run efficiently most of the time, on all 'reasonable' inputs. Not in this class; we require algorithms that *always* run efficiently, even in the worst case.

But how do we measure running time? As a specific example, how long does it take to sing the song BottlesOfBeer(n)? This is obviously a function of the input value n, but it also depends on how quickly you can sing. Some singers might take ten seconds to sing a verse; others might take twenty. Technology widens the possibilities even further. Dictating the song over a telegraph using Morse code might take a full minute per verse. Downloading an mp3 over the Web might take a tenth of a second per verse. Duplicating the mp3 in a computer's main memory might take only a few microseconds per verse.

[12]If induction is *not* your friend, you will have a hard time in this course.

What's important here is how the singing time changes as n grows. Singing BOTTLESOFBEER$(2n)$ takes about twice as long as singing BOTTLESOFBEER(n), no matter what technology is being used. This is reflected in the asymptotic singing time $\Theta(n)$. We can measure time by counting how many times the algorithm executes a certain instruction or reaches a certain milestone in the 'code'. For example, we might notice that the word 'beer' is sung three times in every verse of BOTTLESOFBEER, so the number of times you sing 'beer' is a good indication of the total singing time. For this question, we can give an exact answer: BOTTLESOFBEER(n) uses exactly $3n+3$ beers.

There are plenty of other songs that have non-trivial singing time. This one is probably familiar to most English-speakers:

```
NDAYSOFCHRISTMAS(gifts[2..n]):
  for i ← 1 to n
      Sing "On the ith day of Christmas, my true love gave to me"
      for j ← i down to 2
          Sing "j gifts[j]"
      if i > 1
          Sing "and"
      Sing "a partridge in a pear tree."
```

The input to NDAYSOFCHRISTMAS is a list of $n-1$ gifts. It's quite easy to show that the singing time is $\Theta(n^2)$; in particular, the singer mentions the name of a gift $\sum_{i=1}^{n} i = n(n+1)/2$ times (counting the partridge in the pear tree). It's also easy to see that during the first n days of Christmas, my true love gave to me exactly $\sum_{i=1}^{n}\sum_{j=1}^{i} j = n(n+1)(n+2)/6 = \Theta(n^3)$ gifts.

There are many other traditional songs that take quadratic time to sing; examples include "Old MacDonald Had a Farm", "There Was an Old Lady Who Swallowed a Fly", "The House that Jack Built", "Hole in the Bottom of the Sea", "Green Grow the Rushes O", "The Rattlin' Bog", "The Barley-Mow", "Eh, Cumpari!", "Alouette", "Echad Mi Yode'a", "Ist das nicht ein Schnitzelbank?", and "Minkurinn í hænsnakofanum". For further details, consult your nearest preschooler.

```
OLDMACDONALD(animals[1..n], noise[1..n]):
  for i ← 1 to n
      Sing "Old MacDonald had a farm, E I E I O"
      Sing "And on this farm he had some animals[i], E I E I O"
      Sing "With a noise[i] noise[i] here, and a noise[i] noise[i] there"
      Sing "Here a noise[i], there a noise[i], everywhere a noise[i] noise[i]"
      for j ← i−1 down to 1
          Sing "noise[j] noise[j] here, noise[j] noise[j] there"
          Sing "Here a noise[j], there a noise[j], everywhere a noise[j] noise[j]"
      Sing "Old MacDonald had a farm, E I E I O."
```

```
ALOUETTE(lapart[1..n]):
  Chantez « Alouette, gentille alouette, alouette, je te plumerais. »
  pour tout i de 1 á n
      Chantez « Je te plumerais lapart[i]. Je te plumerais lapart[i]. »
      pour tout j de i−1 á bas á 1
          Chantez « Et lapart[j] ! Et lapart[j] ! »
      Chantez « Ooooooo! »
      Chantez « Alouette, gentille alluette, alouette, je te plumerais. »
```

A more modern example of the parametrized cumulative song is "The TELNET Song" by Guy Steele, which takes $O(2^n)$ time to sing; Steele recommended $n=4$.

For a slightly less facetious example, consider the algorithm ApportionCongress. Here the running time obviously depends on the implementation of the priority queue operations, but we can certainly bound the running time as $O(N + RI + (R - n)E)$, where N denotes the running time of NewPriorityQueue, I denotes the running time of Insert, and E denotes the running time of ExtractMax. Under the reasonable assumption that $R > 2n$ (on average, each state gets at least two representatives), we can simplify the bound to $O(N + R(I + E))$. The Census Bureau implements the priority queue using an unsorted array of size n; this implementation gives us $N = I = \Theta(1)$ and $E = \Theta(n)$, so the overall running time is $O(Rn)$. This is good enough for government work, but we can do better. Implementing the priority queue using a binary heap (or a heap-ordered array) gives us $N = \Theta(1)$ and $I = R = O(\log n)$, which implies an overall running time of $O(R \log n)$.

Sometimes we are also interested in other computational resources: space, randomness, page faults, inter-process messages, and so forth. We can use the same techniques to analyze those resources as we use to analyze running time.

0.5 A Longer Example: Stable Matching

Every year, thousands of new doctors must obtain internships at hospitals around the United States. During the first half of the 20th century, competition among hospitals for the best doctors led to earlier and earlier offers of internships, sometimes as early as the second year of medical school, along with tighter deadlines for acceptance. In the 1940s, medical schools agreed not to release information until a common date during their students' fourth year. In response, hospitals began demanding faster decisions. By 1950, hospitals would regularly call doctors, offer them internships, and demand *immediate* responses. Interns were forced to gamble if their third-choice hospital called first—accept and risk losing a better opportunity later, or reject and risk having no position at all.[13]

Finally, a central clearinghouse for internship assignments, now called the National Resident Matching Program, was established in the early 1950s. Each year, doctors submit a ranked list of all hospitals where they would accept an internship, and each hospital submits a ranked list of doctors they would accept as interns. The NRMP then computes an assignment of interns to hospitals that satisfies the following *stability* requirement. For simplicity, let's assume that there are n doctors and n hospitals; each hospital offers exactly one internship; each doctor ranks all hospitals and vice versa; and finally, there are no ties in the doctors' and hospitals' rankings.[14] We say that a matching of doctors to hospitals is ***unstable*** if there are two doctors α and β and two hospitals A and B, such that

- α is assigned to A, and β is assigned to B;
- α prefers B to A, and B prefers α to β.

In other words, α and B would both be happier with each other than with their current assignment. The goal of the Resident Match is a ***stable matching***, in which no doctor or hospital has an incentive to cheat the system. At first glance, it is not clear that a stable matching exists!

In 1952, the NRMP adopted the "Boston Pool" algorithm to assign interns, so named because it had been previously used by a regional clearinghouse in the Boston area. The algorithm is

[13]The academic job market involves similar gambles, at least in computer science. Some departments start making offers in February with two-week decision deadlines; other departments don't even start interviewing until late March; MIT notoriously waits until May, when all its interviews are over, before making *any* faculty offers.

[14]In reality, most hospitals offer multiple internships, each doctor ranks only a subset of the hospitals and vice versa, and there are typically more internships than interested doctors. And then it starts getting complicated.

often misattributed to David Gale and Lloyd Shapley, who formally analyzed the algorithm and first proved that it computes a stable matching in 1962; Gale and Shapley used the metaphor of college admissions.[15] Similar algorithms have since been adopted for other matching markets, including faculty recruiting in France, university admission in Germany, public school admission in New York and Boston, billet assignments for US Navy sailors, and kidney-matching programs. Shapley was awarded the 2012 Nobel Prize in Economics for his research on stable matching, together with Alvin Roth, who significantly extended Shapley's work and used it to develop several real-world exchanges.

The Boston Pool algorithm proceeds in rounds until every position has been filled. Each round has two stages:

1. An arbitrary unassigned hospital A offers its position to the best doctor α (according to the hospital's preference list) who has not already rejected it.

2. Each doctor ultimately accepts the best offer that she receives, according to her preference list. Thus, if α is currently unassigned, she (tentatively) accepts the offer from A. If α already has an assignment but prefers A, she rejects her existing assignment and (tentatively) accepts the new offer from A. Otherwise, α rejects the new offer.

For example, suppose four doctors (Dr. Quincy, Dr. Rotwang, Dr. Shephard, and Dr. Tam, represented by lower-case letters) and four hospitals (Arkham Asylum, Bethlem Royal Hospital, County General Hospital, and The Dharma Initiative, represented by upper-case letters) rank each other as follows:

q	r	s	t
A	A	B	D
B	D	A	B
C	C	C	C
D	B	D	A

A	B	C	D
t	r	t	s
s	t	r	r
r	q	s	q
q	s	q	t

Given these preferences as input, the Boston Pool algorithm might proceed as follows:

1. Arkham makes an offer to Dr. Tam.
2. Bedlam makes an offer to Dr. Rotwang.
3. County makes an offer to Dr. Tam, who rejects her earlier offer from Arkham.
4. Dharma makes an offer to Dr. Shephard. (From this point on, because there is only one unmatched hospital, the algorithm has no more choices.)
5. Arkham makes an offer to Dr. Shephard, who rejects her earlier offer from Dharma.
6. Dharma makes an offer to Dr. Rotwang, who rejects her earlier offer from Bedlam.
7. Bedlam makes an offer to Dr. Tam, who rejects her earlier offer from County.
8. County makes an offer to Dr. Rotwang, who rejects it.

[15]The "Gale-Shapley algorithm" is a prime instance of ***Stigler's Law of Eponymy:*** *No scientific discovery is named after its original discoverer.* In his 1980 paper that gives the law its name, the statistician Stephen Stigler claimed that this law was first proposed by sociologist Robert K. Merton. However, similar statements were previously made by Vladimir Arnol'd in the 1970's ("Discoveries are rarely attributed to the correct person."), Carl Boyer in 1968 ("Clio, the muse of history, often is fickle in attaching names to theorems!"), Alfred North Whitehead in 1917 ("Everything of importance has been said before by someone who did not discover it."), and even Stephen's father George Stigler in 1966 ("If we should ever encounter a case where a theory is named for the correct man, it will be noted."). We will see *many* other examples of Stigler's law in this class.

9. County makes an offer to Dr. Shephard, who rejects it.
10. County makes an offer to Dr. Quincy.

At this point, all pending offers are accepted, and the algorithm terminates with a matching: (A, s), (B, t), (C, q), (D, r). You can (and should) verify by brute force that this matching is stable, even though no doctor was hired by her favorite hospital, and no hospital hired its favorite doctor; in fact, County was forced to hire their *least* favorite doctor. This is not the only stable matching for this list of preferences; the matching $(A, r), (B, s), (C, q), (D, t)$ is also stable.

Running Time

Analyzing the algorithm's running time is relatively straightforward. Each hospital makes an offer to each doctor at most once, so the algorithm requires at most n^2 rounds. In an actual implementation, each doctor and hospital can be identified by a unique integer between 1 and n, and the preference lists can be represented as two arrays $\mathit{DocPref}[1..n][1..n]$ and $\mathit{HosPref}[1..n][1..n]$, where $\mathit{DocPref}[\alpha][r]$ represents the rth hospital in doctor α's preference list, and $\mathit{HosPref}[A][r]$ represents the rth doctor in hospital A's preference list. With the input in this form, the Boston Pool algorithm can be implemented to run in ***$O(n^2)$ time***; we leave the details as an easy exercise.

A somewhat harder exercise is to prove that there are inputs (and choices of who makes offers when) that force $\Omega(n^2)$ rounds before the algorithm terminates. Thus, the $O(n^2)$ upper bound on the worst-case running time cannot be improved; in this case, we say our analysis is ***tight***.

Correctness

But why is the algorithm *correct*? How do we know that the Boston Pool algorithm always computes a *stable matching*? Gale and Shapley proved correctness as follows. The algorithm continues as long as there is at least one unfilled position; conversely, when the algorithm terminates (after at most n^2 rounds), every position is filled. No doctor can accept more than one position, and no hospital can hire more than one doctor. Thus, the algorithm always computes a matching; it remains only to prove that the matching is stable.

Suppose doctor α is assigned to hospital A in the final matching, but prefers B. Because every doctor accepts the best offer she receives, α received no offer she liked more than A. In particular, B never made an offer to α. On the other hand, B made offers to every doctor they like more than β. Thus, B prefers β to α, and so there is no instability.

Surprisingly, the correctness of the algorithm does not depend on which hospital makes its offer in which round. In fact, there is a stronger sense in which the order of offers doesn't matter—no matter which unassigned hospital makes an offer in each round, *the algorithm always computes the same matching*! Let's say that α is a ***feasible*** doctor for A if there is a stable matching that assigns doctor α to hospital A.

Lemma 0.1. *During the Boston Pool algorithm, each hospital A is rejected only by doctors that are infeasible for A.*

Proof: We prove the lemma by induction. Consider an arbitrary round of the Boston Pool algorithm, in which doctor α rejects one hospital A for another hospital B. The rejection implies that α prefers B to A. Every doctor that appears higher than α in B's preference list has already rejected B and therefore, by the inductive hypothesis, is infeasible for B.

Now consider an arbitrary matching that assigns α to A. We already established that α prefers B to A. If B prefers α to its partner, the matching is unstable. On the other hand, if B prefers its partner to α, then (by our earlier argument) its partner is infeasible, and again the matching is unstable. We conclude that there is no stable matching that assigns α to A. $\square$

Now let ***best*(A)** denote the highest-ranked *feasible* doctor on A's preference list. Lemma 0.1 implies that every doctor that A prefers to its final assignment is infeasible for A. On the other hand, the final matching is stable, so the doctor assigned to A is feasible for A. The following result is now immediate:

Corollary 0.2. *The Boston Pool algorithm assigns* best(A) *to A, for every hospital A.*

Thus, from the hospitals' point of view, the Boston Pool algorithm computes the best possible stable matching. It turns out that this matching is also the *worst* possible from the doctors' viewpoint! Let **worst(α)** denote the lowest-ranked feasible hospital on doctor α's preference list.

Corollary 0.3. *The Boston Pool algorithm assigns α to* worst(α)*, for every doctor α.*

Proof: Suppose the Boston Pool algorithm assigns doctor α to hospital A; we need to show that $A = \mathit{worst}(\alpha)$. Consider an arbitrary stable matching where A is *not* matched with α but with another doctor β. The previous corollary implies that A prefers $\alpha = \mathit{best}(A)$ to β. Because the matching is stable, α must therefore prefer her assigned hopital to A. This argument works for *any* stable assignment, so α prefers *every* other feasible match to A; in other words, $A = \mathit{worst}(\alpha)$. $\square$

A subtle consequence of these two corollaries, discovered by Dubins and Freeman in 1981, is that a doctor can potentially improve her assignment by lying about her preferences, but a hospital cannot. (However, a set of hospitals can collude so that *some* of their assignments improve.) Partly for this reason, the National Residency Matching Program reversed its matching algorithm in 1998, so that potential residents offer to work for hospitals in preference order, and each hospital accepts its best offer. Thus, the new algorithm computes the best possible stable matching for the doctors, and the worst possible stable matching for the hospitals. In practice, however, this modification affected less than 1% of the resident's assignments. As far as I know, the precise effect of this change on the *patients* is an open problem.

0.6 Why are we here, anyway?

This class is ultimately about learning two skills that are crucial for all computer scientists.

1. ***Intuition:*** How to *think* about abstract computation.
2. ***Language:*** How to *talk* about abstract computation.

The first goal of this course is to help you develop algorithmic *intuition*. How do various algorithms really work? When you see a problem for the first time, how should you attack it? How do you tell which techniques will work at all, and which ones will work best? How do you judge whether one algorithm is better than another? How do you tell whether you have the best possible solution? These are *not* easy questions; anyone who says differently is selling something.

Our second main goal is to help you develop algorithmic *language*. It's not enough just to understand how to solve a problem; you also have to be able to explain your solution to somebody else. I don't mean just how to turn your algorithms into working code—despite what many students (and inexperienced programmers) think, 'somebody else' is *not* just a computer. Nobody programs alone. Code is read far more often than it is written, or even compiled. Perhaps more importantly in the short term, explaining something to somebody else is one of the best ways to clarify your own understanding. As Albert Einstein (or was it Richard Feynman?) apocryphally put it, "You do not really understand something unless you can explain it to your grandmother."

Along the way, you'll pick up a bunch of algorithmic facts—mergesort runs in $\Theta(n \log n)$ time; the amortized time to search in a splay tree is $O(\log n)$; greedy algorithms usually don't produce optimal solutions; the traveling salesman problem is NP-hard—but these aren't the point of the course. You can always look up mere facts in a textbook or on the web, provided you have enough intuition and experience to know what to look for. That's why we let you bring cheat sheets to the exams; we don't want you wasting your study time trying to memorize all the facts you've seen.

You'll also practice a lot of algorithm design and analysis skills—finding useful examples and counterexamples, developing induction proofs, solving recurrences, using big-Oh notation, using probability, giving problems crisp mathematical descriptions, and so on. These skills are *incredibly* useful, and it's impossible to develop good intuition and good communication skills without them, but they aren't the main point of the course either. At this point in your educational career, you should be able to pick up most of those skills on your own, once you know what you're trying to do.

Unfortunately, there is no systematic procedure—no algorithm—to determine which algorithmic techniques are most effective at solving a given problem, or finding good ways to explain, analyze, optimize, or implement a given algorithm. Like many other activities (music, writing, juggling, acting, martial arts, sports, cooking, programming, teaching, etc.), the *only* way to master these skills is to make them your own, through practice, practice, and more practice. You can only develop good problem-solving skills by solving problems. You can only develop good communication skills by communicating. Good intuition is the product of experience, not its replacement. We *can't* teach you how to do well in this class. All we can do (and what we will do) is lay out some fundamental tools, show you how to use them, create opportunities for you to practice with them, and give you honest feedback, based on our own hard-won experience and intuition. The rest is up to you.

Good algorithms are extremely useful, elegant, surprising, deep, even beautiful, but most importantly, algorithms are *fun*! I hope you will enjoy playing with them as much as I do.

Boethius the algorist versus Pythagoras the abacist.
from *Margarita Philosophica* by Gregor Reisch (1503)

Exercises

0. Describe and analyze an efficient algorithm that determines, given a legal arrangement of standard pieces on a standard chess board, which player will win at chess from the given starting position if both players play perfectly. *[Hint: There is a trivial one-line solution!]*

1. "The Barley Mow" is a cumulative drinking song which has been sung throughout the British Isles for centuries. (An early version entitled "Giue vs once a drinke" appears in Thomas Ravenscroft's song collection *Deuteromelia*, which was published in 1609, but the song is almost certainly much older.) The song has many variants, but one version traditionally sung in Devon and Cornwall has the following pseudolyrics, where *vessel*$[i]$ is the name of a vessel that holds 2^i ounces of beer. The traditional song uses the following vessels: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. (Every vessel in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.)

```
BarleyMow(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for i ← 1 to n
      "We'll drink it out of the vessel[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The vessel[j],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

   (a) Suppose each name *vessel*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

   (b) If you want to sing this song for arbitrarily large values of n, you'll have to make up your own vessel names. To avoid repetition, these names must become progressively longer as n increases. ("We'll drink it out of the hemisemidemiyottapint, boys!") Suppose *vessel*$[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

   (c) Suppose each time you mention the name of a vessel, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *vessel*$[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BarleyMow(n)? (Give an *exact* answer, not just an asymptotic bound.)

2. Describe and analyze the Boston Pool stable matching algorithm in more detail, so that the worst-case running time is $O(n^2)$, as claimed earlier in the notes.

3. Prove that it is possible for the Boston Pool algorithm to execute $\Omega(n^2)$ rounds. (You need to describe both a suitable input and a sequence of $\Omega(n^2)$ valid proposals.)

4. Describe and analyze an efficient algorithm to determine whether a given set of hospital and doctor preferences has to a *unique* stable matching.

5. Consider a generalization of the stable matching problem, where some doctors do not rank all hospitals and some hospitals do not rank all doctors, and a doctor can be assigned to a hospital only if each appears in the other's preference list. In this case, there are three additional unstable situations:

   - A hospital prefers an unmatched doctor to its assigned match.
   - A doctor prefers an unmatched hospital to her assigned match.
   - An unmatched doctor and an unmatched hospital appear in each other's preference lists.

   Describe and analyze an efficient algorithm that computes a stable matching in this setting.

   Note that a stable matching may leave some doctors and hospitals unmatched, even though their preference lists are non-empty. For example, if every doctor lists Harvard as their only acceptable hospital, and every hospital lists Dr. House as their only acceptable intern, then only House and Harvard will be matched.

6. Recall that the input to the Huntington-Hill apportionment algorithm ApportionCongress is an array $P[1..n]$, where $P[i]$ is the population of the ith state, and an integer R, the total number of representatives to be allotted. The output is an array $r[1..n]$, where $r[i]$ is the number of representatives allotted to the ith state by the algorithm.

   Let $P = \sum_{i=1}^{n} P[i]$ denote the total population of the country, and let $r_i^* = R \cdot P[i]/P$ denote the ideal number of representatives for the ith state.

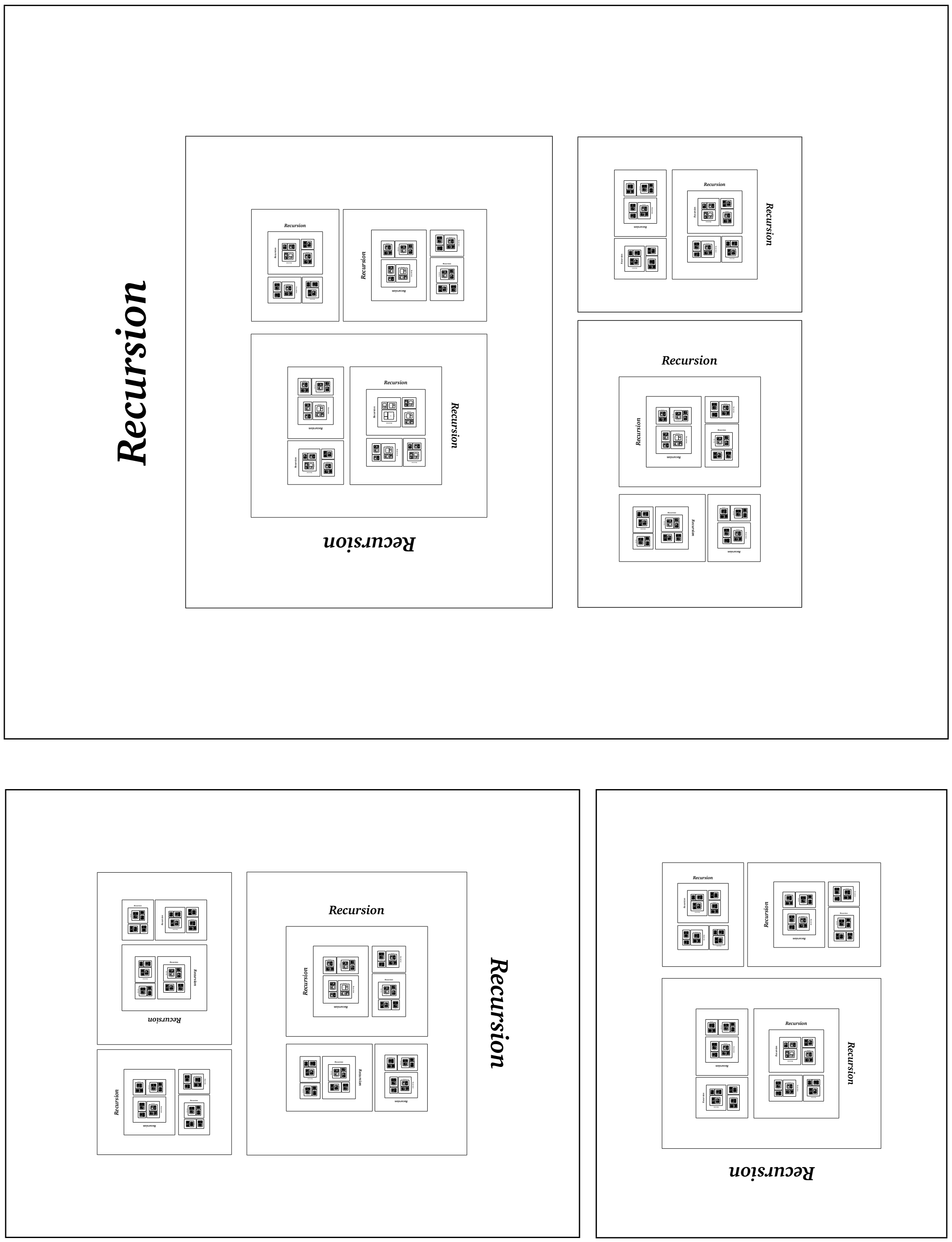
   (a) Prove that $r[i] \geq \lfloor r_i^* \rfloor$ for all i.

   (b) Describe and analyze an algorithm that computes exactly the same congressional apportionment as ApportionCongress in $O(n \log n)$ time. (Recall that the running time of ApportionCongress depends on R, which could be arbitrarily larger than n.)

   *(c) If a state's population is small relative to the other states, its ideal number r_i^* of representatives could be close to zero; thus, tiny states are over-represented by the Huntington-Hill apportionment process. Surprisingly, this can also be true of very large states. Let $\alpha = (1+\sqrt{2})/2 \approx 1.20710678119$. Prove that for any $\varepsilon > 0$, there is an input to ApportionCongress with $\max_i P[i] = P[1]$, such that $r[1] > (\alpha - \varepsilon)\, r_1^*$.

   ★(d) Can you improve the constant α in the previous question?

Recursion



The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers.
— Sun Zi, *The Art of War* (c. 400 C.E.), translated by Lionel Giles (1910)

Our life is frittered away by detail. . . . Simplify, simplify.
— Henry David Thoreau, *Walden* (1854)

Nothing is particularly hard if you divide it into small jobs.
— Henry Ford

Do the hard jobs first. The easy jobs will take care of themselves.
— Dale Carnegie

1 Recursion

1.1 Reductions

Reduction is the single most common technique used in designing algorithms. Reducing one problem X to another problem Y means to write an algorithm for X that uses an algorithm for Y as a black box or subroutine. Crucially, the correctness of the resulting algorithm cannot depend in any way on *how* the algorithm for Y works. The only thing we can assume is that the black box solves Y correctly. The inner workings of the black box are simply *none of our business*; they're somebody else's problem. It's often best to literally think of the black box as functioning by magic.

For example, the Huntington-Hill algorithm described in Lecture 0 reduces the problem of apportioning Congress to the problem of maintaining a priority queue that supports the operations Insert and ExtractMax. The abstract data type "priority queue" is a black box; the correctness of the apportionment algorithm does not depend on any specific priority queue data structure. Of course, the *running time* of the apportionment algorithm depends on the *running time* of the Insert and ExtractMax algorithms, but that's a separate issue from the *correctness* of the algorithm. The beauty of the reduction is that we can create a more efficient apportionment algorithm by simply swapping in a new priority queue data structure. Moreover, the designer of that data structure does not need to know or care that it will be used to apportion Congress.

Similarly, if we want to design an algorithm to compute the smallest deterministic finite-state machine equivalent to a given regular expression, we don't have to start from scratch. Instead we can reduce the problem to three subproblems for which algorithms can be found in earlier lecture notes: (1) build an NFA from the regular expression, using either Thompson's algorithm or Glushkov's algorithm; (2) transform the NFA into an equivalent DFA, using the (incremental) subset construction; and (3) transform the DFA into the smallest equivalent DFA, using Moore's algorithm, for example. Even if your class skipped over the automata notes, merely knowing that those component algorithms exist (Trust me!) allows you to combine them into more complex algorithms; you don't *need* to know the details. (But you should, because they're totally cool. Trust me!) Again swapping in a more efficient algorithm for any of those three subproblems automatically yields a more efficient algorithm for the problem as a whole.

When we design algorithms, we may not know exactly how the basic building blocks we use are implemented, or how our algorithms might be used as building blocks to solve even bigger problems. Even when you do know precisely how your components work, it is often *extremely* useful to pretend that you don't. (Trust *yourself*!)

1.2 Simplify and Delegate

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem is small or simple enough, just solve it.
- Otherwise, reduce the problem to one or more ***simpler instances of the same problem***.

If the self-reference is confusing, it's helpful to imagine that someone else is going to solve the simpler problems, just as you would assume for other types of reductions. I like to call that someone else the Recursion Fairy. Your *only* task is to *simplify* the original problem, or to solve it directly when simplification is either unnecessary or impossible; the Recursion Fairy will magically take care of all the simpler subproblems for you, using Methods That Are None Of Your Business So *Butt Out*.[1] Mathematically sophisticated readers might recognize the Recursion Fairy by its more formal name, the Induction Hypothesis.

There is one mild technical condition that must be satisfied in order for any recursive method to work correctly: There must be no infinite sequence of reductions to 'simpler' and 'simpler' subproblems. Eventually, the recursive reductions must stop with an elementary *base case* that can be solved by some other method; otherwise, the recursive algorithm will loop forever. This finiteness condition is almost always satisfied trivially, but we should always be wary of "obvious" recursive algorithms that actually recurse forever. (All too often, "obvious" is a synonym for "false".)

1.3 Tower of Hanoi

The Tower of Hanoi puzzle was first published by the mathematician François Éduoard Anatole Lucas in 1883, under the pseudonym "N. Claus (de Siam)" (an anagram of "Lucas d'Amiens"). The following year, Henri de Parville described the puzzle with the following remarkable story:[2]

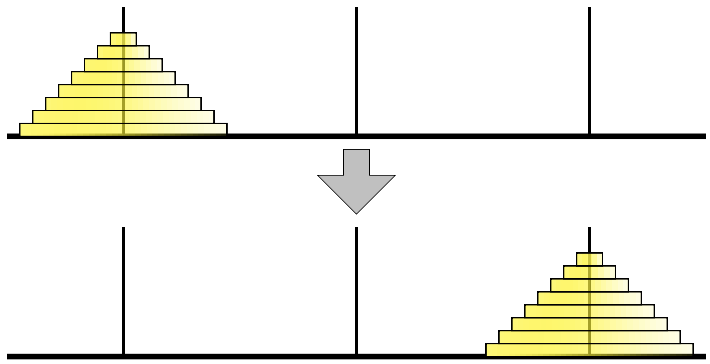
> *In the great temple at Benares beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.*

Of course, as good computer scientists, our first instinct on reading this story is to substitute the variable n for the hardwired constant 64. And following standard practice (since most physical instances of the puzzle are made of wood instead of diamonds and gold), we will refer to the three possible locations for the disks as "pegs" instead of "needles". How can we move a tower of n disks from one peg to another, using a third peg as an occasional placeholder, without ever placing a disk on top of a smaller disk?

The trick to solving this puzzle is to think recursively. Instead of trying to solve the entire puzzle all at once, let's concentrate on moving just the largest disk. We can't move it at the

[1] When I was a student, I used to attribute recursion to "elves" instead of the Recursion Fairy, referring to the Brothers Grimm story about an old shoemaker who leaves his work unfinished when he goes to bed, only to discover upon waking that elves ("Wichtelmänner") have finished everything overnight. Someone more entheogenically experienced than I might recognize them as Terence McKenna's "self-transforming machine elves".

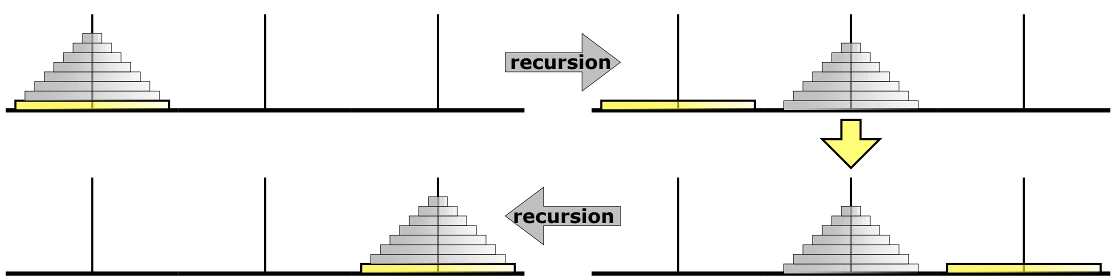
[2] This English translation is from W. W. Rouse Ball and H. S. M. Coxeter's book *Mathematical Recreations and Essays*.



The Tower of Hanoi puzzle

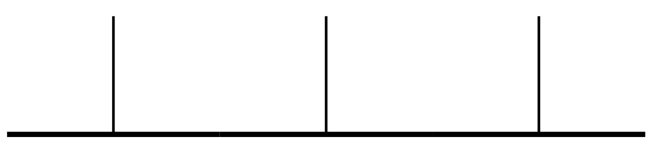
beginning, because all the other disks are covering it; we have to move those $n-1$ disks to the third peg before we can move the nth disk. And then after we move the nth disk, we have to move those $n-1$ disks back on top of it. So now all we have to figure out is how to...

STOP!! That's it! We're done! We've successfully reduced the n-disk Tower of Hanoi problem to two instances of the $(n-1)$-disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy (or, to carry the original story further, to the junior monks at the temple).



The Tower of Hanoi algorithm; ignore everything but the bottom disk

Our recursive reduction does make one subtle but important assumption: *There is a largest disk*. In other words, our recursive algorithm works for any $n \geq 1$, but it breaks down when $n = 0$. We must handle that base case directly. Fortunately, the monks at Benares, being good Buddhists, are quite adept at moving zero disks from one peg to another in no time at all.



The base case for the Tower of Hanoi algorithm. There is no spoon.

While it's tempting to think about how all those smaller disks get moved—or more generally, what happens when the recursion is unrolled—it's not necessary. For even slightly more complicated algorithms, unrolling the recursion is far more confusing than illuminating. Our ***only*** task is to reduce the problem to one or more simpler instances, or to solve the problem directly if such a reduction is impossible. Our algorithm is trivially correct when $n = 0$. For any $n \geq 1$, the Recursion Fairy correctly moves (or more formally, the inductive hypothesis implies that our recursive algorithm correctly moves) the top $n-1$ disks, so (by induction) our algorithm must be correct.

Here's the recursive Hanoi algorithm in more typical pseudocode. This algorithm moves a stack of n disks from a source peg (*src*) to a destination peg (*dst*) using a third temporary peg (*tmp*) as a placeholder.

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n − 1, src, tmp, dst)
      move disk n from src to dst
      HANOI(n − 1, tmp, dst, src)
```

Let $T(n)$ denote the number of moves required to transfer n disks—the running time of our algorithm. Our vacuous base case implies that $T(0) = 0$, and the more general recursive algorithm implies that $T(n) = 2T(n-1) + 1$ for any $n \geq 1$. The annihilator method (or guessing and checking by induction) quickly gives us the closed form solution $\boldsymbol{T(n) = 2^n - 1}$. In particular, moving a tower of 64 disks requires $2^{64} - 1 =$ 18,446,744,073,709,551,615 individual moves. Thus, even at the impressive rate of one move per second, the monks at Benares will be at work for approximately 585 billion years before tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

1.4 Mergesort

Mergesort is one of the earliest algorithms proposed for sorting. According to Donald Knuth, it was proposed by John von Neumann as early as 1945.

1. Divide the input array into two subarrays of roughly equal size.
2. Recursively mergesort each of the subarrays.
3. Merge the newly-sorted subarrays into a single sorted array.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N |	G	E	X	A	M	P	L
Recurse:	I	N	O	S	R	T |	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X

A mergesort example.

The first step is completely trivial—we only need to compute the median array index—and we can delegate the second step to the Recursion Fairy. All the real work is done in the final step; the two sorted subarrays can be merged using a simple linear-time algorithm. Here's a complete description of the algorithm; to keep the recursive structure clear, we separate out the merge step as an independent subroutine.

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])
      MERGESORT(A[m+1..n])
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i+1
      else if i > m
          B[k] ← A[j]; j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i+1
      else
          B[k] ← A[j]; j ← j+1

  for k ← 1 to n
      A[k] ← B[k]
```

To prove that this algorithm is correct, we apply our old friend induction twice, first to the MERGE subroutine then to the top-level MERGESORT algorithm.

- We prove MERGE is correct by induction on $n-k+1$, which is the total size of the two sorted subarrays $A[i\,..\,m]$ and $A[j\,..\,n]$ that remain to be merged into $B[k\,..\,n]$ when the kth iteration of the main loop begins. There are five cases to consider. Yes, five.
  - If $k > n$, the algorithm correctly merges the two empty subarrays by doing absolutely nothing. (This is the base case of the inductive proof.)
  - If $i \le m$ and $j > n$, the subarray $A[j\,..\,n]$ is empty. Because both subarrays are sorted, the smallest element in the union of the two subarrays is $A[i]$. So the assignment $B[k] \leftarrow A[i]$ is correct. The inductive hypothesis implies that the remaining subarrays $A[i+1\,..\,m]$ and $A[j\,..\,n]$ are correctly merged into $B[k+1\,..\,n]$.
  - Similarly, if $i > m$ and $j \le n$, the assignment $B[k] \leftarrow A[j]$ is correct, and The Recursion Fairy correctly merges—sorry, I mean the inductive hypothesis implies that the MERGE algorithm correctly merges—the remaining subarrays $A[i\,..\,m]$ and $A[j+1\,..\,n]$ into $B[k+1\,..\,n]$.
  - If $i \le m$ and $j \le n$ and $A[i] < A[j]$, then the smallest remaining element is $A[i]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly merges the rest of the subarrays.
  - Finally, if $i \le m$ and $j \le n$ and $A[i] \ge A[j]$, then the smallest remaining element is $A[j]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly does the rest.
- Now we prove MERGESORT correct by induction; there are two cases to consider. Yes, two.
  - If $n \le 1$, the algorithm correctly does nothing.
  - Otherwise, the Recursion Fairy correctly sorts—sorry, I mean the induction hypothesis implies that our algorithm correctly sorts—the two smaller subarrays $A[1\,..\,m]$ and $A[m+1\,..\,n]$, after which they are correctly MERGEd into a single sorted array (by the previous argument).

What's the running time? Because the MERGESORT algorithm is recursive, its running time will be expressed by a recurrence. MERGE clearly takes linear time, because it's a simple for-loop with constant work per iteration. We immediately obtain the following recurrence for MERGESORT:

$$T(n) = T\big(\lceil n/2\rceil\big) + T\big(\lfloor n/2\rfloor\big) + O(n).$$

As in most divide-and-conquer recurrences, we can safely strip out the floors and ceilings using a domain transformation,[3] giving us the simpler recurrence

$$T(n) = 2T(n/2) + O(n).$$

The "all levels equal" case of the recursion tree method now immediately implies the closed-form solution $\boldsymbol{T(n) = O(n \log n)}$. (Recursion trees and domain transformations are described in detail in a separate note on solving recurrences.)

1.5 Quicksort

Quicksort is another recursive sorting algorithm, discovered by Tony Hoare in 1962. In this algorithm, the hard work is splitting the array into subsets so that merging the final result is trivial.

1. Choose a *pivot* element from the array.
2. Partition the array into three subarrays containing the elements smaller than the pivot, the pivot element itself, and the elements larger than the pivot.
3. Recursively quicksort the first and last subarray.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	P	$\boxed{\text{L}}$
Partition:	A	G	E	I	| L |	N	R	O	X	S	M	P	T
Recurse:	A	E	G	I	| L |	M	N	O	P	R	S	T	X

A quicksort example.

Here's a more detailed description of the algorithm. In the separate PARTITION subroutine, the input parameter p is index of the pivot element in the unsorted array; the subroutine partitions the array and returns the new index of the pivot.

```
QUICKSORT(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← PARTITION(A, p)
      QUICKSORT(A[1..r − 1])
      QUICKSORT(A[r + 1..n])
```

```
PARTITION(A[1..n], p):
  swap A[p] ↔ A[n]
  i ← 0
  j ← n
  while (i < j)
      repeat i ← i + 1 until (i ≥ j or A[i] ≥ A[n])
      repeat j ← j − 1 until (i ≥ j or A[j] ≤ A[n])
      if (i < j)
          swap A[i] ↔ A[j]
  swap A[i] ↔ A[n]
  return i
```

Just like mergesort, proving QUICKSORT is correct requires two separate induction proofs: one to prove that PARTITION correctly partitions the array, and the other to prove that QUICKSORT correctly sorts *assuming* PARTITION is correct. I'll leave the gory details as an exercise for the reader.

The analysis is also similar to mergesort. PARTITION runs in $O(n)$ time: $j - i = n$ at the beginning, $j - i = 0$ at the end, and we do a constant amount of work each time we increment i

[3] See the course notes on solving recurrences for more details.

or decrement j. For QuickSort, we get a recurrence that depends on r, the *rank* of the chosen pivot element:

$$T(n) = T(r-1) + T(n-r) + O(n)$$

If we could somehow choose the pivot to be the *median* element of the array A, we would have $r = \lceil n/2 \rceil$, the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n) = 2T\big(\lceil n/2 \rceil - 1\big) + T\big(\lfloor n/2 \rfloor\big) + O(n) \le 2T(n/2) + O(n),$$

and we'd have $T(n) = O(n \log n)$ by the recursion tree method.

In fact, as we will see shortly, we *can* locate the median element in an unsorted array in linear time. However, the algorithm is fairly complicated, and the hidden constant in the $O(\cdot)$ notation is large enough to make the resulting sorting algorithm impractical. In practice, most programmers settle for something simple, like choosing the first or last element of the array. In this case, r take any value between 1 and n, so we have

$$T(n) = \max_{1 \le r \le n} \big(T(r-1) + T(n-r) + O(n)\big).$$

In the worst case, the two subproblems are completely unbalanced—either $r = 1$ or $r = n$—and the recurrence becomes $T(n) \le T(n-1) + O(n)$. The solution is $\boldsymbol{T(n) = O(n^2)}$.

Another common heuristic is called "median of three"—choose three elements (usually at the beginning, middle, and end of the array), and take the median of those three elements the pivot. Although this heuristic is somewhat more efficient in practice than just choosing one element, especially when the array is already (nearly) sorted, we can still have $r = 2$ or $r = n - 1$ in the worst case. With the median-of-three heuristic, the recurrence becomes $T(n) \le T(1) + T(n-2) + O(n)$, whose solution is still $T(n) = O(n^2)$.

Intuitively, the pivot element will 'usually' fall somewhere in the middle of the array, say between $n/10$ and $9n/10$. This observation suggests that the *average-case* running time is $O(n \log n)$. Although this intuition is actually correct (at least under the right formal assumptions), we are still far from a *proof* that quicksort is usually efficient. We will formalize this intuition about average-case behavior in a later lecture.

1.6 The Pattern

Both mergesort and and quicksort follow a general three-step pattern shared by all divide and conquer algorithms:

1. **Divide** the given instance of the problem into several *independent smaller* instances.
2. **Delegate** each smaller instance to the Recursion Fairy.
3. **Combine** the solutions for the smaller instances into the final solution for the given instance.

If the size of any subproblem falls below some constant threshold, the recursion bottoms out. Hopefully, at that point, the problem is trivial, but if not, we switch to a different algorithm instead.

Proving a divide-and-conquer algorithm correct almost always requires induction. Analyzing the running time requires setting up and solving a recurrence, which usually (but unfortunately not always!) can be solved using recursion trees, perhaps after a simple domain transformation.

1.7 Median Selection

So how *do* we find the median element of an array in linear time? The following algorithm was discovered by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan in the early 1970s. Their algorithm actually solves the more general problem of selecting the kth largest element in an n-element array, given the array and the integer g as input, using a variant of an algorithm called either "quickselect" or "one-armed quicksort". The basic quickselect algorithm chooses a pivot element, partitions the array using the PARTITION subroutine from QUICKSORT, and then recursively searches only *one* of the two subarrays.

```
QUICKSELECT(A[1..n], k):
  if n = 1
      return A[1]
  else
      Choose a pivot element A[p]
      r ← PARTITION(A[1..n], p)

      if k < r
          return QUICKSELECT(A[1..r−1], k)
      else if k > r
          return QUICKSELECT(A[r+1..n], k−r)
      else
          return A[r]
```

The worst-case running time of QUICKSELECT obeys a recurrence similar to the quicksort recurrence. We don't know the value of r or which subarray we'll recursively search, so we'll just assume the worst.

$$T(n) \le \max_{1\le r\le n} \left(\max\{T(r-1), T(n-r)\} + O(n)\right)$$

We can simplify the recurrence by using ℓ to denote the length of the recursive subproblem:

$$T(n) \le \max_{0\le \ell\le n-1} T(\ell) + O(n) \le T(n-1) + O(n)$$

As with quicksort, we get the solution $T(n) = O(n^2)$ when $\ell = n-1$, which happens when the chosen pivot element is either the smallest element or largest element of the array.

On the other hand, we could avoid this quadratic behavior if we could somehow magically choose a *good* pivot, where $\ell \le \alpha n$ for some constant $\alpha < 1$. In this case, the recurrence would simplify to

$$T(n) \le T(\alpha n) + O(n).$$

This recurrence expands into a descending geometric series, which is dominated by its largest term, so $T(n) = O(n)$.

The Blum-Floyd-Pratt-Rivest-Tarjan algorithm chooses a good pivot for one-armed quicksort by *recursively computing the median* of a carefully-selected subset of the input array.

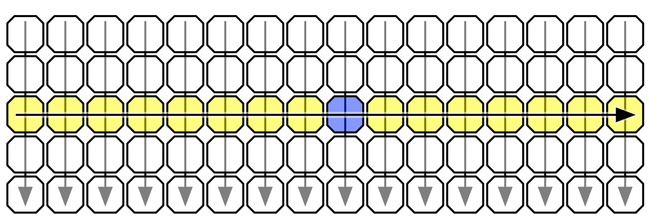
```
MOM5SELECT(A[1 .. n], k):
  if n ≤ 25
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MEDIANOFFIVE(A[5i − 4 .. 5i])   ⟨⟨Brute force!⟩⟩
      mom ← MOMSELECT(M[1 .. m], ⌊m/2⌋)         ⟨⟨Recursion!⟩⟩

      r ← PARTITION(A[1 .. n], mom)

      if k < r
          return MOMSELECT(A[1 .. r − 1], k)     ⟨⟨Recursion!⟩⟩
      else if k > r
          return MOMSELECT(A[r + 1 .. n], k − r) ⟨⟨Recursion!⟩⟩
      else
          return mom
```

The recursive structure of the algorithm requires a slightly larger base case. There's absolutely nothing special about the constant 25 in the pseudocode; for theoretical purposes, any other constant like 42 or 666 or 8765309 would work just as well.

If the input array is too large to handle by brute force, we divide it into $\lceil n/5 \rceil$ *blocks*, each containing exactly 5 elements, except possibly the last. (If the last block isn't full, just throw in a few ∞s.) We find the median of each block by brute force and collect those medians into a new array $M[1..\lceil n/5 \rceil]$. Then we recursively compute the median of this new array. Finally we use the median of medians — hence 'mom' — as the pivot in one-armed quicksort.

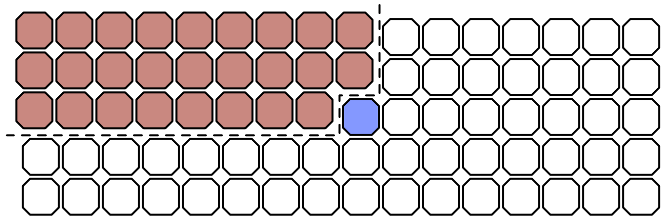
The key insight is that neither of these two subarrays can be too large. The median of medians is larger than $\lceil \lceil n/5 \rceil/2 \rceil - 1 \approx n/10$ block medians, and each of those medians is larger than two other elements in its block. Thus, mom is larger than at least $3n/10$ elements in the input array, and symmetrically, mom is smaller than at least $3n/10$ input elements. Thus, in the worst case, the final recursive call searches an array of size $7n/10$.

We can visualize the algorithm's behavior by drawing the input array as a $5 \times \lceil n/5 \rceil$ grid, which each column represents five consecutive elements. For purposes of illustration, imagine that we sort every column from top down, and then we sort the columns by their middle element. (Let me emphasize that *the algorithm does not actually do this!*) In this arrangement, the median-of-medians is the element closest to the center of the grid.



Visualizing the median of medians

The left half of the first three rows of the grid contains $3n/10$ elements, each of which is smaller than the median-of-medians. If the element we're looking for is larger than the median-of-medians, our algorithm will throw away *everything* smaller than the median-of-median, including those $3n/10$ elements, before recursing. Thus, the input to the recursive subproblem contains at most $7n/10$ elements. A symmetric argument applies when our target element is smaller than the median-of-medians.



Discarding approximately 3/10 of the array

We conclude that the worst-case running time of the algorithm obeys the following recurrence:

$$T(n) \le O(n) + T(n/5) + T(7n/10).$$

The recursion tree method implies the solution $T(n) = O(n)$.

Finer analysis reveals that the constant hidden by the $O()$ is quite large, even if we count only comparisons; this is not a practical algorithm for small inputs. (In particular, mergesort uses fewer comparisons in the worst case when $n < 4{,}000{,}000$.) Selecting the median of 5 elements requires at most 6 comparisons, so we need at most $6n/5$ comparisons to set up the recursive subproblem. We need another $n-1$ comparisons to partition the array after the recursive call returns. So a more accurate recurrence for the total number of comparisons is

$$T(n) \le 11n/5 + T(n/5) + T(7n/10).$$

The recursion tree method implies the upper bound

$$T(n) \le \frac{11n}{5} \sum_{i \ge 0} \left(\frac{9}{10}\right)^i = \frac{11n}{5} \cdot 10 = 22n.$$

1.8 Multiplication

Adding two n-digit numbers takes $O(n)$ time by the standard iterative 'ripple-carry' algorithm, using a lookup table for each one-digit addition. Similarly, multiplying an n-digit number by a one-digit number takes $O(n)$ time, using essentially the same algorithm.

What about multiplying two n-digit numbers? In most of the world, grade school students (supposedly) learn to multiply by breaking the problem into n one-digit multiplications and n additions:

$$\begin{array}{r}
31415962 \\
\underline{\times\ 27182818} \\
251327696 \\
31415962 \\
251327696 \\
62831924 \\
251327696 \\
31415962 \\
219911734 \\
\underline{62831924} \\
853974377340916
\end{array}$$

We could easily formalize this algorithm as a pair of nested for-loops. The algorithm runs in $\Theta(n^2)$ time—altogether, there are $\Theta(n^2)$ digits in the partial products, and for each digit, we spend constant time. The Egyptian/Russian peasant multiplication algorithm described in the first lecture also runs in $\Theta(n^2)$ time.

Perhaps we can get a more efficient algorithm by exploiting the following identity:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

Here is a divide-and-conquer algorithm that computes the product of two n-digit numbers x and y, based on this formula. Each of the four sub-products e, f, g, h is computed recursively. The last line does not involve any multiplications, however; to multiply by a power of ten, we just shift the digits and fill in the right number of zeros.

```
MULTIPLY(x, y, n):
  if n = 1
        return x · y
  else
        m ← ⌈n/2⌉
        a ← ⌊x/10^m⌋;  b ← x mod 10^m
        d ← ⌊y/10^m⌋;  c ← y mod 10^m
        e ← MULTIPLY(a, c, m)
        f ← MULTIPLY(b, d, m)
        g ← MULTIPLY(b, c, m)
        h ← MULTIPLY(a, d, m)
        return 10^{2m} e + 10^m (g + h) + f
```

You can easily prove by induction that this algorithm is correct. The running time for this algorithm is given by the recurrence

$$T(n) = 4T(\lceil n/2 \rceil) + \Theta(n), \qquad T(1) = 1,$$

which solves to $T(n) = \Theta(n^2)$ by the recursion tree method (after a simple domain transformation). Hmm... I guess this didn't help after all.

In the mid-1950s, the famous Russian mathematician Andrey Kolmogorov conjectured that there is *no* algorithm to multiply two n-digit numbers in $o(n^2)$ time. However, in 1960, after Kolmogorov posed his conjecture at a seminar at Moscow University, Anatoliĭ Karatsuba, one of the students in the seminar, discovered a remarkable counterexample. According to Karastuba himself,

> After the seminar I told Kolmogorov about the new algorithm and about the disproof of the n^2 conjecture. Kolmogorov was very agitated because this contradicted his very plausible conjecture. At the next meeting of the seminar, Kolmogorov himself told the participants about my method, and at that point the seminar was terminated.

Karastuba observed that the middle coefficient $bc + ad$ can be computed from the other two coefficients ac and bd using only *one* more recursive multiplication, via the following algebraic identity:

$$ac + bd - (a - b)(c - d) = bc + ad$$

This trick lets us replace the last three lines in the previous algorithm as follows:

```
FASTMULTIPLY(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m
      d ← ⌊y/10^m⌋;  c ← y mod 10^m
      e ← FASTMULTIPLY(a, c, m)
      f ← FASTMULTIPLY(b, d, m)
      g ← FASTMULTIPLY(a − b, c − d, m)
      return 10^(2m) e + 10^m (e + f − g) + f
```

The running time of Karatsuba's FASTMULTIPLY algorithm is given by the recurrence

$$T(n) \leq 3T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1.$$

After a domain transformation, we can plug this into a recursion tree to get the solution $T(n) = O(n^{\lg 3}) = O(n^{1.585})$, a significant improvement over our earlier quadratic-time algorithm.[4] Karastuba's algorithm arguably launched the design and analysis of algorithms as a formal field of study.

Of course, in practice, all this is done in binary instead of decimal.

We can take this idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to obtain even faster multiplication algorithms. Andrei Toom and Stephen Cook discovered an infinite family of algorithms that split any integer into k parts, each with n/k digits, and then compute the product using only $2k-1$ recursive multiplications. For any fixed k, the resulting algorithm runs in $O(n^{1+1/(\lg k)})$ time, where the hidden constant in the $O(\cdot)$ notation depends on k.

Ultimately, this divide-and-conquer strategy led Gauss (yes, really) to the discovery of the *Fast Fourier transform*, which we discuss in detail in the next lecture note. The fastest multiplication algorithm known, published by Martin Fürer in 2007 and based on FFTs, runs in $n \log n 2^{O(\log^* n)}$ time. Here, $\log^* n$ is the slowly growing *iterated logarithm* of n, which is the number of times one must take the logarithm of n before the value is less than 1:

$$\lg^* n = \begin{cases} 1 & \text{if } n \leq 2, \\ 1 + \lg^*(\lg n) & \text{otherwise.} \end{cases}$$

(For all practical purposes, $\log^* n \leq 6$.) It is widely conjectured that the best possible algorithm for multiply two n-digit numbers runs in $\Theta(n \log n)$ time.

1.9 Exponentiation

Given a number a and a positive integer n, suppose we want to compute a^n. The standard naïve method is a simple for-loop that does $n-1$ multiplications by a:

```
SLOWPOWER(a, n):
  x ← a
  for i ← 2 to n
      x ← x · a
  return x
```

[4]Karatsuba actually proposed an algorithm based on the formula $(a+c)(b+d)-ac-bd = bc+ad$. This algorithm also runs in $O(n^{\lg 3})$ time, but the actual recurrence is a bit messier: $a-b$ and $c-d$ are still m-digit numbers, but $a+b$ and $c+d$ might have $m+1$ digits. The simplification presented here is due to Donald Knuth. The same technique was used by Gauss in the 1800s to multiply two complex numbers using only three real mutliplications.

This iterative algorithm requires n multiplications.

Notice that the input a could be an integer, or a rational, or a floating point number. In fact, it doesn't need to be a number at all, as long as it's something that we know how to multiply. For example, the same algorithm can be used to compute powers modulo some finite number (an operation commonly used in cryptography algorithms) or to compute powers of matrices (an operation used to evaluate recurrences and to compute shortest paths in graphs). All we really require is that a belong to a multiplicative group.[5] Since we don't know what kind of things we're multiplying, we can't know how long a multiplication takes, so we're forced analyze the running time in terms of the number of multiplications.

There is a much faster divide-and-conquer method, using the following simple recursive formula:

$$a^n = a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil}.$$

What makes this approach more efficient is that once we compute the first factor $a^{\lfloor n/2 \rfloor}$, we can compute the second factor $a^{\lceil n/2 \rceil}$ using at most one more multiplication.

```
FASTPOWER(a, n):
  if n = 1
      return a
  else
      x ← FASTPOWER(a, ⌊n/2⌋)
      if n is even
          return x · x
      else
          return x · x · a
```

The total number of multiplications satisfies the recurrence $T(n) \le T(\lfloor n/2 \rfloor) + 2$, with the base case $T(1) = 0$. After a domain transformation, recursion trees give us the solution $T(n) = O(\log n)$.

Incidentally, this algorithm is asymptotically optimal—any algorithm for computing a^n must perform at least $\Omega(\log n)$ multiplications. In fact, when n is a power of two, this algorithm is *exactly* optimal. However, there are slightly faster methods for other values of n. For example, our divide-and-conquer algorithm computes a^{15} in six multiplications ($a^{15} = a^7 \cdot a^7 \cdot a$; $a^7 = a^3 \cdot a^3 \cdot a$; $a^3 = a \cdot a \cdot a$), but only five multiplications are necessary ($a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$). It is an open question whether the absolute minimum number of multiplications for a given exponent n can be computed efficiently.

Exercises

1. Prove that the Russian peasant multiplication algorithm runs in $\Theta(n^2)$ time, where n is the total number of input digits.

2. (a) Professor George O'Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character **&**. Preorder and postorder traversals of the tree visit the nodes in the following order:

[5]A *multiplicative group* $(G, \otimes)$ is a set G and a function $\otimes : G \times G \to G$, satisfying three axioms:
1. There is a *unit* element $1 \in G$ such that $1 \otimes g = g \otimes 1$ for any element $g \in G$.
2. Any element $g \in G$ has a *inverse* element $g^{-1} \in G$ such that $g \otimes g^{-1} = g^{-1} \otimes g = 1$
3. The function is *associative*: for any elements $f, g, h \in G$, we have $f \otimes (g \otimes h) = (f \otimes g) \otimes h$.

- Preorder: **I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X**
- Postorder: **H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I**

Draw George's binary tree.

(b) Prove that there is no algorithm to reconstruct an *arbtirary* binary tree from its preorder and postorder node sequences.

(c) Recall that a binary tree is *full* if every non-leaf node has exactly two children. Describe and analyze a recursive algorithm to reconstruct an arbitrary *full* binary tree, given its preorder and postorder node sequences as input.

(d) Describe and analyze a recursive algorithm to reconstruct an arbtirary binary tree, given its preorder and *inorder* node sequences as input.

(e) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence. Assume all input keys are distinct. For extra credit, describe an algorithm that runs in $O(n)$ time.

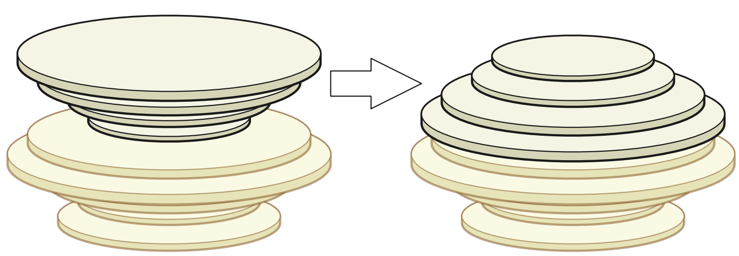
In parts (b), (c), and (d), assume that all keys are distinct and that the input is consistent with at least one binary tree.

3. Consider a $2^n \times 2^n$ chessboard with one (arbitrarily chosen) square removed.

(a) Prove that any such chessboard can be tiled without gaps or overlaps by L-shaped pieces, each composed of 3 squares.

(b) Describe and analyze an algorithm to compute such a tiling, given the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time.

4. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.



Flipping the top four pancakes.

(a) Describe an algorithm to sort an arbitrary stack of n pancakes using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

(b) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

5. Prove that the original recursive Tower of Hanoi algorithm is *exactly equivalent* to each of the following non-recursive algorithms. In other words, prove that all three algorithms move exactly the same sequence of disks, to and from the same pegs, in the same order. The pegs are labeled 0, 1, and 2, and our problem is to move a stack of n disks from peg 0 to peg 2 (as shown on page **??**).

   (a) Repeatedly make the only legal move that satisfies the following constraints:
   - Never move the same disk twice in a row.
   - If n is even, always move the smallest disk forward ($\cdots \to 0 \to 1 \to 2 \to 0 \to \cdots$).
   - If n is odd, always move the smallest disk backward ($\cdots \to 0 \to 2 \to 1 \to 0 \to \cdots$).

   If there is no move that satisfies these three constraints, the puzzle is solved.

   (b) Start by putting your finger on the top of peg 0. Then repeat the following steps:
   i. If n is odd, move your finger to the next peg ($\cdots \to 0 \to 1 \to 2 \to 0 \to \cdots$).
   ii. If n is even, move your finger to the previous peg ($\cdots \to 0 \to 2 \to 1 \to 0 \to \cdots$).
   iii. Make the only legal move that does not require you to lift your finger. If there is no such move, the puzzle is solved.

   (c) Let $\rho(n)$ denote the smallest integer k such that $n/2^k$ is not an integer. For example, $\rho(42) = 2$, because $42/2^1$ is an integer but $42/2^2$ is not. (Equivalently, $\rho(n)$ is one more than the position of the least significant 1 in the binary representation of n.) Because its behavior resembles the marks on a ruler, $\rho(n)$ is sometimes called the *ruler function*:

   $$1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 6, 1, 2, 1, 3, 1, \ldots$$

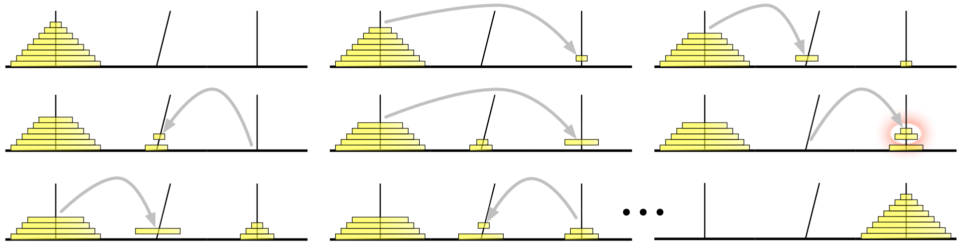
   Here's the non-recursive algorithm in one line:

   $$\boxed{\textbf{In step } \boldsymbol{i}\textbf{, move disk } \boldsymbol{\rho(i)} \textbf{ forward if } \boldsymbol{n-i} \textbf{ is even, backward if } \boldsymbol{n-i} \textbf{ is odd.}}$$

   When this rule requires us to move disk $n+1$, the puzzle is solved.

6. A less familiar chapter in the Tower of Hanoi's history is its brief relocation of the temple from Benares to Pisa in the early 13th century. The relocation was organized by the wealthy merchant-mathematician Leonardo Fibonacci, at the request of the Holy Roman Emperor Frederick II, who had heard reports of the temple from soldiers returning from the Crusades. The Towers of Pisa and their attendant monks became famous, helping to establish Pisa as a dominant trading center on the Italian peninsula.

   Unfortunately, almost as soon as the temple was moved, one of the diamond needles began to lean to one side. To avoid the possibility of the leaning tower falling over from too much use, Fibonacci convinced the priests to adopt a more relaxed rule: ***Any number of disks on the leaning needle can be moved together to another needle in a single move.*** It was still forbidden to place a larger disk on top of a smaller disk, and disks had to be moved one at a time *onto* the leaning needle or between the two vertical needles.



The Towers of Pisa. In the fifth move, two disks are taken off the leaning needle.

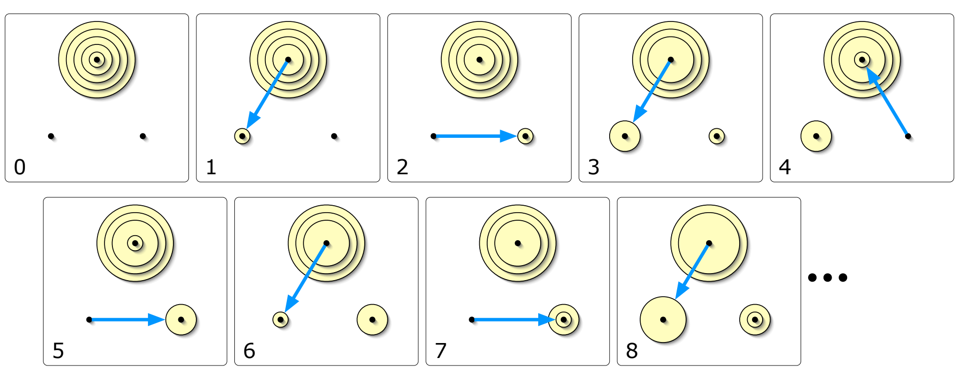
Thanks to Fibonacci's new rule, the priests could bring about the end of the universe somewhat faster from Pisa then they could than could from Benares. Fortunately, the temple was moved from Pisa back to Benares after the newly crowned Pope Gregory IX excommunicated Frederick II, making the local priests less sympathetic to hosting foreign heretics with strange mathematical habits. Soon afterward, a bell tower was erected on the spot where the temple once stood; it too began to lean almost immediately.

Describe an algorithm to transfer a stack of n disks from one *vertical* needle to the other *vertical* needle, using the smallest possible number of moves. *Exactly* how many moves does your algorithm perform?

7. Consider the following restricted variants of the Tower of Hanoi puzzle. In each problem, the pegs are numbered 0, 1, and 2, as in problem **??**, and your task is to move a stack of n disks from peg 1 to peg 2.

   (a) Suppose you are forbidden to move any disk directly between peg 1 and peg 2; *every* move must involve peg 0. Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?

   (b) Suppose you are only allowed to move disks from peg 0 to peg 2, from peg 2 to peg 1, or from peg 1 to peg 0. Equivalently, suppose the pegs are arranged in a circle and numbered in clockwise order, and you are only allowed to move disks counterclockwise. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make?



A top view of the first eight moves in a counterclockwise Towers of Hanoi solution

   ★(c) Finally, suppose your only restriction is that you may never move a disk directly from peg 1 to peg 2. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make? *[Hint: This variant is considerably harder to analyze than the other two.]*

8. A German mathematician developed a new variant of the Towers of Hanoi game, known in the US literature as the "Liberty Towers" game.[6] In this variant, there is a row of k pegs, numbered from 1 to k. In a single turn, you are allowed to move the smallest disk on peg i to either peg $i-1$ or peg $i+1$, for any index i; as usual, you are not allowed to place a bigger disk on a smaller disk. Your mission is to move a stack of n disks from peg 1 to peg k.

   (a) Describe a recursive algorithm for the case $k = 3$. *Exactly* how many moves does your algorithm make? (This is the same as part (a) of the previous question.)

   (b) Describe a recursive algorithm for the case $k = n+1$ that requires at most $O(n^3)$ moves. *[Hint: Use part (a).]*

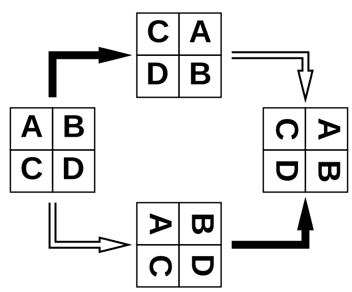
   (c) Describe a recursive algorithm for the case $k = n+1$ that requires at most $O(n^2)$ moves. *[Hint: Don't use part (a).]*

   (d) Describe a recursive algorithm for the case $k = \sqrt{n}$ that requires at most a polynomial number of moves. (What polynomial??)

   *(e) Describe and analyze a recursive algorithm for arbitrary n and k. How small must k be (as a function of n) so that the number of moves is bounded by a polynomial in n?

9. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

   Suppose we want to rotate an $n \times n$ pixel map 90° clockwise. One way to do this, at least when n is a power of two, is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. (Why five? For the same reason the Tower of Hanoi puzzle needs a third peg.) Alternately, we could *first* recursively rotate the blocks and *then* blit them into place.



Two algorithms for rotating a pixel map.
Black arrows indicate blitting the blocks into place; white arrows indicate recursively rotating the blocks.

   (a) Prove that both versions of the algorithm are correct when n is a power of 2.

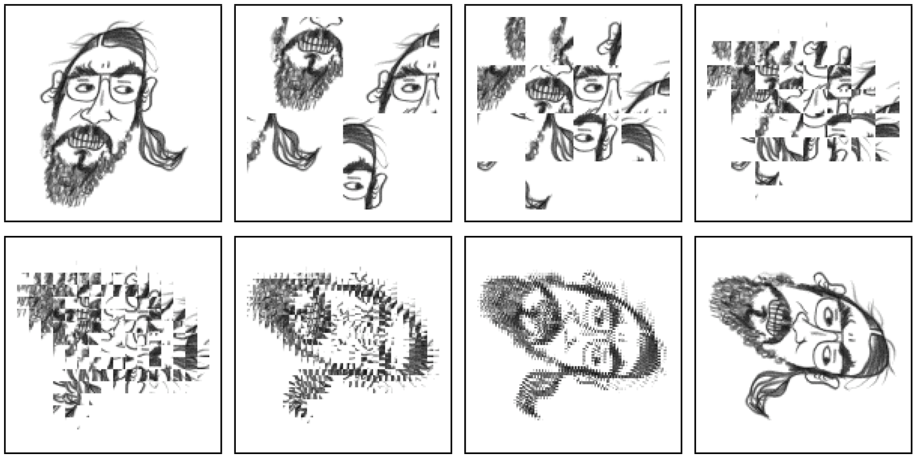
   (b) *Exactly* how many blits does the algorithm perform when n is a power of 2?

   (c) Describe how to modify the algorithm so that it works for arbitrary n, not just powers of 2. How many blits does your modified algorithm perform?

   (d) What is your algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

   (e) What if a $k \times k$ blit takes only $O(k)$ time?

[6] No it isn't.



The first rotation algorithm (blit then recurse) in action.

10. Prove that quicksort with the median-of-three heuristic requires $\Omega(n^2)$ time to sort an array of size n in the worst case. Specifically, for any integer n, describe a permutation of the integers 1 through n, such that in every recursive call to median-of-three-quicksort, the pivot is always the second smallest element of the array. Designing this permutation requires intimate knowledge of the PARTITION subroutine.

    (a) As a warm-up exercise, assume that the PARTITION subroutine is *stable*, meaning it preserves the existing order of all elements smaller than the pivot, and it preserves the existing order of all elements smaller than the pivot.

    (b) Assume that the PARTITION subroutine uses the specific algorithm listed on page **??** of this lecture note, which is *not* stable.

11. (a) Prove that the following algorithm actually sorts its input.

```
StoogeSort(A[0..n−1]):
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m = ⌈2n/3⌉
      StoogeSort(A[0..m−1])
      StoogeSort(A[n−m..n−1])
      StoogeSort(A[0..m−1])
```

    (b) Would STOOGESORT still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.

    (c) State a recurrence (including the base case(s)) for the number of comparisons executed by STOOGESORT.

    (d) Solve the recurrence, and prove that your solution is correct. *[Hint: Ignore the ceiling.]*

    (e) Prove that the number of *swaps* executed by STOOGESORT is at most $\binom{n}{2}$.

12. Consider the following cruel and unusual sorting algorithm.

```
CRUEL(A[1..n]):
  if n > 1
      CRUEL(A[1..n/2])
      CRUEL(A[n/2+1..n])
      UNUSUAL(A[1..n])
```

```
UNUSUAL(A[1..n]):
  if n = 2
      if A[1] > A[2]                  ⟨⟨the only comparison!⟩⟩
          swap A[1] ↔ A[2]
  else
      for i ← 1 to n/4                ⟨⟨swap 2nd and 3rd quarters⟩⟩
          swap A[i+n/4] ↔ A[i+n/2]
      UNUSUAL(A[1..n/2])              ⟨⟨recurse on left half⟩⟩
      UNUSUAL(A[n/2+1..n])            ⟨⟨recurse on right half⟩⟩
      UNUSUAL(A[n/4+1..3n/4])         ⟨⟨recurse on middle half⟩⟩
```

Notice that the comparisons performed by the algorithm do not depend at all on the values in the input array; such a sorting algorithm is called ***oblivious***. Assume for this problem that the input size n is always a power of 2.

(a) Prove by induction that CRUEL correctly sorts any input array. *[Hint: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough?]*

(b) Prove that CRUEL would *not* correctly sort if we removed the for-loop from UNUSUAL.

(c) Prove that CRUEL would *not* correctly sort if we swapped the last two lines of UNUSUAL.

(d) What is the running time of UNUSUAL? Justify your answer.

(e) What is the running time of CRUEL? Justify your answer.

13. You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates; each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the *same* party or not simply by introducing them to each other—members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

(a) Suppose more than half of the delegates belong to the same political party. Describe an efficient algorithm that identifies all members of this majority party.

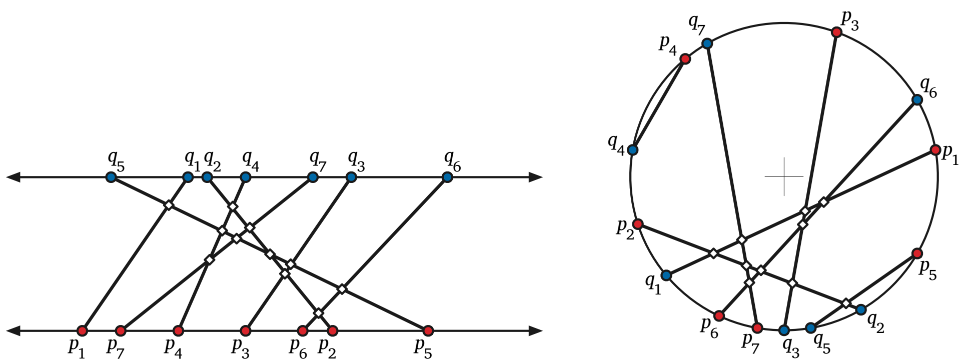
(b) Now suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick out the people from the plurality political party as parsimoniously as possible. (Please.)

14. An *inversion* in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward). Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. *[Hint: Modify mergesort.]*

15. (a) Suppose you are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Create a set of n line segments by connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time.

    (b) Now suppose you are given two sets $\{p_1, p_2, \ldots, p_n\}$ and $\{q_1, q_2, \ldots, q_n\}$ of n points *on the unit circle*. Connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect in $O(n \log^2 n)$ time. *[Hint: Use your solution to part (a).]*

    (c) Solve the previous problem in $O(n \log n)$ time.



Eleven intersecting pairs of segments with endpoints on parallel lines,
and ten intersecting pairs of segments with endpoints on a circle.

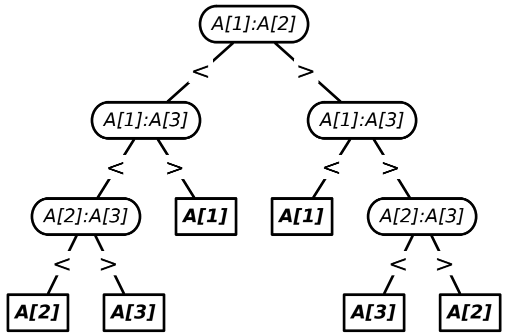
16. Suppose we are given a set S of n items, each with a *value* and a *weight*. For any element $x \in S$, we define two subsets

    - $S_{<x}$ is the set of all elements of S whose value is smaller than the value of x.
    - $S_{>x}$ is the set of all elements of S whose value is larger than the value of x.

    For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The ***weighted median*** of R is any element x such that $w(S_{<x}) \le w(S)/2$ and $w(S_{>x}) \le w(S)/2$.

    Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

17. Describe an algorithm to compute the median of an array $A[1..5]$ of distinct numbers using at most 6 comparisons. Instead of writing pseudocode, describe your algorithm using a ***decision tree***: A binary tree where each internal node contains a comparison of the form "$A[i] \gtrless A[j]$?" and each leaf contains an index into the array.



Finding the median of a 3-element array using at most 3 comparisons

18. Consider the following generalization of the Blum-Floyd-Pratt-Rivest-Tarjan SELECT algorithm, which partitions the input array into $\lceil n/b \rceil$ blocks of size b, instead of $\lceil n/5 \rceil$ blocks of size 5, but is otherwise identical. In the pseudocode below, the necessary modifications are indicated in red.

```
MomₙSelect(A[1..n], k):
  if n ≤ b²
      use brute force
  else
      m ← ⌈n/b⌉
      for i ← 1 to m
          M[i] ← MedianOfB(A[b(i−1)+1..bi])
      mom_b ← Mom_bSelect(M[1..m], ⌊m/2⌋)
      r ← Partition(A[1..n], mom_b)
      if k < r
          return Mom_bSelect(A[1..r−1], k)
      else if k > r
          return Mom_bSelect(A[r+1..n], k−r)
      else
          return mom_b
```

(a) State a recurrence for the running time of $\text{Mom}_b\text{Select}$, assuming that b is a constant (so the subroutine MedianOfB runs in $O(1)$ time). In particular, how do the sizes of the recursive subproblems depend on the constant b? Consider even b and odd b separately.

(b) What is the running time of $\text{Mom}_1\text{Select}$? *[Hint: This is a trick question.]*

*(c) What is the running time of $\text{Mom}_2\text{Select}$? *[Hint: This is an unfair question.]*

(d) What is the running time of $\text{Mom}_3\text{Select}$?

(e) What is the running time of $\text{Mom}_4\text{Select}$?

(f) For any constants $b \geq 5$, the algorithm $\text{Mom}_b\text{Select}$ runs in $O(n)$ time, but different values of b lead to different constant factors. Let $M(b)$ denote the minimum number of comparisons required to find the median of b numbers. The exact value of $M(b)$ is known only for $b \leq 13$:

b	1	2	3	4	5	6	7	8	9	10	11	12	13
$M(b)$	0	1	3	4	6	8	10	12	14	16	18	20	23

For each b between 5 and 13, find an upper bound on the running time of $\text{MoM}_b\text{Select}$ of the form $T(n) \le \alpha_b n$ for some explicit constant α_b. (For example, on page 8 we showed that $\alpha_5 \le 22$.)

(g) Which value of b yields the smallest constant α_b? *[Hint: This is a trick question.]*

19. An array $A[0..n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3	is bitonic, but
3	6	9	8	7	5	1	2	4	is *not* bitonic.

Describe and analyze an algorithm to find the *smallest* element in an n-element bitonic array in $O(\log n)$ time. You may assume that the numbers in the input array are distinct.

20. Suppose we are given an array $A[1..n]$ with the special property that $A[1] \ge A[2]$ and $A[n-1] \le A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \ge A[x]$ and $A[x] \le A[x+1]$. For example, there are six local minima in the following array:

9	7	7	2	1	3	7	5	4	7	3	3	4	8	6	9

We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array **must** have at least one local minimum. Why?]*

21. Suppose you are given a sorted array of n distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, you are given an array $A[1..n]$ such that the prefix $A[1..k]$ is sorted in increasing order, the suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$.

For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	−4	0	2	5	7	8

(a) Describe and analyze an algorithm to compute the unknown integer k.

(b) Describe and analyze an algorithm to determine if the given array contains a given number x.

22. You are a contestant on the hit game show "Beat Your Neighbors!" You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than any of your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni™, to which you are hopelessly addicted.

(a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than either of its neighbors. How many boxes does your algorithm open in the worst case?

*(b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

*(c) Prove that your solution to part (b) is optimal up to a constant factor.

23. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B in $\Theta(\log n)$ time. For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, your algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicate elements. *[Hint: First solve the special case $k = n$.]*

    (b) Now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B \cup C$ in $O(\log n)$ time.

    (c) Finally, suppose we are given a two dimensional array $A[1..m][1..n]$ in which every row $A[i][\,]$ is sorted, and an integer k. Describe an algorithm to find the kth smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? *[Hint: Use the linear-time* Select *algorithm as a subroutine.]*

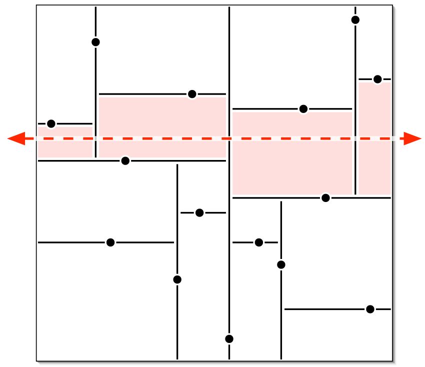
24. (a) Describe an algorithm that sorts an input array $A[1..n]$ by calling a subroutine SqrtSort(k), which sorts the subarray $A\left[k+1..k+\sqrt{n}\right]$ in place, given an arbitrary integer k between 0 and $n-\sqrt{n}$ as input. (To simplify the problem, assume that $\sqrt{n}$ is an integer.) Your algorithm is ***only*** allowed to inspect or modify the input array by calling SqrtSort; in particular, your algorithm must not directly compare, move, or copy array elements. How many times does your algorithm call SqrtSort in the worst case?

    (b) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SqrtSort, prove that no algorithm can sort using $o(f(n))$ calls to SqrtSort. *[Hint: See Lecture 19.]*

    (c) Now suppose SqrtSort is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k}, so that repeated square roots are always integers.)

25. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the points as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point inside the box (*not* on its boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.

    (a) How many cells are there, as a function of n? Prove your answer is correct.



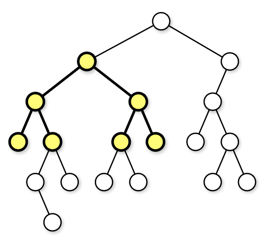
A kd-tree for 15 points. The dashed line crosses the four shaded cells.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[Hint: There is more than one function f such that $f(16) = 4$.]*

(c) Suppose we are given n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

(d) Describe an analyze an efficient algorithm that counts, given a kd-tree storing n points, the number of points that lie inside a rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

26. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

*27. Bob Ratenbur, a new student in CS 225, is trying to write code to perform preorder, inorder, and postorder traversals of binary trees. Bob understands the basic idea behind the traversal algorithms, but whenever he tries to implement them, he keeps mixing up the recursive calls. Five minutes before the deadline, Bob frantically submits code with the following structure:

```
PreOrder(v):
  if v = Null
      return
  else
      print label(v)
      [redacted]Order(left(v))
      [redacted]Order(right(v))
```

```
InOrder(v):
  if v = Null
      return
  else
      [redacted]Order(left(v))
      print label(v)
      [redacted]Order(right(v))
```

```
PostOrder(v):
  if v = Null
      return
  else
      [redacted]Order(left(v))
      [redacted]Order(right(v))
      print label(v)
```

Each [redacted] hides one of the prefixes Pre, In, or Post. Moreover, each of the following function calls appears exactly once in Bob's submitted code:

PreOrder(*left*(v))	PreOrder(*right*(v))
InOrder(*left*(v))	InOrder(*right*(v))
PostOrder(*left*(v))	PostOrder(*right*(v))

Thus, there are precisely 36 possibilities for Bob's code. Unfortunately, Bob accidentally deleted his source code after submitting the executable, so neither you nor he knows which functions were called where.

Now suppose you are given the output of Bob's traversal algorithms, executed on some ***unknown*** binary tree T. Bob's output has been helpfully parsed into three arrays $Pre[1..n]$, $In[1..n]$, and $Post[1..n]$. You may assume that these traversal sequences are consistent with exactly one binary tree T; in particular, the vertex labels of the unknown tree T are distinct, and every internal node in T has exactly two children.

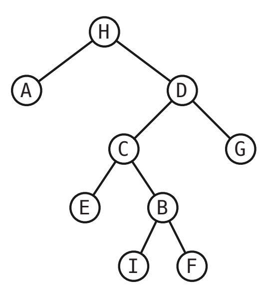
(a) Describe an algorithm to reconstruct the unknown tree T from the given traversal sequences.

(b) Describe an algorithm that either reconstruct Bob's code from the given traversal sequences, or correctly reports that the traversal sequences are consistent with more than one set of algorithms.

For example, given the input

$$Pre[1..n] = [\texttt{H A E C B I F G D}]$$
$$In[1..n] = [\texttt{A H D C E I F B G}]$$
$$Post[1..n] = [\texttt{A E I B F C D G H}]$$

your first algorithm should return the following tree:



and your second algorithm should reconstruct the following code:

```
PREORDER(v):
  if v = NULL
      return
  else
      print label(v)
      PREORDER(left(v))
      POSTORDER(right(v))
```

```
INORDER(v):
  if v = NULL
      return
  else
      POSTORDER(left(v))
      print label(v)
      PREORDER(right(v))
```

```
POSTORDER(v):
  if v = NULL
      return
  else
      INORDER(left(v))
      INORDER(right(v))
      print label(v)
```

28. Consider the following classical recursive algorithm for computing the factorial $n!$ of a non-negative integer n:

```
FACTORIAL(n):
  if n = 0
      return 1
  else
      return n · FACTORIAL(n − 1)
```

(a) How many multiplications does this algorithm perform?

(b) How many bits are required to write $n!$ in binary? Express your answer in the form $\Theta(f(n))$, for some familiar function $f(n)$. *[Hint: $(n/2)^{n/2} < n! < n^n$.]*

(c) Your answer to (b) should convince you that the number of multiplications is *not* a good estimate of the actual running time of FACTORIAL. We can multiply any k-digit number and any l-digit number in $O(k \cdot l)$ time using the grade-school algorithm (or the Russian peasant algorithm). What is the running time of FACTORIAL if we use this multiplication algorithm as a subroutine?

*(d) The following algorithm also computes the factorial function, but using a different grouping of the multiplications:

```
FACTORIAL2(n, m):          ⟨⟨Compute n!/(n − m)!⟩⟩
  if m = 0
      return 1
  else if m = 1
      return n
  else
      return FACTORIAL2(n, ⌊m/2⌋) · FACTORIAL2(n − ⌊m/2⌋, ⌈m/2⌉)
```

What is the running time of FACTORIAL2(n, n) if we use grade-school multiplication? *[Hint: Ignore the floors and ceilings.]*

(e) Describe and analyze a variant of Karastuba's algorithm that can multiply any k-digit number and any l-digit number, where $k \geq l$, in $O(k \cdot l^{\lg 3 - 1}) = O(k \cdot l^{0.585})$ time.

*(f) What are the running times of FACTORIAL(n) and FACTORIAL2(n, n) if we use the modified Karatsuba multiplication from part (e)?

Ceterum in problematis natura fundatum est, ut methodi quaecunque *continuo prolixiores evadant, quo maiores sunt numeri, ad quos applicantur; attamen pro methodis sequentibus difficultates perlente increscunt, numerique e septem, octos vel adeo adhuc pluribus figuris constantes praesertim per secundam felici semper successu tractati fuerunt, omnique celeritate, quam pro tantis numeris exspectare aequum est, qui secundum omnes methodos hactenus notas laborem, etiam calculatori indefatigabili intolerabilem, requirerent.*

[It is in the nature of the problem that any *method will become more prolix as the numbers to which it is applied grow larger. Nevertheless, in the following methods the difficulties increase rather slowly, and numbers with seven, eight, or even more digits have been handled with success and speed beyond expectation, especially by the second method. The techniques that were previously known would require intolerable labor even for the most indefatigable calculator.]*

— Carl Friedrich Gauß, *Disquisitiones Arithmeticae* (1801)
English translation by A.A. Clarke (1965)

After much deliberation, the distinguished members of the international committee decided unanimously (when the Russian members went out for a caviar break) that since the Chinese emperor invented the method before anybody else had even been born, the method should be named after him. The Chinese emperor's name was Fast, so the method was called the Fast Fourier Transform.

— Thomas S. Huang, "How the fast Fourier transform got its name" (1971)

★2 Fast Fourier Transforms

2.1 Polynomials

In this lecture we'll talk about algorithms for manipulating *polynomials*: functions of one variable built from additions, subtractions, and multiplications (but no divisions). The most common representation for a polynomial $p(x)$ is as a sum of weighted powers of the variable x:

$$p(x) = \sum_{j=0}^{n} a_j x^j.$$

The numbers a_j are called the *coefficients* of the polynomial. The *degree* of the polynomial is the largest power of x whose coefficient is not equal to zero; in the example above, the degree is *at most* n. Any polynomial of degree n can be represented by an array $P[0\,..\,n]$ of $n+1$ coefficients, where $P[j]$ is the coefficient of the x^j term, and where $P[n] \neq 0$.

Here are three of the most common operations that are performed with polynomials:

- **Evaluate:** Give a polynomial p and a number x, compute the number $p(x)$.
- **Add:** Give two polynomials p and q, compute a polynomial $r = p + q$, so that $r(x) = p(x) + q(x)$ for all x. If p and q both have degree n, then their sum $p + q$ also has degree n.
- **Multiply:** Give two polynomials p and q, compute a polynomial $r = p \cdot q$, so that $r(x) = p(x) \cdot q(x)$ for all x. If p and q both have degree n, then their product $p \cdot q$ has degree $2n$.

We learned simple algorithms for all three of these operations in high-school algebra:

```
EVALUATE(P[0..n], x):
  X ← 1    ⟨⟨X = x^j⟩⟩
  y ← 0
  for j ← 0 to n
      y ← y + P[j] · X
      X ← X · x
  return y
```

```
ADD(P[0..n], Q[0..n]):
  for j ← 0 to n
      R[j] ← P[j] + Q[j]
  return R[0..n]
```

```
MULTIPLY(P[0..n], Q[0..m]):
  for j ← 0 to n + m
      R[j] ← 0
  for j ← 0 to n
      for k ← 0 to m
          R[j + k] ← R[j + k] + P[j] · Q[k]
  return R[0..n + m]
```

EVALUATE uses $O(n)$ arithmetic operations.[1] This is the best we can hope for, but we can cut the number of multiplications in half using *Horner's rule*:

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + xa_n)).$$

```
HORNER(P[0..n], x):
  y ← P[n]
  for i ← n − 1 downto 0
      y ← x · y + P[i]
  return y
```

The addition algorithm also runs in $O(n)$ time, and this is clearly the best we can do.

The multiplication algorithm, however, runs in $O(n^2)$ time. In the previous lecture, we saw a divide and conquer algorithm (due to Karatsuba) for multiplying two n-bit integers in only $O(n^{\lg 3})$ steps; precisely the same algorithm can be applied here. Even cleverer divide-and-conquer strategies lead to multiplication algorithms whose running times are arbitrarily close to linear—$O(n^{1+\varepsilon})$ for your favorite value $e > 0$—but with great cleverness comes great confusion. These algorithms are difficult to understand, even more difficult to implement correctly, and not worth the trouble in practice thanks to large constant factors.

2.2 Alternate Representations

Part of what makes multiplication so much harder than the other two operations is our input representation. Coefficients vectors are the most common representation for polynomials, but there are at least two other useful representations.

2.2.1 Roots

The Fundamental Theorem of Algebra states that every polynomial p of degree n has exactly n *roots* $r_1, r_2, \ldots r_n$ such that $p(r_j) = 0$ for all j. Some of these roots may be irrational; some of these roots may by complex; and some of these roots may be repeated. Despite these complications,

[1] I'm going to assume in this lecture that each arithmetic operation takes $O(1)$ time. This may not be true in practice; in fact, one of the most powerful applications of fast Fourier transforms is fast *integer* multiplication. The fastest algorithm currently known for multiplying two n-bit integers, published by Martin Fürer in 2007, uses $O(n \log n \, 2^{O(\log^* n)})$ bit operations and is based on fast Fourier transforms.

this theorem implies a unique representation of any polynomial of the form

$$p(x) = s \prod_{j=1}^{n} (x - r_j)$$

where the r_j's are the roots and s is a scale factor. Once again, to represent a polynomial of degree n, we need a list of $n+1$ numbers: one scale factor and n roots.

Given a polynomial in this root representation, we can clearly evaluate it in $O(n)$ time. Given two polynomials in root representation, we can easily multiply them in $O(n)$ time by multiplying their scale factors and just concatenating the two root sequences.

Unfortunately, if we want to add two polynomials in root representation, we're out of luck. There's essentially *no* correlation between the roots of p, the roots of q, and the roots of $p+q$. We could convert the polynomials to the more familiar coefficient representation first—this takes $O(n^2)$ time using the high-school algorithms—but there's no easy way to convert the answer back. In fact, for most polynomials of degree 5 or more in coefficient form, it's *impossible* to compute roots exactly.[2]

2.2.2 Samples

Our third representation for polynomials comes from a different consequence of the Fundamental Theorem of Algebra. Given a list of $n+1$ pairs $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$, there is *exactly one* polynomial p of degree n such that $p(x_j) = y_j$ for all j. This is just a generalization of the fact that any two points determine a unique line, because a line is the graph of a polynomial of degree 1. We say that the polynomial p ***interpolates*** the points (x_j, y_j). As long as we agree on the sample locations x_j in advance, we once again need exactly $n+1$ numbers to represent a polynomial of degree n.

Adding or multiplying two polynomials in this sample representation is easy, as long as they use the same sample locations x_j. To add the polynomials, just add their sample values. To multiply two polynomials, just multiply their sample values; however, if we're multiplying two polynomials of degree n, we must *start* with $2n+1$ sample values for each polynomial, because that's how many we need to uniquely represent their product. Both algorithms run in $O(n)$ time.

Unfortunately, evaluating a polynomial in this representation is no longer straightforward. The following formula, due to Lagrange, allows us to compute the value of any polynomial of degree n at any point, given a set of $n+1$ samples.

$$p(x) \;=\; \sum_{j=0}^{n-1} \left(\frac{y_j}{\prod_{k \neq j}(x_j - x_k)} \prod_{k \neq j} (x - x_k) \right)$$

Hopefully it's clear that formula actually describes a polynomial function of x, since each term in the sum is a scaled product of monomials. It's also not hard to verify that $p(x_j) = y_j$ for every index j; most of the terms of the sum vanish. As I mentioned earlier, the Fundamental Theorem of Algebra implies that p is *the only* polynomial that interpolates the points $\{(x_j, y_j)\}$. Lagrange's formula can be translated mechanically into an $O(n^2)$-time algorithm.

2.2.3 Summary

We find ourselves in the following frustrating situation. We have three representations for polynomials and three basic operations. Each representation allows us to almost trivially perform

[2]This is where numerical analysis comes from.

a different pair of operations in linear time, but the third takes at least quadratic time, if it can be done at all!

	evaluate	add	multiply
coefficients	$O(n)$	$O(n)$	$\boldsymbol{O(n^2)}$
roots + scale	$O(n)$	$\boldsymbol{\infty}$	$O(n)$
samples	$\boldsymbol{O(n^2)}$	$O(n)$	$O(n)$

2.3 Converting Between Representations

What we need are fast algorithms to convert quickly from one representation to another. That way, when we need to perform an operation that's hard for our default representation, we can switch to a different representation that makes the operation easy, perform that operation, and then switch back. This strategy immediately rules out the root representation, since (as I mentioned earlier) finding roots of polynomials is impossible in general, at least if we're interested in exact results.

So how do we convert from coefficients to samples and back? Clearly, once we choose our sample positions x_j, we can compute each sample value $y_j = p(x_j)$ in $O(n)$ time from the coefficients using Horner's rule. So we can convert a polynomial of degree n from coefficients to samples in $O(n^2)$ time. Lagrange's formula can be used to convert the sample representation back to the more familiar coefficient form. If we use the naïve algorithms for adding and multiplying polynomials (in coefficient form), this conversion takes $O(n^3)$ time.

We can improve the cubic running time by observing that *both* conversion problems boil down to computing the product of a matrix and a vector. The explanation will be slightly simpler if we assume the polynomial has degree $n-1$, so that n is the number of coefficients or samples. Fix a sequence $x_0, x_1, \ldots, x_{n-1}$ of sample *positions*, and let V be the $n \times n$ matrix where $v_{ij} = x_i^j$ (indexing rows and columns from 0 to $n-1$):

$$V = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix}.$$

The matrix V is called a ***Vandermonde matrix***. The vector of coefficients $\vec{a} = (a_0, a_1, \ldots, a_{n-1})$ and the vector of sample *values* $\vec{y} = (y_0, y_1, \ldots, y_{n-1})$ are related by the matrix equation

$$V\vec{a} = \vec{y},$$

or in more detail,

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}.$$

Given this formulation, we can clearly transform any coefficient vector $\vec{a}$ into the corresponding sample vector $\vec{y}$ in $O(n^2)$ time.

Conversely, if we know the sample values $\vec{y}$, we can recover the coefficients by solving a system of n linear equations in n unknowns, which can be done in $O(n^3)$ time using Gaussian elimination.[3] But we can speed this up by implicitly hard-coding the sample positions into the algorithm, To convert from samples to coefficients, we can simply multiply the sample vector by the inverse of V, again in $O(n^2)$ time.

$$\vec{a} = V^{-1}\vec{y}$$

Computing V^{-1} would take $O(n^3)$ time if we had to do it from scratch using Gaussian elimination, but because we fixed the set of sample positions in advance, the matrix V^{-1} can be hard-coded directly into the algorithm.[4]

So we can convert from coefficients to samples and back in $O(n^2)$ time. At first lance, this result seems pointless; we can already add, multiply, or evaluate directly in either representation in $O(n^2)$ time, so why bother? But there's a degree of freedom we haven't exploited—***We get to choose the sample positions!*** Our conversion algorithm is slow only because we're trying to be too general. If we choose a set of sample positions with the right recursive structure, we can perform this conversion more quickly.

2.4 Divide and Conquer

Any polynomial of degree at most $n-1$ can be expressed as a combination of two polynomials of degree at most $(n/2)-1$ as follows:

$$p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2).$$

The coefficients of p_{even} are just the even-degree coefficients of p, and the coefficients of p_{odd} are just the odd-degree coefficients of p. Thus, we can evaluate $p(x)$ by recursively evaluating $p_{\text{even}}(x^2)$ and $p_{\text{odd}}(x^2)$ and performing $O(1)$ additional arithmetic operations.

Now call a set X of n values ***collapsing*** if either of the following conditions holds:

- X has one element.
- The set $X^2 = \{x^2 \mid x \in X\}$ has exactly $n/2$ elements and is (recursively) collapsing.

Clearly the size of any collapsing set is a power of 2. Given a polynomial p of degree $n-1$, and a collapsing set X of size n, we can compute the set $\{p(x) \mid x \in X\}$ of sample values as follows:

1. Recursively compute $\{p_{\text{even}}(x^2) \mid x \in X\} = \{p_{\text{even}}(y) \mid y \in X^2\}$.
2. Recursively compute $\{p_{\text{odd}}(x^2) \mid x \in X\} = \{p_{\text{odd}}(y) \mid y \in X^2\}$.
3. For each $x \in X$, compute $p(x) = p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2)$.

The running time of this algorithm satisfies the familiar recurrence $T(n) = 2T(n/2) + \Theta(n)$, which as we all know solves to $T(n) = \Theta(n \log n)$.

[3]In fact, Lagrange's formula is just a special case of Cramer's rule for solving linear systems.

[4]Actually, it is possible to invert an $n \times n$ matrix in $o(n^3)$ time, using fast matrix multiplication algorithms that closely resemble Karatsuba's sub-quadratic divide-and-conquer algorithm for integer/polynomial multiplication. On the other hand, my numerical-analysis colleagues have reasonable cause to shoot me in the face for daring to suggest, even in passing, that anyone actually invert a matrix at all, ever.

Great! Now all we need is a sequence of arbitrarily large collapsing sets. The simplest method to construct such sets is just to invert the recursive definition: If X is a collapsible set of size n *that does not contain the number* 0, then $\sqrt{X} = \{\pm\sqrt{x} \mid x \in X\}$ is a collapsible set of size $2n$. This observation gives us an infinite sequence of collapsible sets, starting as follows:[5]

$$\begin{aligned}
X_1 &:= \{1\}\\
X_2 &:= \{1, -1\}\\
X_4 &:= \{1, -1, i, -i\}\\
X_8 &:= \left\{1, -1, i, -i, \frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i, -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i, -\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i\right\}
\end{aligned}$$

2.5 The Discrete Fourier Transform

For any n, the elements of X_n are called the ***complex nth roots of unity***; these are the roots of the polynomial $x^n - 1 = 0$. These n complex values are spaced exactly evenly around the unit circle in the complex plane. Every nth root of unity is a power of the *primitive* nth root

$$\boxed{\omega_n = e^{2\pi i/n} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}.}$$

A typical nth root of unity has the form

$$\omega_n^k = e^{(2\pi i/n)k} = \cos\left(\frac{2\pi}{n}k\right) + i\sin\left(\frac{2\pi}{n}k\right).$$

These complex numbers have several useful properties for any integers n and k:

- There are exactly n different nth roots of unity: $\omega_n^k = \omega_n^{k \bmod n}$.
- If n is even, then $\omega_n^{k+n/2} = -\omega_n^k$; in particular, $\omega_n^{n/2} = -\omega_n^0 = -1$.
- $1/\omega_n^k = \omega_n^{-k} = \overline{\omega_n^k} = (\overline{\omega_n})^k$, where the bar represents complex conjugation: $\overline{a+bi} = a - bi$
- $\omega_n = \omega_{kn}^k$. Thus, every nth root of unity is also a (kn)th root of unity.

These properties imply immediately that if n is a power of 2, then the set of all nth roots of unity is collapsible!

If we sample a polynomial of degree $n-1$ at the nth roots of unity, the resulting list of sample values is called the ***discrete Fourier transform*** of the polynomial (or more formally, of its coefficient vector). Thus, given an array $P[0\,..\,n-1]$ of coefficients, its discrete Fourier transform is the vector $P^*[0\,..\,n-1]$ defined as follows:

$$\boxed{P^*[j] := p(\omega_n^j) = \sum_{k=0}^{n-1} P[k]\cdot\omega_n^{jk}}$$

[5]In this lecture, i always represents the square root of -1. Computer scientists are used to thinking of i as an integer index into a sequence, an array, or a for-loop, but we obviously can't do that here. The physicist's habit of using $j = \sqrt{-1}$ just delays the problem (How do physicists write quaternions?), and typographical tricks like I or $\mathbf{i}$ or Mathematica's ⅈ are just stupid.

As we already observed, the fact that sets of roots of unity are collapsible implies that we can compute the discrete Fourier transform in $O(n \log n)$ time. The resulting algorithm, called the ***fast Fourier transform***, was popularized by Cooley and Tukey in 1965.[6] The algorithm assumes that n is a power of two; if necessary, we can just pad the coefficient vector with zeros.

```
FFT(P[0..n−1]):
  if n = 1
      return P

  for j ← 0 to n/2−1
      U[j] ← P[2j]
      V[j] ← P[2j+1]

  U* ← FFT(U[0..n/2−1])
  V* ← FFT(V[0..n/2−1])

  ω_n ← cos(2π/n) + i sin(2π/n)
  ω ← 1

  for j ← 0 to n/2−1
      P*[j]       ← U*[j] + ω · V*[j]
      P*[j + n/2] ← U*[j] − ω · V*[j]
      ω ← ω · ω_n

  return P*[0..n−1]
```

Minor variants of this divide-and-conquer algorithm were previously described by Good in 1958, by Thomas in 1948, by Danielson and Lánczos in 1942, by Stumpf in 1937, by Yates in 1932, and by Runge in 1903; some special cases were published even earlier by Everett in 1860, by Smith in 1846, and by Carlini in 1828. But the algorithm, in its full modern recursive generality, was first *used* by Gauss around 1805 for calculating the periodic orbits of asteroids from a finite number of observations. In fact, Gauss's recursive algorithm predates even Fourier's introduction of harmonic analysis by two years. So, of course, the algorithm is universally called the *Cooley-Tukey* algorithm. Gauss's work built on earlier research on trigonometric interpolation by Bernoulli, Lagrange, Clairaut, and Euler; in particular, the first explicit description of the discrete "Fourier" transform was published by Clairaut in 1754, more than half a century before Fourier's work. Hooray for Stigler's Law![7]

2.6 Inverting the FFT

We also need to recover the coefficients of the product from the new sample values. Recall that the transformation from coefficients to sample values is *linear*; the sample vector is the product of a Vandermonde matrix V and the coefficient vector. For the discrete Fourier transform, each

[6]Tukey apparently studied the algorithm to help detect Soviet nuclear tests without actually visiting Soviet nuclear facilities, by interpolating off-shore seismic readings. Without his rediscovery, the nuclear test ban treaty would never have been ratified, and we'd all be speaking Russian, or more likely, whatever language radioactive glass speaks.

[7]Lest anyone believe that Stigler's Law has treated Gauss unfairly, remember that "Gaussian elimination" was not discovered by Gauss; the algorithm was not even given that name until the mid-20th century! Elimination became the standard method for solving systems of linear equations in Europe in the early 1700s, when it appeared in an influential algebra textbook by Isaac Newton (published over his objections; he didn't want anyone to think it was his latest research). Although Newton apparently (and perhaps correctly) believed he had invented the elimination method, it actually appears in several earlier works, the oldest of which the eighth chapter of the Chinese manuscript *The Nine Chapters of the Mathematical Art*. The authors and precise age of the *Nine Chapters* are unknown, but commentary written by Liu Hui in 263CE claims that the text was already several centuries old. It was almost certainly *not* invented by a Chinese emperor named Fast.

entry in V is an nth root of unity; specifically,

$$\boxed{v_{jk} = \omega_n^{jk}}$$

for all integers j and k. Thus,

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

To invert the discrete Fourier transform, converting sample values back to coefficients, we just have to multiply P^* by the inverse matrix V^{-1}. The following amazing fact implies that this is almost the same as multiplying by V itself:

Claim: $V^{-1} = \overline{V}/n$

Proof: We just have to show that $M = V\overline{V}$ is the identity matrix scaled by a factor of n. We can compute a single entry in M as follows:

$$m_{jk} = \sum_{l=0}^{n-1} \omega_n^{jl} \cdot \overline{\omega_n}^{lk} = \sum_{l=0}^{n-1} \omega_n^{jl-lk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l$$

If $j = k$, then $\omega_n^{j-k} = \omega_n^0 = 1$, so

$$m_{jk} = \sum_{l=0}^{n-1} 1 = n,$$

and if $j \neq k$, we have a geometric series

$$m_{jk} = \sum_{l=0}^{n-1} (\omega_n^{j-k})^l = \frac{(\omega_n^{j-k})^n - 1}{\omega_n^{j-k} - 1} = \frac{(\omega_n^n)^{j-k} - 1}{\omega_n^{j-k} - 1} = \frac{1^{j-k} - 1}{\omega_n^{j-k} - 1} = 0. \qquad \square$$

In other words, if $W = V^{-1}$ then $w_{jk} = \overline{v_{jk}}/n = \overline{\omega_n^{jk}}/n = \omega_n^{-jk}/n$. What this means for us computer scientists is that any algorithm for computing the discrete Fourier transform can be easily modified to compute the inverse transform as well.

```
INVERSEFFT(P*[0..n−1]):
  if n = 1
      return P

  for j ← 0 to n/2−1
      U*[j] ← P*[2j]
      V*[j] ← P*[2j+1]

  U ← INVERSEFFT(U[0..n/2−1])
  V ← INVERSEFFT(V[0..n/2−1])

  ω̄_n ← cos(2π/n) − i sin(2π/n)
  ω ← 1

  for j ← 0 to n/2−1
      P[j]       ← 2(U[j] + ω·V[j])
      P[j + n/2] ← 2(U[j] − ω·V[j])
      ω ← ω·ω̄_n

  return P[0..n−1]
```

2.7 Fast Polynomial Multiplication

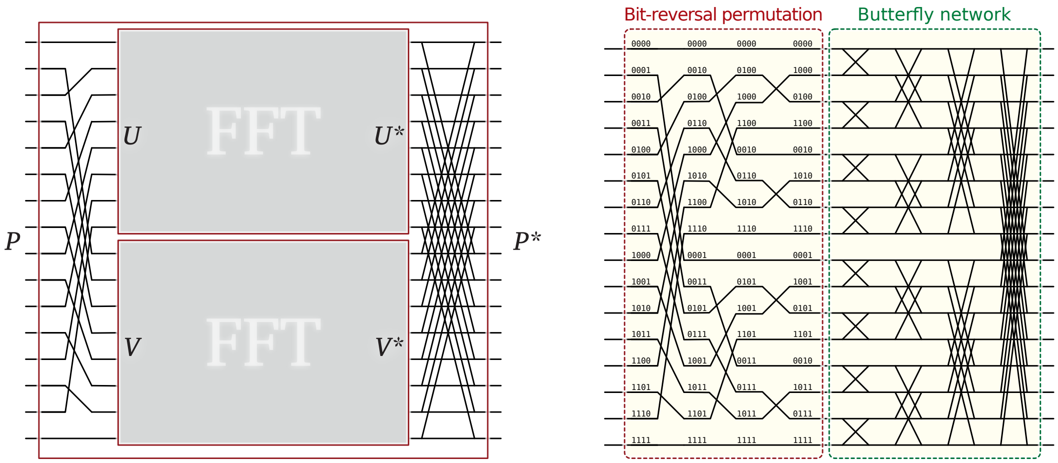
Finally, given two polynomials p and q, each represented by an array of coefficients, we can multiply them in $\Theta(n \log n)$ arithmetic operations as follows. First, pad the coefficient vectors and with zeros until the size is a power of two greater than or equal to the sum of the degrees. Then compute the DFTs of each coefficient vector, multiply the sample values one by one, and compute the inverse DFT of the resulting sample vector.

```
FFTMULTIPLY(P[0..n−1], Q[0..m−1]):
  ℓ ← ⌈lg(n+m)⌉
  for j ← n to 2^ℓ − 1
      P[j] ← 0
  for j ← m to 2^ℓ − 1
      Q[j] ← 0

  P* ← FFT(P)
  Q* ← FFT(Q)
  for j ← 0 to 2^ℓ − 1
      R*[j] ← P*[j]·Q*[j]
  return INVERSEFFT(R*)
```

2.8 Inside the FFT

FFTs are often implemented in hardware as circuits. To see the recursive structure of the circuit, let's connect the top-level inputs and outputs to the inputs and outputs of the recursive calls. On the left we split the input P into two recursive inputs U and V. On the right, we combine the outputs U^* and V^* to obtain the final output P^*.

If we expand this recursive structure completely, we see that the circuit splits naturally into two parts. The left half computes the *bit-reversal permutation* of the input. To find the position of $P[k]$ in this permutation, write k in binary, and then read the bits backward. For example, in an 8-element bit-reversal permutation, $P[3] = P[011_2]$ ends up in position $6 = 110_2$. The right half of the FFT circuit is a *butterfly network*. Butterfly networks are often used to route between processors in massively-parallel computers, because they allow any two processors to communicate in only $O(\log n)$ steps.



The recursive structure of the FFT algorithm.

Exercises

1. For any two sets X and Y of integers, the Minkowski sum $X + Y$ is the set of all pairwise sums $\{x + y \mid x \in X, y \in Y\}$.

   (a) Describe an analyze and algorithm to compute the number of elements in $X + Y$ in $O(n^2 \log n)$ time. *[Hint: The answer is **not** always n^2.]*

   (b) Describe and analyze an algorithm to compute the number of elements in $X + Y$ in $O(M \log M)$ time, where M is the largest absolute value of any element of $X \cup Y$. *[Hint: What's this lecture about?]*

2. Suppose we are given a bit string $B[1..n]$. A triple of distinct indices $1 \le i < j < k \le n$ is called a ***well-spaced triple*** in B if $B[i] = B[j] = B[k] = 1$ and $k - j = j - i$.

   (a) Describe a brute-force algorithm to determine whether B has a well-spaced triple in $O(n^2)$ time.

   (b) Describe an algorithm to determine whether B has a well-spaced triple in $O(n \log n)$ time. *[Hint: Hint.]*

   (c) Describe an algorithm to determine the *number* of well-spaced triples in B in $O(n \log n)$ time.

3. (a) Describe an algorithm that determines whether a given set of n integers contains two elements whose sum is zero, in $O(n \log n)$ time.

   (b) Describe an algorithm that determines whether a given set of n integers contains *three* elements whose sum is zero, in $O(n^2)$ time.

   (c) Now suppose the input set X contains only integers between $-10000n$ and $10000n$. Describe an algorithm that determines whether X contains three elements whose sum is zero, in $O(n \log n)$ time. *[Hint: Hint.]*

4. Describe an algorithm that applies the bit-reversal permutation to an array $A[1..n]$ in $O(n)$ time when n is a power of 2.

$$\boxed{\texttt{BIT-}} \iff \boxed{\texttt{BTI-}}$$
$$\boxed{\texttt{REVERSAL}} \iff \boxed{\texttt{RRVAESEL}}$$
$$\boxed{\texttt{BUTTERFLYNETWORK}} \iff \boxed{\texttt{BYEWTEFRUNROTTLK}}$$
$$\boxed{\texttt{THISISTHEBITREVERSALPERMUTATION!}} \iff \boxed{\texttt{TREUIPRIIAIATRVNHSBTSEEOSLTTHME!}}$$

5. The FFT algorithm we described in this lecture is limited to polynomials with 2^k coefficients for some integer k. Of course, we can always pad the coefficient vector with zeros to force it into this form, but this padding artificially inflates the input size, leading to a slower algorithm than necessary.

   Describe and analyze a similar DFT algorithm that works for polynomials with 3^k coefficients, by splitting the coefficient vector into three smaller vectors of length 3^{k-1}, recursively computing the DFT of each smaller vector, and correctly combining the results.

'Tis a lesson you should heed,
Try, try again;
If at first you don't succeed,
Try, try again;
Then your courage should appear,
For, if you will persevere,
You will conquer, never fear;
Try, try again.

— Thomas H. Palmer, *The Teacher's Manual: Being an Exposition of an Efficient and Economical System of Education Suited to the Wants of a Free People* (1840)

I dropped my dinner, and ran back to the laboratory. There, in my excitement, I tasted the contents of every beaker and evaporating dish on the table. Luckily for me, none contained any corrosive or poisonous liquid.

— Constantine Fahlberg on his discovery of saccharin, *Scientific American* (1886)

To resolve the question by a careful enumeration of solutions via trial and error, continued Gauss, would take only an hour or two. Apparently such inelegant work held little attraction for Gauss, for he does not seem to have carried it out, despite outlining in detail how to go about it.

— Paul Campbell, "Gauss and the Eight Queens Problem: A Study in Miniature of the Propagation of Historical Error" (1977)

3 Backtracking

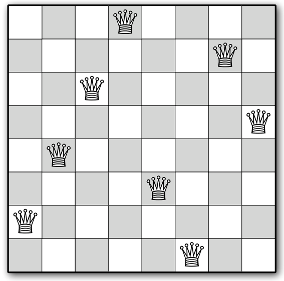
In this lecture, I want to describe another recursive algorithm strategy called ***backtracking***. A backtracking algorithm tries to build a solution to a computational problem incrementally. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it simply tries all possible options recursively.

3.1 *n* Queens

The prototypical backtracking problem is the classical ***n* Queens Problem**, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym "Schachfreund") for the standard 8×8 board and by François-Joseph Eustache Lionnet in 1869 for the more general $n \times n$ board. The problem is to place n queens on an $n \times n$ chessboard, so that no two queens can attack each other. For readers not familiar with the rules of chess, this means that no two queens are in the same row, column, or diagonal.

Obviously, in any solution to the n-Queens problem, there is exactly one queen in each row. So we will represent our possible solutions using an array $Q[1..n]$, where $Q[i]$ indicates which square in row i contains a queen, or 0 if no queen has yet been placed in row i. To find a solution, we put queens on the board row by row, starting at the top. A *partial* solution is an array $Q[1..n]$ whose first $r-1$ entries are positive and whose last $n-r+1$ entries are all zeros, for some integer r.

The following recursive algorithm, essentially due to Gauss (who called it "methodical groping"), recursively enumerates all complete n-queens solutions that are consistent with a given partial solution. The input parameter r is the first empty row. Thus, to compute all n-queens solutions with no restrictions, we would call RecursiveNQueens($Q[1..n], 1$).



One solution to the 8 queens problem, represented by the array [4,7,3,8,2,5,1,6]

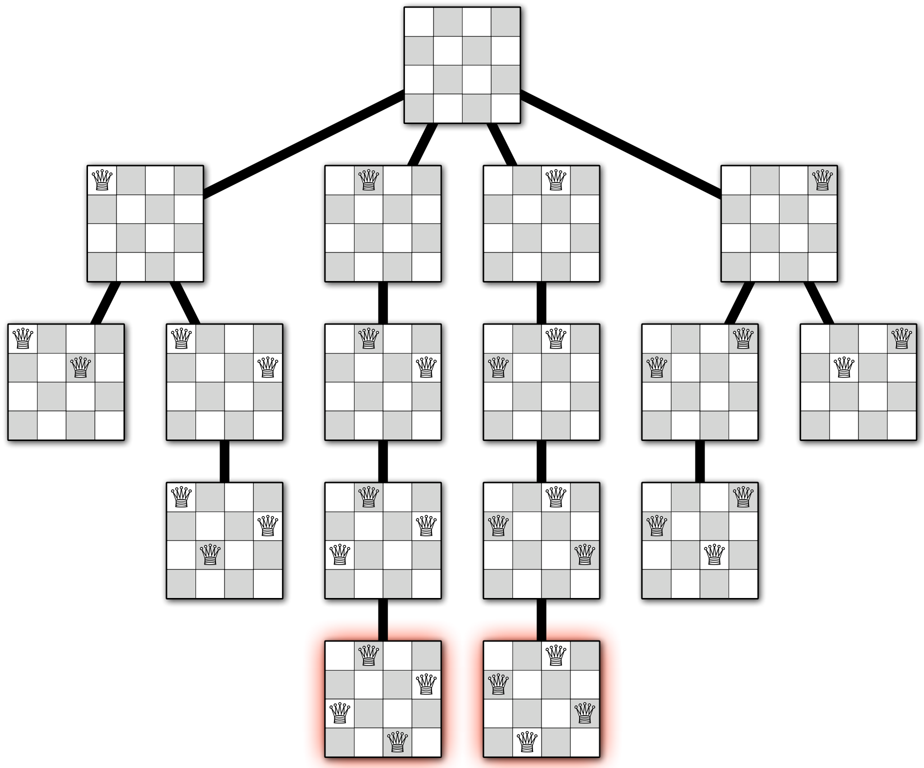
```
RecursiveNQueens(Q[1..n], r):
  if r = n + 1
      print Q
  else
      for j ← 1 to n
          legal ← True
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
                  legal ← False
          if legal
              Q[r] ← j
              RecursiveNQueens(Q[1..n], r + 1)
```

Like most recursive algorithms, the execution of a backtracking algorithm can be illustrated using a ***recursion tree***. The root of the recursion tree corresponds to the original invocation of the algorithm; edges in the tree correspond to recursive calls. A path from the root down to any node shows the history of a partial solution to the n-Queens problem, as queens are added to successive rows. The leaves correspond to partial solutions that cannot be extended, either because there is already a queen on every row, or because every position in the next empty row is in the same row, column, or diagonal as an existing queen. The backtracking algorithm simply performs a depth-first traversal of this tree.

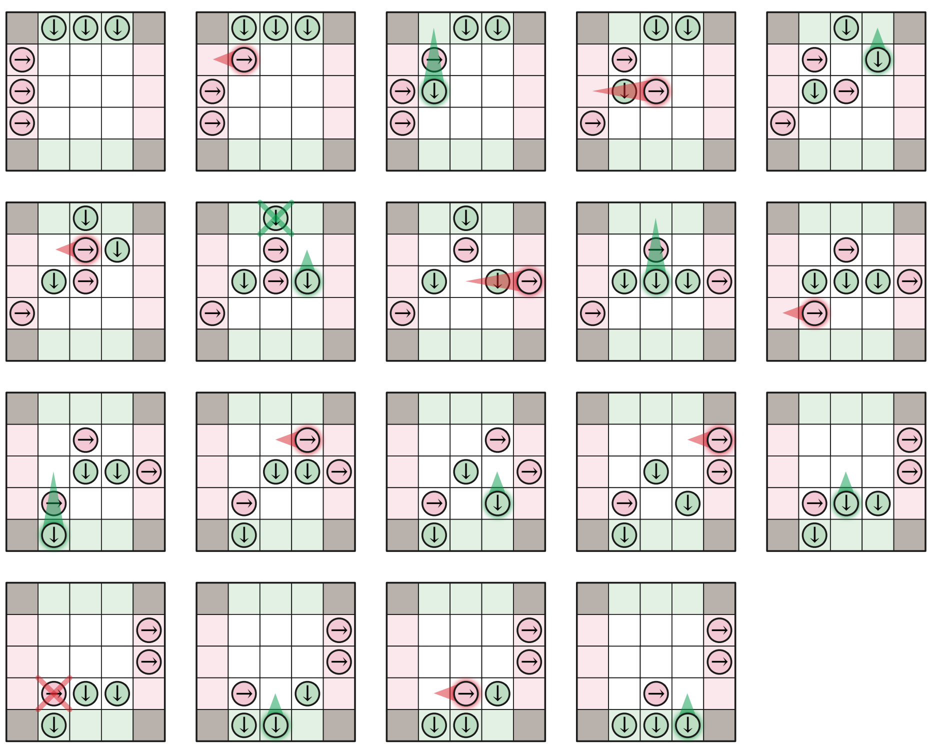
3.2 Game Trees

Consider the following simple two-player game played on an $n \times n$ square grid with a border of squares; let's call the players Horatio Fahlberg-Remsen and Vera Rebaudi.[1] Each player has n tokens that they move across the board from one side to the other. Horatio's tokens start in the left border, one in each row, and move to the right; symmetrically, Vera's tokens start in the top border, one in each column, and move down. The players alternate turns. In each of his turns, Horatio either *moves* one of his tokens one step to the right into an empty square, or *jumps* one of his tokens over exactly one of Vera's tokens into an empty square two steps to the right. However, if no legal moves or jumps are available, Horatio simply passes. Similarly, Vera either moves or jumps one of her tokens downward in each of her turns, unless no moves or jumps are possible. The first player to move all their tokens off the edge of the board wins.

[1] I don't know what this game is called, or even if I'm remembering the rules correctly; I learned it (or something like it) from Lenny Pitt, who recommended playing it with fake-sugar packets at restaurants. Constantin Fahlberg and Ira Remsen synthesized saccharin for the first time in 1878, while Fahlberg was a postdoc in Remsen's lab investigating coal tar derivatives. In 1900, Ovidio Rebaudi published the first chemical analysis of *ka'a he'ê*, a medicinal plant cultivated by the Guaraní for more than 1500 years, now more commonly known as *Stevia rebaudiana*.



The complete recursion tree for our algorithm for the 4 queens problem.

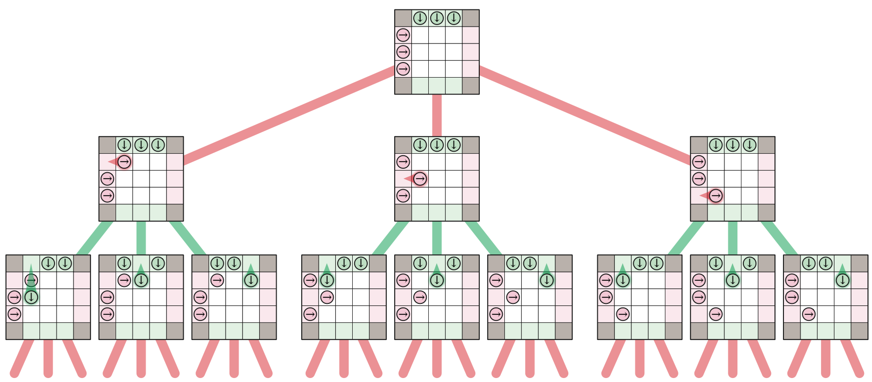


Vera wins the 3 × 3 fake-sugar-packet game.

We can use a simple backtracking algorithm to determine the best move for each player at each turn. The *state* of the game consists of the locations of all the pieces and the player whose turn it is. We recursively define a game state to be *good* or *bad* as follows:

- A game state is *bad* if all the opposing player's tokens have reached their goals.
- A game state is *good* if the current player can move to a state that is bad for the opposing player.
- A configuration is *bad* if every move leads to a state that is good for the opposing player.

This recursive definition immediately suggests a recursive backtracking algorithm to determine whether a given state of the game is good or bad. Moreover, for any good state, the backtracking algorithm finds a move leading to a bad state for the opposing player. Thus, by induction, any player that finds the game in a good state on their turn can win the game, even if their opponent plays perfectly; on the other hand, starting from a bad state, a player can win only if their opponent makes a mistake.



The first two levels of the fake-sugar-packet game tree.

All computer game players are ultimately based on this simple backtracking strategy. However, since most games have an enormous number of states, it is not possible to traverse the entire game tree in practice. Instead, game programs employ other heuristics[2] to *prune* the game tree, by ignoring states that are obviously good or bad (or at least obviously better or worse that other states), and/or by cutting off the tree at a certain depth (or *ply*) and using a more efficient heuristic to evaluate the leaves.

3.3 Subset Sum

Let's consider a more complicated problem, called SUBSETSUM: Given a set X of positive integers and *target* integer T, is there a subset of elements in X that add up to T? Notice that there can be more than one such subset. For example, if $X = \{8, 6, 7, 5, 3, 10, 9\}$ and $T = 15$, the answer is TRUE, thanks to the subsets $\{8, 7\}$ or $\{7, 5, 3\}$ or $\{6, 9\}$ or $\{5, 10\}$. On the other hand, if $X = \{11, 6, 5, 1, 7, 13, 12\}$ and $T = 15$, the answer is FALSE.

There are two trivial cases. If the target value T is zero, then we can immediately return TRUE, because empty set is a subset of *every* set X, and the elements of the empty set add up to zero.[3] On the other hand, if $T < 0$, or if $T \neq 0$ but the set X is empty, then we can immediately return FALSE.

[2]A heuristic is an algorithm that doesn't work.

[3]There's no base case like the vacuous base case!

For the general case, consider an arbitrary element $x \in X$. (We've already handled the case where X is empty.) There is a subset of X that sums to T if and only if one of the following statements is true:

- There is a subset of X that *includes* x and whose sum is T.
- There is a subset of X that *excludes* x and whose sum is T.

In the first case, there must be a subset of $X \setminus \{x\}$ that sums to $T - x$; in the second case, there must be a subset of $X \setminus \{x\}$ that sums to T. So we can solve SubsetSum(X, T) by reducing it to two simpler instances: SubsetSum$(X \setminus \{x\}, T - x)$ and SubsetSum$(X \setminus \{x\}, T)$. Here's how the resulting recusive algorithm might look if X is stored in an array.

```
SubsetSum(X[1..n], T):
  if T = 0
      return True
  else if T < 0 or n = 0
      return False
  else
      return (SubsetSum(X[1..n−1], T) ∨ SubsetSum(X[1..n−1], T − X[n]))
```

Proving this algorithm correct is a straightforward exercise in induction. If $T = 0$, then the elements of the empty subset sum to T, so True is the correct output. Otherwise, if T is negative or the set X is empty, then no subset of X sums to T, so False is the correct output. Otherwise, if there is a subset that sums to T, then either it contains $X[n]$ or it doesn't, and the Recursion Fairy correctly checks for each of those possibilities. Done.

The running time $T(n)$ clearly satisfies the recurrence $T(n) \le 2T(n-1) + O(1)$, which we can solve using either recursion trees or annihilators (or just guessing) to obtain the upper bound $T(n) = O(2^n)$. In the worst case, the recursion tree for this algorithm is a complete binary tree with depth n.

Here is a similar recursive algorithm that actually *constructs* a subset of X that sums to T, if one exists. This algorithm also runs in $O(2^n)$ time.

```
ConstructSubset(X[1..n], T):
  if T = 0
      return ∅
  if T < 0 or n = 0
      return None

  Y ← ConstructSubset(X[1..n−1], T)
  if Y ≠ None
      return Y

  Y ← ConstructSubset(X[1..n−1], T − X[n])
  if Y ≠ None
      return Y ∪ {X[n]}

  return None
```

3.4 The General Pattern

★★★

Find a small choice whose correct answer would reduce the problem size. For each possible answer, temporarily adopt that choice and recurse. (Don't try to be clever about which choices to try; just try them all.) The recursive subproblem is often more general than the original target problem; in each recursive subproblem, we must consider *only* solutions that are consistent with the choices we have already made.

3.5 NFA acceptance

Recall that a nondeterministic finite-state automaton, or NFA, can be described as a directed graph, whose edges are called *states* and whose edges have *labels* drawn from a finite set Σ called the *alphabet*. Every NFA has a designated *start* state and a subset of *accepting* states. Any walk in this graph has a label, which is a string formed by concatenating the labels of the edges in the walk. A string w is *accepted* by an NFA if and only if there is a walk from the start state to one of the accepting states whose label is w.

More formally (or at least, more symbolically), an NFA consists of a finite set Q of states, a start state $s \in Q$, a set of accepting states $A \subseteq Q$, and a transition function $\delta: Q \times \Sigma \to 2^Q$. We recursively extend the transition function to strings by defining

$$\delta^*(q,w) = \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \delta(q,a)} \delta^*(r,x) & \text{if } w = ax. \end{cases}$$

The NFA accepts string w if and only if the set $\delta^*(s,w)$ contains at least one accepting state.

We can express this acceptance criterion more directly as follows. We define a boolean function *Accepts?*(q,w), which is TRUE if the NFA would accept string w if we started in state q, and FALSE otherwise. This function has the following recursive definition:

$$\mathit{Accepts?}(q,w) := \begin{cases} \textsc{True} & \text{if } w = \varepsilon \text{ and } q \in A \\ \textsc{False} & \text{if } w = \varepsilon \text{ and } q \in A \\ \displaystyle\bigvee_{r \in \delta(q,a)} \mathit{Accepts?}(r,x) & \text{if } w = ax \end{cases}$$

The NFA accepts w if and only if *Accepts?*$(s,w) = \textsc{True}$.

In the magical world of non-determinism, we can imagine that the NFA always magically makes the right decision when faces with multiple transitions, or perhaps spawns off an independent parallel thread for each possible choice. Alas, real computers are neither clairvoyant nor (despite the increasing use of multiple cores) infinitely parallel. To simulate the NFA's behavior directly, we must recursively explore the consequences of each choice explicitly.

The recursive definition of *Accepts?* translates directly into the following recursive backtracking algorithm. Here, the transition function δ and the accepting states A are represented as global boolean arrays, where $\delta[q,a,r] = \textsc{True}$ if and only if $r \in \delta(q,a)$, and $A[q] = \textsc{True}$ if and only if $q \in A$.

```
ACCEPTS?(q, w[1..n]):
  if n = 0
      return A[q]
  for all states r
      if δ[q, w[1], r] and ACCEPTS?(r, w[2..n])
          return TRUE
  return FALSE
```

To determine whether the NFA accepts a string w, we call ACCEPTS?(δ, A, s, w).

The running time of this algorithm satisfies the recursive inequailty $T(n) \le O(|Q|) \cdot T(n-1)$, which immediately implies that $T(n) = O(|Q|^n)$.

3.6 Longest Increasing Subsequence

Now suppose we are given a sequence of integers, and we want to find the longest subsequence whose elements are in increasing order. More concretely, the input is an array $A[1..n]$ of integers, and we want to find the longest sequence of indices $1 \le i_1 < i_2 < \cdots i_k \le n$ such that $A[i_j] < A[i_{j+1}]$ for all j.

To derive a recursive algorithm for this problem, we start with a recursive definition of the kinds of objects we're playing with: sequences and subsequences.

> A *sequence of integers* is either empty
> or an integer followed by a sequence of integers.

This definition suggests the following strategy for devising a recursive algorithm. If the input sequence is empty, there's nothing to do. Otherwise, we only need to figure out what to do with the first element of the input sequence; the Recursion Fairy will take care of everything else. We can formalize this strategy somewhat by giving a recursive definition of subsequence (using array notation to represent sequences):

> The only *subsequence* of the empty sequence is the empty sequence.
>
> A *subsequence* of $A[1..n]$ is either a subsequence of $A[2..n]$
> or $A[1]$ followed by a subsequence of $A[2..n]$.

We're not just looking for just *any* subsequence, but a *longest* subsequence with the property that elements are in *increasing* order. So let's try to add those two conditions to our definition. (I'll omit the familiar vacuous base case.)

> The *LIS* of $A[1..n]$ is
> either the LIS of $A[2..n]$
> or $A[1]$ followed by the LIS of $A[2..n]$ *with elements larger than* $A[1]$,
> *whichever is longer*.

This definition is correct, but it's not quite recursive—we're defining the object 'longest increasing subsequence' in terms of the slightly *different* object 'longest increasing subsequence with elements larger than x', which we haven't properly defined yet. Fortunately, this second object has a very similar recursive definition. (Again, I'm omitting the vacuous base case.)

> If $A[1] \le x$, the LIS of $A[1..n]$ with elements larger than x is
> the LIS of $A[2..n]$ with elements larger than x.
>
> Otherwise, the LIS of $A[1..n]$ with elements larger than x is
> either the LIS of $A[2..n]$ with elements larger than x
> or $A[1]$ followed by the LIS of $A[2..n]$ with elements larger than $A[1]$,
> whichever is longer.

The longest increasing subsequence without restrictions can now be redefined as the longest increasing subsequence with elements larger than $-\infty$. Rewriting this recursive definition into pseudocode gives us the following recursive algorithm.

```
LIS(A[1..n]):
  return LISBIGGER(−∞, A[1..n])
```

```
LISBIGGER(prev, A[1..n]):
  if n = 0
      return 0
  else
      max ← LISBIGGER(prev, A[2..n])
      if A[1] > prev
          L ← 1 + LISBIGGER(A[1], A[2..n])
          if L > max
              max ← L
      return max
```

The running time of this algorithm satisfies the recurrence $T(n) \leq 2T(n-1) + O(1)$, which as usual implies that $T(n) = O(2^n)$. We really shouldn't be surprised by this running time; in the worst case, the algorithm examines each of the 2^n subsequences of the input array.

The following alternative strategy avoids defining a new object with the "larger than x" constraint. We still only have to decide whether to include or exclude the first element $A[1]$. We consider the case where $A[1]$ is excluded exactly the same way, but to consider the case where $A[1]$ is included, we remove any elements of $A[2..n]$ that are larger than $A[1]$ *before* we recurse. This new strategy gives us the following algorithm:

```
FILTER(A[1..n], x):
  j ← 1
  for i ← 1 to n
      if A[i] > x
          B[j] ← A[i];  j ← j + 1
  return B[1..j]
```

```
LIS(A[1..n]):
  if n = 0
      return 0
  else
      max ← LIS(prev, A[2..n])
      L ← 1 + LIS(A[1], FILTER(A[2..n], A[1]))
      if L > max
          max ← L
      return max
```

The FILTER subroutine clearly runs in $O(n)$ time, so the running time of LIS satisfies the recurrence $T(n) \leq 2T(n-1) + O(n)$, which solves to $T(n) \leq O(2^n)$ by the annihilator method. This upper bound pessimistically assumes that FILTER never actually removes any elements; indeed, if the input sequence is sorted in increasing order, this assumption is correct.

3.7 Optimal Binary Search Trees

★★★

Retire this example? It's not a *bad* example, exactly—it's infinitely better than the execrable matrix-chain multiplication problem from Aho, Hopcroft, and Ullman—but it's not the best *first* example of tree-like backtracking. Minimum-ink triangulation of convex polygons is both more intuitive (geometry FTW!) and structurally equivalent. CFG parsing and regular expression matching (really just a special case of parsing) have similar recursive structure, but are a bit more complicated.

Our next example combines recursive backtracking with the divide-and-conquer strategy. Recall that the running time for a successful search in a binary search tree is proportional to the number of ancestors of the target node.[4] As a result, the worst-case search time is proportional to the depth of the tree. Thus, to minimize the worst-case search time, the height of the tree should be as small as possible; by this metric, the ideal tree is perfectly balanced.

[4]An *ancestor* of a node v is either the node itself or an ancestor of the parent of v. A *proper* ancestor of v is either the parent of v or a proper ancestor of the parent of v.

In many applications of binary search trees, however, it is more important to minimize the total cost of several searches rather than the worst-case cost of a single search. If x is a more 'popular' search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y, even if that means increasing the overall depth of the tree. A perfectly balanced tree is *not* the best choice if some items are significantly more popular than others. In fact, a totally unbalanced tree of depth $\Omega(n)$ might actually be the best choice!

This situation suggests the following problem. Suppose we are given a sorted array of ***keys*** $A[1..n]$ and an array of corresponding ***access frequencies*** $f[1..n]$. Our task is to build the binary search tree that minimizes the *total* search time, assuming that there will be exactly $f[i]$ searches for each key $A[i]$.

Before we think about how to solve this problem, we should first come up with a good recursive definition of the function we are trying to optimize! Suppose we are also given a binary search tree T with n nodes. Let v_i denote the node that stores $A[i]$, and let r be the index of the root node. Ignoring constant factors, the cost of searching for $A[i]$ is the number of nodes on the path from the root v_r to v_i. Thus, the total cost of performing all the binary searches is given by the following expression:

$$\mathit{Cost}(T, f[1..n]) = \sum_{i=1}^{n} f[i] \cdot \#\text{nodes between } v_r \text{ and } v_i$$

Every search path includes the root node v_r. If $i < r$, then all other nodes on the search path to v_i are in the left subtree; similarly, if $i > r$, all other nodes on the search path to v_i are in the right subtree. Thus, we can partition the cost function into three parts as follows:

$$\begin{aligned}\mathit{Cost}(T, f[1..n]) = &\sum_{i=1}^{r-1} f[i] \cdot \#\text{nodes between } \mathit{left}(v_r) \text{ and } v_i \\ &+ \sum_{i=1}^{n} f[i] \\ &+ \sum_{i=r+1}^{n} f[i] \cdot \#\text{nodes between } \mathit{right}(v_r) \text{ and } v_i\end{aligned}$$

Now the first and third summations look exactly like our original expression (*) for $\mathit{Cost}(T, f[1..n])$. Simple substitution gives us our recursive definition for *Cost*:

$$\boxed{\mathit{Cost}(T, f[1..n]) = \mathit{Cost}(\mathit{left}(T), f[1..r-1]) \; + \; \sum_{i=1}^{n} f[i] \; + \; \mathit{Cost}(\mathit{right}(T), f[r+1..n])}$$

The base case for this recurrence is, as usual, $n = 0$; the cost of performing no searches in the empty tree is zero.

Now our task is to compute the tree T_{opt} that minimizes this cost function. Suppose we somehow magically knew that the root of T_{opt} is v_r. Then the recursive definition of $\mathit{Cost}(T, f)$ immediately implies that the left subtree left(T_{opt}) must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$. Similarly, the right subtree right(T_{opt}) must be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$. **Once we choose the correct key to store at the root, the Recursion Fairy automatically constructs the rest of the optimal tree.** More formally, let $\mathit{OptCost}(f[1..n])$ denote the total cost of the optimal search tree for the given frequency counts. We immediately have the following recursive definition.

$$\boxed{OptCost(f[1..n]) = \min_{1\le r\le n}\left\{OptCost(f[1..r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; OptCost(f[r+1..n])\right\}}$$

Again, the base case is $OptCost(f[1..0]) = 0$; the best way to organize no keys, which we will plan to search zero times, is by storing them in the empty tree!

This recursive definition can be translated mechanically into a recursive algorithm, whose running time $T(n)$ satisfies the recurrence

$$T(n) = \Theta(n) + \sum_{k=1}^{n}\big(T(k-1) + T(n-k)\big).$$

The $\Theta(n)$ term comes from computing the total number of searches $\sum_{i=1}^{n} f[i]$.

Yeah, that's one ugly recurrence, but it's actually easier to solve than it looks. To transform it into a more familiar form, we regroup and collect identical terms, subtract the recurrence for $T(n-1)$ to get rid of the summation, and then regroup again.

$$\begin{aligned} T(n) &= \Theta(n) + 2\sum_{k=0}^{n-1} T(k) \\ T(n-1) &= \Theta(n-1) + 2\sum_{k=0}^{n-2} T(k) \\ T(n) - T(n-1) &= \Theta(1) + 2T(n-1) \\ T(n) &= 3T(n-1) + \Theta(1) \end{aligned}$$

The solution $\boldsymbol{T(n) = \Theta(3^n)}$ now follows from the annihilator method.

Let me emphasize that this recursive algorithm does *not* examine all possible binary search trees. The number of binary search trees with n nodes satisfies the recurrence

$$N(n) = \sum_{r=1}^{n-1}\big(N(r-1)\cdot N(n-r)\big),$$

which has the closed-from solution $N(n) = \Theta(4^n/\sqrt{n})$. Our algorithm saves considerable time by searching *independently* for the optimal left and right subtrees. A full enumeration of binary search trees would consider all possible *pairings* of left and right subtrees; hence the product in the recurrence for $N(n)$.

*3.8 CFG Parsing

Our final example is the ***parsing*** problem for context-free languages. Given a string w and a context-free grammar G, does w belong to the language generated by G? Recall that a context-free grammar over the alphabet Σ consists of a finite set Γ of *non-terminals* (disjoint from Σ) and a finite set of *production rules* of the form $A \to w$, where A is a nonterminal and w is a string over $\Sigma \cup \Gamma$.

Real-world applications of parsing normally require more information than just a single bit. For example, compilers require parsers that output a *parse tree* of the input code; some natural language applications require the *number* of distinct parse trees for a given string; others assign probabilities to the production rules and then ask for the *most likely* parse tree for a given string. However, once we have an algorithm for the decision problem, it it not hard to extend it to answer these more general questions.

We define a boolean function *Generates?*: $\Sigma^* \times \Gamma$, where *Generates?*$(A, x) =$ True if and only if x can be derived from A. At first glance, it seems that the production rules of the CFL immediately give us a (rather complicated) recursive definition for this function; unfortunately, there are a few problems.

- Consider the context-free grammar $S \to \varepsilon \mid SS \mid \mathtt{(}S\mathtt{)}$ that generates all properly balanced strings of parentheses. The "obvious" recursive algorithm for *Generates?*(S, w) would recursively check whether $x \in L(S)$ and $y \in L(S)$, for every possible partition $w = x \bullet y$, *including the trivial partition* $w = \varepsilon \bullet w$. It follows that *Generates?*(S, w) calls itself, leading to an infinite loop.

- Consider another grammar that includes the productions $S \to A$, $A \to B$, and $B \to S$, possibly among others. The "obvious" recursive algorithm for *Generates?*(S, w) must call *Generates?*(A, w), which calls *Generates?*(B, w), which calls *Generates?*(S, w), and we are again in an infinite loop.

To avoid these issues, we will make the simplifying assumption that our input grammar is in ***Chomsky normal form***. Recall that a CNF grammar has the following special structure:

- The starting non-terminal S does not appear on the right side of any production rule.
- The starting non-terminal S *may* have the production rule $S \to \varepsilon$.
- Every other production rule has the form $A \to BC$ (two non-terminals) or $A \to a$ (one terminal).

In an earlier lecture note, I describe an algorithm to convert any context-free grammar into Chomsky normal form. Unfortunately, I still haven't introduced all the algorithmic tools you might need to really understand that algorithm; fortunately, for purposes of this note, it's enough to know that such an algorithm exists.

With this simplifying assumption in place, the function *Generates?* now has a relatively straightforward recursive definition.

$$
\textit{Generates?}(A, x) = \begin{cases} \textsc{True} & \text{if } |x| \le 1 \text{ and } A \to x \\ \textsc{False} & \text{if } |x| \le 1 \text{ and } A \not\to x \\ \displaystyle\bigvee_{A \to BC} \bigvee_{y \bullet z = x} \textit{Generates?}(B, y) \wedge \textit{Generates?}(C, z) & \text{otherwise} \end{cases}
$$

The first two cases take care of terminal productions $A \to a$ and the ε-production $S \to \varepsilon$ (if the grammar contains it). The notation $A \not\to x$ means that $A \to x$ is *not* a production rule in the given grammar. In the generic case, for all production rules $A \to BC$, and for all ways of splitting x into a *non-empty* prefix y and a *non-empty* suffix z, we recursively check whether $y \in L(B)$ and $z \in L(C)$. Because we pass strictly smaller strings in the second argument of these recursive calls, every branch of the recursion tree eventually terminates.

This recursive definition translates mechanically into a recursive algorithm. To bound the precise running time of this algorithm, we need to solve a system of mutually recursive functions, one for each non-terminal, where the function for each non-terminal A depends on the number of production rules $A \to BC$. For the sake of illustration, suppose each non-terminal has at most ℓ non-terminating production rules. Then the running time can be bounded by the recurrence

$$T(n) = \Theta(n) + \ell \cdot \sum_{k=1}^{n-1} \big(T(k) + T(n-k)\big) = \Theta(n) + 2\ell \cdot \sum_{k=1}^{n-1} T(k)$$

where the $\Theta(n)$ term accounts for the overhead of splitting the input string in n different ways. The same approach as our analysis of optimal binary search trees (difference transformation followed by annihilators) implies the solution $\boldsymbol{T(n) = \Theta((2\ell + 1)^n)}$.

Exercises

1. (a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of A and B is both a subsequence of A and a subsequence of B. Give a simple recursive definition for the function *lcs*(A, B), which gives the length of the *longest* common subsequence of A and B.

   (b) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common supersequence* of A and B is another sequence that contains both A and B as subsequences. Give a simple recursive definition for the function *scs*(A, B), which gives the length of the *shortest* common supersequence of A and B.

   (c) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Give a simple recursive definition for the function *los*(A), which gives the length of the longest oscillating subsequence of an arbitrary array A of integers.

   (d) Give a simple recursive definition for the function *sos*(A), which gives the length of the shortest oscillating supersequence of an arbitrary array A of integers.

   (e) Call a sequence $X[1..n]$ *accelerating* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Give a simple recursive definition for the function *lxs*(A), which gives the length of the longest accelerating subsequence of an arbitrary array A of integers.

For more backtracking exercises, see the next two lecture notes!

Wouldn't the sentence "I want to put a hyphen between the words Fish and And and And and Chips in my Fish-And-Chips sign." have been clearer if quotation marks had been placed before Fish, and between Fish and and, and and and And, and And and and, and and and And, and And and and, and and and Chips, as well as after Chips?[1]

— Martin Gardner, *Aha! Insight* (1978)

★4 Efficient Exponential-Time Algorithms

In another lecture note, we discuss the class of ***NP-hard*** problems. For every problem in this class, the fastest algorithm anyone knows has an exponential running time. Moreover, there is *very* strong evidence (but alas, no proof) that it is *impossible* to solve any NP-hard problem in less than exponential time—it's not that we're all stupid; the problems really are that hard! Unfortunately, an enormous number of problems that arise in practice are NP-hard; for some of these problems, even *approximating* the right answer is NP-hard.

Suppose we absolutely have to find the exact solution to some NP-hard problem. A polynomial-time algorithm is almost certainly out of the question; the best running time we can hope for is exponential. But *which* exponential? An algorithm that runs in $O(1.5^n)$ time, while still unusable for large problems, is still significantly better than an algorithm that runs in $O(2^n)$ time!

For most NP-hard problems, the only approach that is guaranteed to find an optimal solution is recursive backtracking. The most straightforward version of this approach is to recursively generate *all* possible solutions and check each one: all satisfying assignments, or all vertex colorings, or all subsets, or all permutations, or whatever. However, most NP-hard problems have some additional structure that allows us to prune away most of the branches of the recursion tree, thereby drastically reducing the running time.

4.1 3SAT

Let's consider the mother of all NP-hard problems: 3SAT. Given a boolean formula in conjunctive normal form, with at most three literals in each clause, our task is to determine whether any assignment of values of the variables makes the formula true. Yes, this problem is NP-hard, which means that a polynomial algorithm is almost certainly impossible. Too bad; we have to solve the problem anyway.

The trivial solution is to try every possible assignment. We'll evaluate the running time of our 3SAT algorithms in terms of the number of variables in the formula, so let's call that n. Provided any clause appears in our input formula at most once—a condition that we can easily enforce in polynomial time—the overall input size is $O(n^3)$. There are 2^n possible assignments, and we can evaluate each assignment in $O(n^3)$ time, so the overall running time is $O(2^n n^3)$.

[1]If you ever decide to read this sentence out loud, be sure to pause briefly between 'Fish and and' and 'and and and And', 'and and and And' and 'and And and and', 'and And and and' and 'and and and And', 'and and and And' and 'and And and and', and 'and And and and' and 'and and and Chips'!

Did you notice the punctuation I carefully inserted between 'Fish and and' and 'and', 'and' and 'and and and And', 'and and and And' and 'and and and And', 'and and and And' and 'and', 'and' and 'and And and and', 'and And and and' and 'and And and and', 'and And and and' and 'and', 'and' and 'and and and And', 'and and and And' and 'and and and And', 'and and and And' and 'and', 'and' and 'and And and and', 'and And and and' and 'and', 'and' and 'and And and and', 'and And and and' and 'and', and 'and' and 'and and and Chips'?

Since polynomial factors like n^3 are essentially noise when the overall running time is exponential, from now on I'll use $\text{poly}(n)$ to represent some arbitrary polynomial in n; in other words, $\text{poly}(n) = n^{O(1)}$. For example, the trivial algorithm for 3SAT runs in time $O(2^n \text{poly}(n))$.

We can make this algorithm smarter by exploiting the special recursive structure of 3CNF formulas:

A 3CNF formula is either nothing or a clause with three literals $\wedge$ a 3CNF formula

Suppose we want to decide whether some 3CNF formula Φ with n variables is satisfiable. Of course this is trivial if Φ is the empty formula, so suppose

$$\Phi = (x \vee y \vee z) \wedge \Phi'$$

for some literals x, y, z and some 3CNF formula Φ'. By distributing the $\wedge$ across the $\vee$s, we can rewrite Φ as follows:

$$\Phi = (x \wedge \Phi') \vee (y \wedge \Phi') \vee (z \wedge \Phi')$$

For any boolean formula Ψ and any literal x, let $\Psi|x$ (pronounced "sigh given eks") denote the simpler boolean formula obtained by assuming x is true. It's not hard to prove by induction (hint, hint) that $x \wedge \Psi = x \wedge \Psi|x$, which implies that

$$\Phi = (x \wedge \Phi'|x) \vee (y \wedge \Phi'|y) \vee (z \wedge \Phi'|z).$$

Thus, in any satisfying assignment for Φ, either x is true and $\Phi'|x$ is satisfiable, or y is true and $\Phi'|y$ is satisfiable, or z is true and $\Phi'|z$ is satisfiable. Each of the smaller formulas has at most $n-1$ variables. If we recursively check all three possibilities, we get the running time recurrence

$$T(n) \leq 3T(n-1) + \text{poly}(n),$$

whose solution is $O(3^n \text{poly}(n))$. So we've actually done *worse!*

But these three recursive cases are not mutually exclusive! If $\Phi'|x$ is *not* satisfiable, then x *must* be false in any satisfying assignment for Φ. So instead of recursively checking $\Phi'|y$ in the second step, we can check the even simpler formula $\Phi'|\bar{x}y$. Similarly, if $\Phi'|\bar{x}y$ is not satisfiable, then we know that y must be false in any satisfying assignment, so we can recursively check $\Phi'|\bar{x}\bar{y}z$ in the third step.

```
3SAT(Φ):
  if Φ = ∅
      return TRUE
  (x ∨ y ∨ z) ∧ Φ′ ← Φ
  if 3SAT(Φ|x)
      return TRUE
  if 3SAT(Φ|x̄y)
      return TRUE
  return 3SAT(Φ|x̄ȳz)
```

The running time off this algorithm obeys the recurrence

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \text{poly}(n),$$

where poly(n) denotes the polynomial time required to simplify boolean formulas, handle control flow, move stuff into and out of the recursion stack, and so on. The annihilator method gives us the solution

$$T(n) = O(\lambda^n \operatorname{poly}(n)) = \boxed{O(1.83928675522^n)}$$

where $\lambda \approx 1.83928675521\ldots$ is the largest root of the characteristic polynomial $r^3 - r^2 - r - 1$. (Notice that we cleverly eliminated the polynomial noise by increasing the base of the exponent ever so slightly.)

We can improve this algorithm further by eliminating *pure* literals from the formula before recursing. A literal x is *pure* in if it appears in the formula Φ but its negation $\bar{x}$ does not. It's not hard to prove (hint, hint) that if Φ has a satisfying assignment, then it has a satisfying assignment where every pure literal is true. If $\Phi = (x \vee y \vee z) \wedge \Phi'$ has no pure literals, then some in Φ contains the literal $\bar{x}$, so we can write

$$\Phi = (x \vee y \vee z) \wedge (\bar{x} \vee u \vee v) \wedge \Phi'$$

for some literals u and v (each of which might be y, $\bar{y}$, z, or $\bar{z}$). It follows that the first recursive formula $\Phi|x$ has contains the clause $(u \vee v)$. We can recursively eliminate the variables u and v just as we tested the variables y and x in the second and third cases of our previous algorithm:

$$\Phi|x = (u \vee v) \wedge \Phi'|x = (u \wedge \Phi'|xu) \vee (v \wedge \Phi'|x\bar{u}v).$$

Here is our new faster algorithm:

```
3SAT(Φ):
  if Φ = ∅
      return True
  if Φ has a pure literal x
      return 3SAT(Φ|x)
  (x ∨ y ∨ z) ∧ (x̄ ∨ u ∨ v) ∧ Φ' ← Φ
  if 3SAT(Φ|xu)
      return True
  if 3SAT(Φ|xūv)
      return True
  if 3SAT(Φ|x̄y)
      return True
  return 3SAT(Φ|x̄ȳz)
```

The running time $T(n)$ of this new algorithm satisfies the recurrence

$$T(n) = 2T(n-2) + 2T(n-3) + \operatorname{poly}(n),$$

and the annihilator method implies that

$$T(n) = O(\mu^n \operatorname{poly}(n)) = \boxed{O(1.76929235425^n)}$$

where $\mu \approx 1.76929235424\ldots$ is the largest root of the characteristic polynomial $r^3 - 2r - 2$.

Naturally, this approach can be extended much further; since 1998, at least fifteen different 3SAT algorithms have been published, each improving the running time by a small amount. As of 2010, the fastest deterministic algorithm for 3SAT runs in $O(1.33334^n)$ time[2], and the fastest

[2] Robin A. Moser and Dominik Scheder. A full derandomization of Schöning's k-SAT algorithm. ArXiv:1008.4067, 2010.

randomized algorithm runs in $O(1.32113^n)$ expected time[3], but there is good reason to believe that these are *not* the best possible.

4.2 Maximum Independent Set

Now suppose we are given an undirected graph G and are asked to find the size of the *largest independent set*, that is, the largest subset of the vertices of G with no edges between them. Once again, we have an obvious recursive algorithm: Try every subset of nodes, and return the largest subset with no edges. Expressed recursively, the algorithm might look like this.

```
MaximumIndSetSize(G):
  if G = ∅
      return 0
  else
      v ← any node in G
      withv ← 1 + MaximumIndSetSize(G \ N(v))
      withoutv ← MaximumIndSetSize(G \ {v})
      return max{withv, withoutv}.
```

Here, $N(v)$ denotes the *neighborhood* of v: The set containing v and all of its neighbors. Our algorithm is exploiting the fact that if an independent set contains v, then by definition it contains none of v's neighbors. In the worst case, v has no neighbors, so $G \setminus \{v\} = G \setminus N(v)$. Thus, the running time of this algorithm satisfies the recurrence $T(n) = 2T(n-1) + \text{poly}(n) = O(2^n \text{poly}(n))$. Surprise, surprise.

This algorithm is mirroring a crude recursive upper bound for the number of *maximal* independent sets in a graph; an independent set is maximal if every vertex in G is either already in the set or a neighbor of a vertex in the set. If the graph is non-empty, then every maximal independent set either includes or excludes each vertex. Thus, the number of maximal independent sets satisfies the recurrence $M(n) \leq 2M(n-1)$, with base case $M(1) = 1$. The annihilator method gives us $M(n) \leq 2^n - 1$. The only subset that we aren't counting with this upper bound is the empty set!

We can speed up our algorithm by making several careful modifications to avoid the worst case of the running-time recurrence.

- If v has no neighbors, then $N(v) = \{v\}$, and both recursive calls consider a graph with $n-1$ nodes. But in this case, v is in *every* maximal independent set, so one of the recursive calls is redundant. On the other hand, if v has at least one neighbor, then $G \setminus N(v)$ has at most $n-2$ nodes. So now we have the following recurrence.

$$T(n) \leq O(\text{poly}(n)) + \max \left\{ \begin{array}{l} T(n-1) \\ T(n-1) + T(n-2) \end{array} \right\} = O(1.61803398875^n)$$

  The upper bound is derived by solving each case separately using the annihilator method and taking the larger of the two solutions. The first case gives us $T(n) = O(\text{poly}(n))$; the second case yields our old friends the Fibonacci numbers.

- We can improve this bound even more by examining the new worst case: v has exactly one neighbor w. In this case, either v or w appears in every maximal independent set.

[3] Kazuo Iwama, Kazuhisa Seto, Tadashi Takai, and Suguru Tamaki. Improved randomized algorithms for 3-SAT. To appear in *Proc. STACS*, 2010.

However, given any independent set that includes w, removing w and adding v creates another independent set of the same size. It follows that *some maximum independent set includes* v, so we don't need to search the graph $G \setminus \{v\}$, and the $G \setminus N(v)$ has at most $n-2$ nodes. On the other hand, if the degree of v is at least 2, then $G \setminus N(v)$ has at most $n-3$ nodes.

$$T(n) \le O(\text{poly}(n)) + \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-3) \end{array} \right\} = O(1.46557123188^n)$$

The base of the exponent is the largest root of the characteristic polynomial $r^3 - r^2 - 1$.

- Now the worst-case is a graph where every node has degree at least 2; we split this worst case into two subcases. If G has a node v with degree 3 or more, then $G \setminus N(v)$ has at most $n-4$ nodes. Otherwise (since we have already considered nodes of degree 0 and 1), *every* node in G has degree 2. Let u, v, w be a path of three nodes in G (possibly with u adjacent to w). In any maximal independent set, either v is present and u, w are absent, or u is present and its two neighbors are absent, or w is present and its two neighbors are absent. In all three cases, we recursively count maximal independent sets in a graph with $n-3$ nodes.

$$T(n) \le O(\text{poly}(n)) + \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-4) \\ 3T(n-3) \end{array} \right\} = O(3^{n/3}\,\text{poly}(n)) = O(1.44224957031^n)$$

The base of the exponent is $\sqrt[3]{3}$, the largest root of the characteristic polynomial $r^3 - 3$. The third case would give us a bound of $O(1.3802775691^n)$, where the base is the largest root of the characteristic polynomial $r^4 - r^3 - 1$.

- Now the worst case for our algorithm is a graph with an extraordinarily special structure: *Every node has degree 2*. In other words, every component of G is a cycle. But it is easy to prove that the largest independent set in a cycle of length k has size $\lfloor k/2 \rfloor$. So we can handle this case directly in polynomial time, without no recursion at all!

$$T(n) \le O(\text{poly}(n)) + \max \left\{ \begin{array}{l} T(n-1) \\ T(n-2) \\ T(n-1) + T(n-4) \end{array} \right\} = O(1.3802775691^n)$$

Again, the base of the exponential running time is the largest root of the characteristic polynomial $r^4 - r^3 - 1$.

```
MaximumIndSetSize(G):
  if G = ∅
      return 0

  else if G has a node v with degree 0 or 1
      return 1 + MaximumIndSetSize(G \ N(v))        ⟨⟨≤ n − 1⟩⟩

  else if G has a node v with degree greater than 2
      withv ← 1 + MaximumIndSetSize(G \ N(v))       ⟨⟨≤ n − 4⟩⟩
      withoutv ← MaximumIndSetSize(G \ {v})         ⟨⟨≤ n − 1⟩⟩
      return max{withv, withoutv}

  else ⟨⟨every node in G has degree 2⟩⟩
      total ← 0
      for each component of G
          k ← number of vertices in the component
          total ← total + ⌊k/2⌋
      return total
```

As with 3SAT, further improvements are possible but increasingly complex. As of 2010, the fastest published algorithm for computing maximum independent sets runs in $O(1.2210^n)$ time[4]. However, in an unpublished technical report, Robson describes a *computer-generated* algorithm that runs in $O(2^{n/4}\operatorname{poly}(n)) = O(1.1889^n)$ time; just the description of this algorithm requires more than 15 pages.[5]

Exercises

1. (a) Prove that any n-vertex graph has at most $3^{n/3}$ maximal independent sets. *[Hint: Modify the* MaximumIndSetSize *algorithm so that it lists all maximal independent sets.]*

   (b) Describe an n-vertex graph with exactly $3^{n/3}$ maximal independent sets, for every integer n that is a multiple of 3.

*2. Describe an algorithm to solve 3SAT in time $O(\phi^n \operatorname{poly}(n))$, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$. *[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]*

[4] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Measure and conquer: A simple $O(2^{0.288n})$ independent set algorithm. *Proc. SODA*, 18–25, 2006.

[5] Mike Robson. Finding a maximum independent set in time $O(2^{n/4})$. Technical report 1251-01, LaBRI, 2001. ⟨http://www.labri.fr/perso/robson/mis/techrep.ps⟩.

Those who cannot remember the past are doomed to repeat it.

— George Santayana, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word 'research'. *I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term* 'research' *in his presence. You can imagine how he felt, then, about the term* 'mathematical'. *The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?*

— Richard Bellman, on the origin of his term 'dynamic programming' (1984)

If we all listened to the professor, we may be all looking for professor jobs.

— Pittsburgh Steelers' head coach Bill Cowher, responding to David Romer's dynamic-programming analysis of football strategy (2003)

5 Dynamic Programming

5.1 Fibonacci Numbers

5.1.1 Recursive Definitions Are Recursive Algorithms

The Fibonacci numbers F_n, named after Leonardo Fibonacci Pisano[1], the mathematician who popularized 'algorism' in Europe in the 13th century, are defined as follows: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them:

```
RecFibo(n):
  if (n < 2)
      return n
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

How long does this algorithm take? Except for the recursive calls, the entire algorithm requires only a constant number of steps: one comparison and possibly one addition. If $T(n)$ represents the number of recursive calls to RecFibo, we have the recurrence

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1.$$

This looks an awful lot like the recurrence for Fibonacci numbers! The annihilator method gives us an asymptotic bound of $\Theta(\phi^n)$, where $\phi = (\sqrt{5}+1)/2 \approx 1.61803398875$, the so-called *golden ratio*, is the largest root of the polynomial $r^2 - r - 1$. But it's fairly easy to prove (hint, hint) the exact solution $\boldsymbol{T(n) = 2F_{n+1} - 1}$. In other words, computing F_n using this algorithm takes more than twice as many steps as just counting to F_n!

Another way to see this is that the RecFibo is building a big binary tree of additions, with nothing but zeros and ones at the leaves. Since the eventual output is F_n, our algorithm must

[1] literally, "Leonardo, son of Bonacci, of Pisa"

call RecRibo(1) (which returns 1) exactly F_n times. A quick inductive argument implies that RecFibo(0) is called exactly F_{n-1} times. Thus, the recursion tree has $F_n + F_{n-1} = F_{n+1}$ leaves, and therefore, because it's a full binary tree, it must have $2F_{n+1} - 1$ nodes.

5.1.2 Memo(r)ization: Remember Everything

The obvious reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers over and over and over. A single call to RecFibo(n) results in one recursive call to RecFibo($n-1$), two recursive calls to RecFibo($n-2$), three recursive calls to RecFibo($n-3$), five recursive calls to RecFibo($n-4$), and in general F_{k-1} recursive calls to RecFibo($n-k$) for any integer $0 \le k < n$. Each call is recomputing some Fibonacci number from scratch.

We can speed up our recursive algorithm considerably just by writing down the results of our recursive calls and looking them up again if we need them later. This process was dubbed *memoization* by Richard Michie in the late 1960s.[2]

```
MemFibo(n):
  if (n < 2)
        return n
  else
        if F[n] is undefined
              F[n] ← MemFibo(n − 1) + MemFibo(n − 2)
        return F[n]
```

Memoization clearly decreases the running time of the algorithm, but by how much? If we actually trace through the recursive calls made by MemFibo, we find that the array $F[\,]$ is filled from the bottom up: first $F[2]$, then $F[3]$, and so on, up to $F[n]$. This pattern can be verified by induction: Each entry $F[i]$ is filled only after its predecessor $F[i-1]$. If we ignore the time spent in recursive calls, it requires only constant time to evaluate the recurrence for each Fibonacci number F_i. But by design, the recurrence for F_i is evaluated only once for each index i! We conclude that MemFibo performs only $O(n)$ additions, an *exponential* improvement over the naïve recursive algorithm!

5.1.3 Dynamic Programming: Fill Deliberately

But once we see how the array $F[\,]$ is filled, we can replace the recursion with a simple loop that intentionally fills the array in order, instead of relying on the complicated recursion to do it for us 'accidentally'.

```
IterFibo(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
        F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

Now the time analysis is immediate: IterFibo clearly uses ***$O(n)$ additions*** and stores $O(n)$ *integers*.

This gives us our first explicit *dynamic programming* algorithm. The dynamic programming paradigm was developed by Richard Bellman in the mid-1950s, while working at the RAND

[2]"My name is Elmer J. Fudd, millionaire. I own a mansion and a yacht."

Corporation. Bellman deliberately chose the name 'dynamic programming' to hide the mathematical character of his work from his military bosses, who were actively hostile toward anything resembling mathematical research. Here, the word 'programming' does not refer to writing code, but rather to the older sense of *planning* or *scheduling*, typically by filling in a table. For example, sports programs and theater programs are schedules of important events (with ads); television programming involves filling each available time slot with a show (and ads); degree programs are schedules of classes to be taken (with ads). The Air Force funded Bellman and others to develop methods for constructing training and logistics schedules, or as they called them, 'programs'. The word 'dynamic' is meant to suggest that the table is filled in over time, rather than all at once (as in 'linear programming', which we will see later in the semester).[3]

5.1.4 Don't Remember Everything After All

In many dynamic programming algorithms, it is not necessary to retain *all* intermediate results through the entire computation. For example, we can significantly reduce the space requirements of our algorithm IterFibo by maintaining only the two newest elements of the array:

```
ITERFIBO2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

(This algorithm uses the non-standard but perfectly consistent base case $F_{-1} = 1$ so that IterFibo2(0) returns the correct value 0.)

5.1.5 Faster! Faster!

Even this algorithm can be improved further, using the following wonderful fact:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x+y \end{bmatrix}$$

In other words, multiplying a two-dimensional vector by the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$ does exactly the same thing as one iteration of the inner loop of IterFibo2. This might lead us to believe that multiplying by the matrix n times is the same as iterating the loop n times:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$

A quick inductive argument proves this fact. So if we want the nth Fibonacci number, we just have to compute the nth power of the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$. If we use repeated squaring, computing the nth power of something requires only $O(\log n)$ multiplications. In this case, that means $O(\log n)$ 2×2 matrix multiplications, each of which reduces to a constant number of integer multiplications and additions. Thus, we can compute F_n in only ***$O(\log n)$ integer arithmetic operations***.

This is an exponential speedup over the standard iterative algorithm, which was already an exponential speedup over our original recursive algorithm. Right?

[3] "I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

5.1.6 Whoa! Not so fast!

Well, not exactly. Fibonacci numbers grow exponentially fast. The nth Fibonacci number is approximately $n \log_{10} \phi \approx n/5$ decimal digits long, or $n \log_2 \phi \approx 2n/3$ bits. So we can't possibly compute F_n in logarithmic time — we need $\Omega(n)$ time just to write down the answer!

The way out of this apparent paradox is to observe that ***we can't perform arbitrary-precision arithmetic in constant time***. Let $M(n)$ denote the time required to multiply two n-digit numbers. The matrix-based algorithm's actual running time obeys the recurrence $T(n) = T(\lfloor n/2 \rfloor) + M(n)$, which solves to $T(n) = \boldsymbol{O(M(n))}$ using recursion trees. The fastest known multiplication algorithm runs in time $O(n \log n 2^{O(\log^* n)})$, so that is also the running time of the fastest algorithm known to compute Fibonacci numbers.

Is this algorithm slower than our initial "linear-time" iterative algorithm? No! Addition isn't free, either. Adding two n-digit numbers takes $O(n)$ time, so the running time of the iterative algorithm is $\boldsymbol{O(n^2)}$. (Do you see why?) The matrix-squaring algorithm really is faster than the iterative addition algorithm, but not exponentially faster.

In the original recursive algorithm, the extra cost of arbitrary-precision arithmetic is overwhelmed by the huge number of recursive calls. The correct recurrence is $T(n) = T(n-1) + T(n-2) + O(n)$, for which the annihilator method still implies the solution $T(n) = O(\phi^n)$.

5.2 Longest Increasing Subsequence

In a previous lecture, we developed a recursive algorithm to find the length of the longest increasing subsequence of a given sequence of numbers. Given an array $A[1..n]$, the length of the longest increasing subsequence is computed by the function call LISBIGGER($-\infty$, $A[1..n]$), where LISBIGGER is the following recursive algorithm:

```
LISBIGGER(prev, A[1..n]):
  if n = 0
      return 0
  else
      max ← LISBIGGER(prev, A[2..n])
      if A[1] > prev
          L ← 1 + LISBIGGER(A[1], A[2..n])
          if L > max
              max ← L
      return max
```

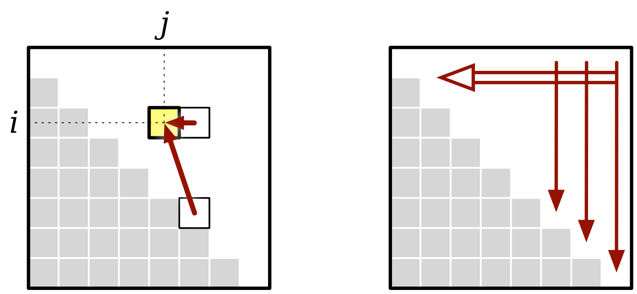
We can simplify our notation slightly with two simple observations. First, the input variable *prev* is always either $-\infty$ or an element of the input array. Second, the second argument of LISBIGGER is always a *suffix* of the original input array. If we add a new sentinel value $A[0] = -\infty$ to the input array, we can identify any recursive subproblem with two array indices.

Thus, we can rewrite the recursive algorithm as follows. Add the sentinel value $A[0] = -\infty$. Let $LIS(i, j)$ denote the length of the longest increasing subsequence of $A[j..n]$ with all elements larger than $A[i]$. Our goal is to compute $LIS(0, 1)$. For all $i < j$, we have

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i, j+1),\ 1 + LIS(j, j+1)\} & \text{otherwise} \end{cases}$$

Because each recursive subproblem can be identified by two indices i and j, we can store the intermediate values in a two-dimensional array $\boldsymbol{LIS[0\,..\,n, 1\,..\,n]}$.[4] Since there are $O(n^2)$ entries in the table, our memoized algorithm uses $\boldsymbol{O(n^2)}$ ***space***. Each entry in the table can be computed in $O(1)$ time once we know its predecessors, so our memoized algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

It's not immediately clear what order the recursive algorithm fills the rest of the table; all we can tell from the recurrence is that each entry $LIS[i, j]$ is filled in *after* the entries $LIS[i, j+1]$ and $LIS[j, j+1]$ in the next columns. But just this partial information is enough to give us an explicit evaluation order. If we fill in our table one column at a time, from right to left, then whenever we reach an entry in the table, the entries it depends on are already available.



Dependencies in the memoization table for longest increasing subsequence, and a legal evaluation order

Finally, putting everything together, we obtain the following dynamic programming algorithm:

```
LIS(A[1..n]):
  A[0] ← −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                     ⟨⟨Base cases⟩⟩
      LIS[i, n+1] ← 0

  for j ← n downto 1
      for i ← 0 to j − 1
          if A[i] ≥ A[j]
              LIS[i, j] ← LIS[i, j+1]
          else
              LIS[i, j] ← max{LIS[i, j+1], 1 + LIS[j, j+1]}

  return LIS[0, 1]
```

As expected, the algorithm clearly uses $\boldsymbol{O(n^2)}$ ***time and space***. However, we can reduce the space to $O(n)$ by only maintaining the two most recent columns of the table, $LIS[\cdot, j]$ and $LIS[\cdot, j+1]$.[5]

This is not the only recursive strategy we could use for computing longest increasing subsequences efficiently. Here is another recurrence that gives us the $O(n)$ space bound for free. Let $LIS'(i)$ denote the length of the longest increasing subsequence of $A[i\,..\,n]$ that starts with $A[i]$. Our goal is to compute $LIS'(0) - 1$; we subtract 1 to ignore the sentinel value $-\infty$. To define $LIS'(i)$ recursively, we only need to specify the *second* element in subsequence; the Recursion Fairy will do the rest.

$$LIS'(i) = 1 + \max\left\{LIS'(j) \mid j > i \text{ and } A[j] > A[i]\right\}$$

Here, I'm assuming that $\max \varnothing = 0$, so that the base case is $L'(n) = 1$ falls out of the recurrence automatically. Memoizing this recurrence requires only $\boldsymbol{O(n)}$ ***space***, and the resulting algorithm

[4]In fact, we only need half of this array, because we always have $i < j$. But even if we cared about constant factors in this class (we don't), this would be the wrong time to worry about them. The first order of business is to find an algorithm that actually *works*; once we have that, then we can think about optimizing it.

[5]See, I told you not to worry about constant factors yet!

runs in ***$O(n^2)$ time***. To transform this memoized recurrence into a dynamic programming algorithm, we only need to guarantee that $LIS'(j)$ is computed before $LIS'(i)$ whenever $i < j$.

```
LIS2(A[1..n]):
  A[0] = −∞                    ⟨⟨Add a sentinel⟩⟩
  for i ← n downto 0
      LIS'[i] ← 1
      for j ← i+1 to n
          if A[j] > A[i] and 1 + LIS'[j] > LIS'[i]
              LIS'[i] ← 1 + LIS'[j]
  return LIS'[0] − 1           ⟨⟨Don't count the sentinel⟩⟩
```

5.3 The Pattern: Smart Recursion

In a nutshell, dynamic programming is *recursion without repetition*. Dynamic programming algorithms store the solutions of intermediate subproblems, often *but not always* in some kind of array or table. Many algorithms students make the mistake of focusing on the table (because tables are easy and familiar) instead of the *much* more important (and difficult) task of finding a correct recurrence. As long as we memoize the correct recurrence, an explicit table isn't really necessary, but if the recursion is incorrect, nothing works.

Dynamic programming is *not* about filling in tables.
It's about smart recursion!

Dynamic programming algorithms are almost always developed in two distinct stages.

1. **Formulate the problem recursively.** Write down a recursive formula or algorithm for the whole problem in terms of the answers to smaller subproblems. This is the hard part. It generally helps to think in terms of a recursive definition of the object you're trying to construct. A complete recursive formulation has two parts:

   (a) Describe the precise function you want to evaluate, in coherent English. Without this specification, it is impossible, even in principle, to determine whether your solution is correct.

   (b) Give a formal recursive definition of that function.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

   (a) **Identify the subproblems.** What are all the different ways can your recursive algorithm call itself, starting with some initial input? For example, the argument to RecFibo is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2}, we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

(c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

(d) **Identify dependencies between subproblems.** Except for the base cases, every recursive subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

(e) **Find a good evaluation order.** Order the subproblems so that each subproblem comes *after* the subproblems it depends on. Typically, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. ***Be careful!***

(f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence. ***You don't need to do this on homework or exams.***

Of course, you have to prove that each of these steps is correct. If your recurrence is wrong, or if you try to build up answers in the wrong order, your algorithm won't work!

5.4 Warning: Greed is Stupid

If we're very very very very lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is look for the best first step, take it, and then continue. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are *very* rare. Nevertheless, for many problems that should be solved by dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions don't cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string. If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution.

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

What, never?
No, never!
What, *never*?

Well. . . hardly ever.[6]

A different lecture note describes the effort required to prove that greedy algorithms are correct, in the rare instances when they are. **You will not receive *any* credit for *any* greedy algorithm for *any* problem in this class without a *formal* proof of correctness.** We'll push through the formal proofs for several greedy algorithms later in the semester.

5.5 Edit Distance

The *edit distance* between two words—sometimes also called the *Levenshtein distance*—is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word into another. For example, the edit distance between `FOOD` and `MONEY` is at most four:

$$\texttt{\underline{F}OOD} \longrightarrow \texttt{MO\underline{O}D} \longrightarrow \texttt{MON}_{\wedge}\texttt{D} \longrightarrow \texttt{MONE\underline{D}} \longrightarrow \texttt{MONEY}$$

A better way to display this editing process is to place the words one above the other, with a gap in the first word for every insertion, and a gap in the second word for every deletion. Columns with two *different* characters correspond to substitutions. Thus, the number of editing steps is just the number of columns that don't contain the same character twice.

```
F O O   D
M O N E Y
```

It's fairly obvious that you can't get from `FOOD` to `MONEY` in three steps, so their edit distance is exactly four. Unfortunately, this is not so easy in general. Here's a longer example, showing that the distance between `ALGORITHM` and `ALTRUISTIC` is at most six. Is this optimal?

```
A L G O R   I   T H M
A L   T R U I S T I C
```

To develop a dynamic programming algorithm to compute the edit distance between two strings, we first need to develop a recursive definition. Our gap representation for edit sequences has a crucial "optimal substructure" property. Suppose we have the gap representation for the shortest edit sequence for two strings. **If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining substrings.** We can easily prove this by contradiction. If the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings. Once we figure out what should go in the last column, the Recursion Fairy will magically give us the rest of the optimal gap representation.

So let's recursively define the edit distance between two strings $A[1\,..\,m]$ and $B[1\,..\,n]$, which we denote by $\mathit{Edit}(A[1\,..\,m], B[1\,..\,n])$. If neither string is empty, there are three possibilities for the last column in the shortest edit sequence:

- **Insertion:** The last entry in the bottom row is empty. In this case, the edit distance is equal to $\mathit{Edit}(A[1\,..\,m-1], B[1\,..\,n]) + 1$. The $+1$ is the cost of the final insertion, and the recursive expression gives the minimum cost for the other columns.

[6]Greedy methods hardly ever work!
So give three cheers, and one cheer more,
for the careful Captain of the *Pinafore*!
Then give three cheers, and one cheer more,
for the Captain of the *Pinafore*!

- **Deletion:** The last entry in the top row is empty. In this case, the edit distance is equal to $Edit(A[1..m], B[1..n-1]) + 1$. The +1 is the cost of the final deletion, and the recursive expression gives the minimum cost for the other columns.

- **Substitution:** Both rows have characters in the last column. If the characters are the same, the substitution is free, so the edit distance is equal to $Edit(A[1..m-1], B[1..n-1])$. If the characters are different, then the edit distance is equal to $Edit(A[1..m-1], B[1..n-1])+1$.

The edit distance between A and B is the smallest of these three possibilities:[7]

$$\boxed{Edit(A[1..m], B[1..n]) = \min\left\{\begin{array}{l} Edit(A[1..m-1], B[1..n]) + 1 \\ Edit(A[1..m], B[1..n-1]) + 1 \\ Edit(A[1..m-1], B[1..n-1]) + \big[A[m] \neq B[n]\big] \end{array}\right\}}$$

This recurrence has two easy base cases. The only way to convert the empty string into a string of n characters is by performing n insertions. Similarly, the only way to convert a string of m characters into the empty string is with m deletions, Thus, if ε denotes the empty string, we have

$$Edit(A[1..m], \varepsilon) = m, \qquad Edit(\varepsilon, B[1..n]) = n.$$

Both of these expressions imply the trivial base case $Edit(\varepsilon, \varepsilon) = 0$.

Now notice that the arguments to our recursive subproblems are always *prefixes* of the original strings A and B. We can simplify our notation by using the lengths of the prefixes, instead of the prefixes themselves, as the arguments to our recursive function.

$$\boxed{\text{Let } Edit(i, j) \text{ denote the edit distance between the prefixes } A[1..i] \text{ and } B[1..j].}$$

This function satisfies the following recurrence:

$$\boxed{Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min\left\{\begin{array}{l} Edit(i-1, j) + 1, \\ Edit(i, j-1) + 1, \\ Edit(i-1, j-1) + \big[A[i] \neq B[j]\big] \end{array}\right\} & \text{otherwise} \end{cases}}$$

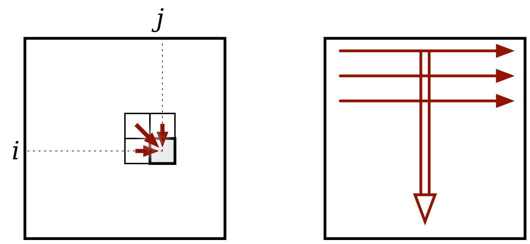
The edit distance between the original strings A and B is just $Edit(m, n)$. This recurrence translates directly into a recursive algorithm; the precise running time is not obvious, but it's clearly exponential in m and n. ***Fortunately, we don't care about the precise running time of the recursive algorithm.*** The recursive running time wouldn't tell us anything about our eventual dynamic programming algorithm, so we're just not going to bother computing it.[8]

Because each recursive subproblem can be identified by two indices i and j, we can memoize intermediate values in a two-dimensional array $\boldsymbol{Edit[0..m, 0..n]}$. Note that the index ranges start at zero to accommodate the base cases. Since there are $\Theta(mn)$ entries in the table, our memoized algorithm uses $\boldsymbol{\Theta(mn)}$ ***space***. Since each entry in the table can be computed in $\Theta(1)$ time once we know its predecessors, our memoized algorithm runs in $\boldsymbol{\Theta(mn)}$ ***time***.

[7]Once again, I'm using Iverson's bracket notation $[P]$ to denote the *indicator variable* for the logical proposition P, which has value 1 if P is true and 0 if P is false.

[8]In case you're curious, the running time of the unmemoized recursive algorithm obeys the following recurrence:

$$T(m,n) = \begin{cases} O(1) & \text{if } n = 0 \text{ or } m = 0, \\ T(m, n-1) + T(m-1, n) + T(n-1, m-1) + O(1) & \text{otherwise.} \end{cases}$$



Dependencies in the memoization table for edit distance, and a legal evaluation order

Each entry $Edit[i, j]$ depends only on its three neighboring entries $Edit[i-1, j]$, $Edit[i, j-1]$, and $Edit[i-1, j-1]$. If we fill in our table in the standard row-major order—row by row from top down, each row from left to right—then whenever we reach an entry in the table, the entries it depends on are already available. Putting everything together, we obtain the following dynamic programming algorithm:

```
EditDistance(A[1..m], B[1..n]):
  for j ← 1 to n
      Edit[0, j] ← j

  for i ← 1 to m
      Edit[i, 0] ← i
      for j ← 1 to n
          if A[i] = B[j]
              Edit[i, j] ← min {Edit[i − 1, j] + 1, Edit[i, j − 1] + 1, Edit[i − 1, j − 1]}
          else
              Edit[i, j] ← min {Edit[i − 1, j] + 1, Edit[i, j − 1] + 1, Edit[i − 1, j − 1] + 1}

  return Edit[m, n]
```

The resulting table for ALGORITHM $\to$ ALTRUISTIC is shown on the next page. Bold numbers indicate places where characters in the two strings are equal. The arrows represent the predecessor(s) that actually define each entry. Each direction of arrow corresponds to a different edit operation: horizontal=deletion, vertical=insertion, and diagonal=substitution. Bold diagonal arrows indicate "free" substitutions of a letter for itself. Any path of arrows from the top left corner to the bottom right corner of this table represents an optimal edit sequence between the two strings. (There can be many such paths.) Moreover, since we can compute these arrows in a post-processing phase from the values stored in the table, we can reconstruct the actual optimal editing sequence in $O(n+m)$ additional time.

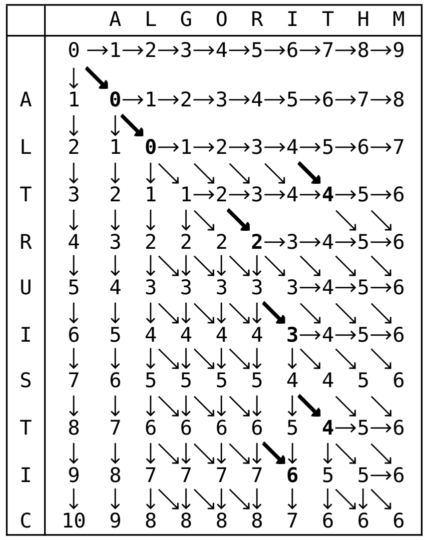
The edit distance between ALGORITHM and ALTRUISTIC is indeed six. There are three paths through this table from the top left to the bottom right, so there are three optimal edit sequences:

```
A L G O R I   T H M
A L T R U I S T I C

A L G O R   I   T H M
A L   T R U I S T I C
```

I don't know of a general closed-form solution for this mess, but we can derive an upper bound by defining a new function

$$T'(N) = \max_{n+m=N} T(n,m) = \begin{cases} O(1) & \text{if } N = 0, \\ 2T(N-1) + T(N-2) + O(1) & \text{otherwise.} \end{cases}$$

The annihilator method implies that $T'(N) = O((1+\sqrt{2})^N)$. Thus, the running time of our recursive edit-distance algorithm is *at most* $T'(n+m) = O((1+\sqrt{2})^{n+m})$.



The memoization table for *Edit*(ALGORITHM, ALTRUISTIC)

A	L	G	O	R		I		T	H	M
A	L	T		R	U	I	S	T	I	C

5.6 More Examples

In the previous note on backtracking algorithms, we saw two other examples of recursive algorithms that we can significantly speed up via dynamic programming.

5.6.1 Subset Sum

Recall that the *Subset Sum* problem asks, given a set X of positive integers (represented as an array $X[1..n]$ and an integer T, whether any subset of X sums to T. In that lecture, we developed a recursive algorithm which can be reformulated as follows. Fix the original input array $X[1..n]$ and the original target sum T, and define the boolean function

$$SS(i,t) = \text{some subset of } X[i..n] \text{ sums to } t.$$

Our goal is to compute $S(1,T)$, using the recurrence

$$SS(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0, \\ \textsc{False} & \text{if } t < 0 \text{ or } i > n, \\ SS(i+1,t) \vee SS(i+1,t-X[i]) & \text{otherwise.} \end{cases}$$

There are only nT possible values for the input parameters that lead to the interesting case of this recurrence, and we can memoize all such values in an $n \times T$ array. If $S(i+1,t)$ and $S(i+1,t-X[i])$ are already known, we can compute $S(i,t)$ in constant time, so memoizing this recurrence gives us and algorithm that runs in $\boldsymbol{O(nT)}$ ***time***.[9] To turn this into an explicit dynamic programming algorithm, we only need to consider the subproblems $S(i,t)$ in the proper order:

[9]Even though *SubsetSum* is NP-complete, this bound does *not* imply that P=NP, because T is not necessary bounded by a polynomial function of the input size.

```
SUBSETSUM(X[1..n], T):
  S[n+1, 0] ← TRUE
  for t ← 1 to T
      S[n+1, t] ← FALSE
  for i ← n downto 1
      S[i, 0] = TRUE
      for t ← 1 to X[i] − 1
          S[i, t] ← S[i+1, t]          ⟨⟨Avoid the case t < 0⟩⟩
      for t ← X[i] to T
          S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

This iterative algorithm clearly always uses ***$O(nT)$ time and space***. In particular, if T is significantly larger than 2^n, this algorithm is actually slower than our naïve recursive algorithm. Dynamic programming isn't *always* an improvement!

5.6.2 NFA acceptance

The other problem we considered in the previous lecture note was determining whether a given NFA $M = (\Sigma, Q, s, A, \delta)$ accepts a given string $w \in \Sigma^*$. To make the problem concrete, we can assume without loss of generality that the alphabet is $\Sigma = \{1, 2, \ldots, |\Sigma|\}$, the state set is $Q = \{1, 2, \ldots, |Q|\}$, the start state is state 1, and our input consists of three arrays:

- A boolean array $A[1 .. |Q|]$, where $A[q] = \text{TRUE}$ if and only if $q \in A$.
- A boolean array $\delta[1 .. |Q|, 1 .. |\Sigma|, 1 .. |Q|]$, where $\delta[p, a, q] = \text{TRUE}$ if and only if $p \in \delta(q, a)$.
- An array $w[1 .. n]$ of symbols, representing the input string.

Now consider the boolean function

$$\textit{Accepts?}(q, i) = \text{TRUE if and only if } M \text{ accepts the suffix } w[i .. n] \text{ starting in state } q,$$

or equivalently,

$$\textit{Accepts?}(q, i) = \text{TRUE if and only if } \delta^*(q, w[i .. n]) \text{ contains at least one state in } A.$$

We need to compute *Accepts*$(1, 1)$. The recursive definition of the string transition function δ^* implies the following recurrence for *Accepts?*:

$$\textit{Accepts?}(q, i) := \begin{cases} \text{TRUE} & \text{if } i > n \text{ and } q \in A \\ \text{FALSE} & \text{if } i > n \text{ and } q \in A \\ \displaystyle\bigvee_{r \in \delta(q,a)} \textit{Accepts?}(r, x) & \text{if } w = ax \end{cases}$$

Rewriting this recurrence in terms of our input representation gives us the following:

$$\textit{Accepts?}(q, i) := \begin{cases} \text{TRUE} & \text{if } i > n \text{ and } A[q] = \text{TRUE} \\ \text{FALSE} & \text{if } i > n \text{ and } A[q] = \text{FALSE} \\ \displaystyle\bigvee_{r=1}^{|Q|} \big(\delta[q, w[i], r] \wedge \textit{Accepts?}(r, i+1)\big) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $Accepts?[1..|Q|, 1..n+1]$. Each entry $Accepts?[q,i]$ depends on some subset of entries of the form $Accepts?[r,i+1]$. So we can fill the memoization table by considering the possible indices i in decreasing order in the outer loop, and consider states q in arbitrary order in the inner loop. Evaluating each entry $Accepts?[q,i]$ requires $O(|Q|)$ time, using an even deeper loop over all states r, and there are $O(n|Q|)$ such entries. Thus, the entire dynamic programming algorithm requires $\boldsymbol{O(n|Q|^2)}$ ***time***.

```
NFAACCEPTS?(A[1..|Q|], δ[1..|Q|,1..|Σ|,1..|Q|], w[1..n]):
  for q ← 1 to |Q|
      Accepts?[q,n+1] ← A[q]
  for i ← n down to 1
      for q ← 1 to |Q|
          Accepts?[q,i] ← FALSE
          for r ← 1 to |Q|
              if δ[q,w[i],r] and Accepts?[r,i+1]
                  Accepts?[q,i] ← TRUE
  return Accepts?[1,1]
```

5.7 Optimal Binary Search Trees

In an earlier lecture, we developed a recursive algorithm for the optimal binary search tree problem. We are given a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches to $A[i]$. Our task is to construct a binary search tree for that set such that the total cost of all the searches is as small as possible. We developed the following recurrence for this problem:

$$OptCost(f[1..n]) = \min_{1\le r\le n}\left\{OptCost(f[1..r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; OptCost(f[r+1..n])\right\}$$

To put this recurrence in more standard form, fix the frequency array f, and let $OptCost(i,j)$ denote the total search time in the optimal search tree for the subarray $A[i..j]$. To simplify notation a bit, let $F(i,j)$ denote the total frequency count for all the keys in the interval $A[i..j]$:

$$F(i,j) := \sum_{k=i}^{j} f[k]$$

We can now write

$$\boxed{OptCost(i,j) = \begin{cases} 0 & \text{if } j < i \\ F(i,j) + \min\limits_{i\le r\le j}\Big(OptCost(i,r-1) + OptCost(r+1,j)\Big) & \text{otherwise} \end{cases}}$$

The base case might look a little weird, but all it means is that the total cost for searching an empty set of keys is zero.

The algorithm will be somewhat simpler and more efficient if we precompute all possible values of $F(i,j)$ and store them in an array. Computing each value $F(i,j)$ using a separate for-loop would $O(n^3)$ time. A better approach is to turn the recurrence

$$F(i,j) = \begin{cases} f[i] & \text{if } i = j \\ F(i,j-1) + f[j] & \text{otherwise} \end{cases}$$

into the following $O(n^2)$-time dynamic programming algorithm:

```
InitF(f[1..n]):
  for i ← 1 to n
      F[i, i−1] ← 0
      for j ← i to n
          F[i, j] ← F[i, j−1] + f[j]
```

This will be used as an initialization subroutine in our final algorithm.

So now let's compute the optimal search tree cost $OptCost(1, n)$ from the bottom up. We can store all intermediate results in a table $OptCost[1..n, 0..n]$. Only the entries $OptCost[i, j]$ with $j \geq i-1$ will actually be used. The base case of the recurrence tells us that any entry of the form $OptCost[i, i-1]$ can immediately be set to 0. For any other entry $OptCost[i, j]$, we can use the following algorithm fragment, which comes directly from the recurrence:

```
ComputeOptCost(i, j):
  OptCost[i, j] ← ∞
  for r ← i to j
      tmp ← OptCost[i, r−1] + OptCost[r+1, j]
      if OptCost[i, j] > tmp
          OptCost[i, j] ← tmp
  OptCost[i, j] ← OptCost[i, j] + F[i, j]
```

The only question left is what order to fill in the table.

Each entry $OptCost[i, j]$ depends on all entries $OptCost[i, r-1]$ and $OptCost[r+1, j]$ with $i \leq k \leq j$. In other words, every entry in the table depends on all the entries directly to the left or directly below. In order to fill the table efficiently, we must choose an order that computes all those entries before $OptCost[i, j]$. There are at least three different orders that satisfy this constraint. The one that occurs to most people first is to scan through the table one diagonal at a time, starting with the trivial base cases $OptCost[i, i-1]$. The complete algorithm looks like this:

```
OptimalSearchTree(f[1..n]):
  InitF(f[1..n])
  for i ← 1 to n
      OptCost[i, i−1] ← 0
  for d ← 0 to n−1
      for i ← 1 to n−d
          ComputeOptCost(i, i+d)
  return OptCost[1, n]
```

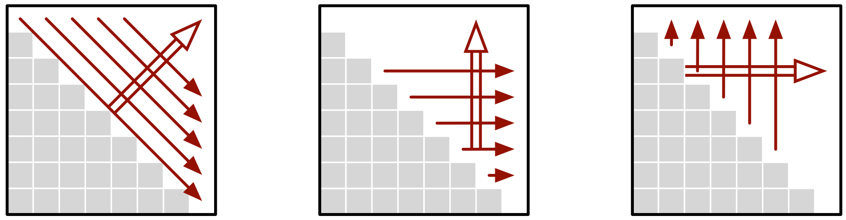
We could also traverse the array row by row from the bottom up, traversing each row from left to right, or column by column from left to right, traversing each columns from the bottom up.

```
OptimalSearchTree2(f[1..n]):
  InitF(f[1..n])
  for i ← n downto 1
      OptCost[i, i−1] ← 0
      for j ← i to n
          ComputeOptCost(i, j)
  return OptCost[1, n]
```

```
OptimalSearchTree3(f[1..n]):
  InitF(f[1..n])
  for j ← 0 to n
      OptCost[j+1, j] ← 0
      for i ← j downto 1
          ComputeOptCost(i, j)
  return OptCost[1, n]
```

No matter which of these orders we actually use, the resulting algorithm runs in $\boldsymbol{\Theta(n^3)}$ ***time*** and uses $\boldsymbol{\Theta(n^2)}$ ***space***. We could have predicted these space and time bounds directly from the original recurrence.

$$OptCost(i, j) = \begin{cases} 0 & \text{if } j = i - i \\ F(i, j) + \min\limits_{i \leq r \leq j} \big(OptCost(i, r-1) + OptCost(r+1, j)\big) & \text{otherwise} \end{cases}$$



Three different evaluation orders for the table $OptCost[i, j]$.

First, the function has two arguments, each of which can take on any value between 1 and n, so we probably need a table of size $O(n^2)$. Next, there are *three* variables in the recurrence (i, j, and r), each of which can take any value between 1 and n, so it should take us $O(n^3)$ time to fill the table.

5.8 The CYK Parsing Algorithm

In the same earlier lecture, we developed a recursive backtracking algorithm for parsing context-free languages. The input consists of a string w and a context-free grammar G in Chomsky normal form—meaning every production has the form $A \to a$, for some symbol a, or $A \to BC$, for some non-terminals B and C. Our task is to determine whether w is in the language generated by G.

Our backtracking algorithm recursively evaluates the boolean function *Generates?*(A, x), which equals TRUE if and only if string x can be derived from non-terminal A, using the following recurrence:

$$\textit{Generates?}(A, x) = \begin{cases} \textsc{True} & \text{if } |x| = 1 \text{ and } A \to x \\ \textsc{False} & \text{if } |x| = 1 \text{ and } A \not\to x \\ \displaystyle\bigvee_{A \to BC} \bigvee_{y \bullet z = x} \textit{Generates?}(B, y) \wedge \textit{Generates?}(C, z) & \text{otherwise} \end{cases}$$

This recurrence was transformed into a dynamic programming algorithm by Tadao Kasami in 1965, and again independently by Daniel Younger in 1967, and again independently by John Cocke in 1970, so naturally the resulting algorithm is known as "Cocke-Younger-Kasami", or more commonly ***the CYK algorithm***.

We can derive the CYK algorithm from the previous recurrence as follows. As usual for recurrences involving strings, we need to modify the function slightly to ease memoization. Fix the input string w, and then let *Generates?*$(A, i, j) =$ TRUE if and only if the substring $w[i..j]$ can be derived from non-terminal A. Now our earlier recurrence can be rewritten as follows:

$$\textit{Generates?}(A, i, j) = \begin{cases} \textsc{True} & \text{if } i = j \text{ and } A \to w[i] \\ \textsc{False} & \text{if } i = j \text{ and } A \not\to w[i] \\ \displaystyle\bigvee_{A \to BC} \bigvee_{k=i}^{j-1} \textit{Generates?}(B, i, k) \wedge \textit{Generates?}(C, k+1, j) & \text{otherwise} \end{cases}$$

This recurrence can be memoized into a three-dimensional boolean array $Gen[1..|\Gamma|, 1..n, 1..n]$, where the first dimension is indexed by the non-terminals Γ in the input grammar. Each entry $Gen[A, i, j]$ in this array depends on entries of the form $Gen[\,\cdot\,, i, k]$ for some $k < j$, or $Gen[\,\cdot\,, k+1, j]$ for some $k \geq i$. Thus, we can fill the array by increasing j in the outer loop, decreasing i in the middle loop, and considering non-terminals A in arbitrary order in the inner loop. The resulting dynamic programming algorithm runs in $\boldsymbol{O(n^3 \cdot |\Gamma|)}$ ***time***.

```
CYK(w, G):
  for i ← 1 to n
      for all non-terminals A
          if G contains the production A → w[i]
              Gen[A, i, i] ← TRUE
          else
              Gen[A, i, i] ← FALSE
  for j ← 1 to n
      for i ← n down to j + 1
          for all non-terminals A
              Gen[A, i, j] ← FALSE
          for all production rules A → BC
              for k ← i to j − 1
                  if Gen[B, i, k] and Gen[C, k + 1, j]
                      Gen[A, i, j] ← TRUE
  return Gen[S, 1, n]
```

5.9 Dynamic Programming on Trees

So far, all of our dynamic programming example use a multidimensional array to store the results of recursive subproblems. However, as the next example shows, this is not always the most appropriate date structure to use.

A ***independent set*** in a graph is a subset of the vertices that have no edges between them. Finding the largest independent set in an arbitrary graph is extremely hard; in fact, this is one of the canonical NP-hard problems described in another lecture note. But from some special cases of graphs, we can find the largest independent set efficiently. In particular, when the input graph is a tree (a connected and acyclic graph) with n vertices, we can compute the largest independent set in $O(n)$ time.

In the recursion notes, we saw a recursive algorithm for computing the size of the largest independent set in an arbitrary graph:

```
MAXIMUMINDSETSIZE(G):
  if G = ∅
      return 0
  v ← any node in G
  withv ← 1 + MAXIMUMINDSETSIZE(G \ N(v))
  withoutv ← MAXIMUMINDSETSIZE(G \ {v})
  return max{withv, withoutv}.
```

Here, $N(v)$ denotes the *neighborhood* of v: the set containing v and all of its neighbors. As we observed in the other lecture notes, this algorithm has a worst-case running time of $O(2^n \operatorname{poly}(n))$, where n is the number of vertices in the input graph.

Now suppose we require that the input graph is a tree; we will call this tree T instead of G from now on. We need to make a slight change to the algorithm to make it truly recursive. The subgraphs $T \setminus \{v\}$ and $T \setminus N(v)$ are forests, which may have more than one component. But the largest independent set in a disconnected graph is just the union of the largest independent sets in its components, so we can separately consider each tree in these forests. Fortunately, this has the added benefit of making the recursive algorithm more efficient, especially if we can choose the node v such that the trees are all significantly smaller than T. Here is the modified algorithm:

```
MaximumIndSetSize(T):
  if T = ∅
      return 0
  v ← any node in T
  withv ← 1
  for each tree T' in T \ N(v)
      withv ← withv + MaximumIndSetSize(T')
  withoutv ← 0
  for each tree T' in T \ {v}
      withoutv ← withoutv + MaximumIndSetSize(T')
  return max{withv, withoutv}.
```

Now let's try to memoize this algorithm. Each recursive subproblem considers a subtree (that is, a connected subgraph) of the original tree T. Unfortunately, a single tree T can have exponentially many subtrees, so we seem to be doomed from the start!

Fortunately, there's a degree of freedom that we have not yet exploited: *We get to choose the vertex v.* We need a recipe—an algorithm!—for choosing v in each subproblem that limits the number of different subproblems the algorithm considers. To make this work, we impose some additional structure on the original input tree. Specifically, we declare one of the vertices of T to be the *root*, and we orient all the edges of T away from that root. Then we let v be the root of the input tree; this choice guarantees that each recursive subproblem considers a *rooted* subtree of T. Each vertex in T is the root of exactly one subtree, so now the number of distinct subproblems is exactly n. We can further simplify the algorithm by only passing a single node instead of the entire subtree:

```
MaximumIndSetSize(v):
  withv ← 1
  for each grandchild x of v
      withv ← withv + MaximumIndSetSize(x)
  withoutv ← 0
  for each child w of v
      withoutv ← withoutv + MaximumIndSetSize(w)
  return max{withv, withoutv}.
```

What data structure should we use to store intermediate results? The most natural choice is the tree itself! Specifically, for each node v, we store the result of MaximumIndSetSize(v) in a new field $v.MIS$. (We *could* use an array, but then we'd have to add a new field to each node anyway, pointing to the corresponding array entry. Why bother?)

What's the running time of the algorithm? The non-recursive time associated with each node v is proportional to the number of children and grandchildren of v; this number can be very different from one vertex to the next. But we can turn the analysis around: Each vertex contributes a constant amount of time to its parent and its grandparent! Since each vertex has at most one parent and at most one grandparent, the total running time is $O(n)$.

What's a good order to consider the subproblems? The subproblem associated with any node v depends on the subproblems associated with the children and grandchildren of v. So we can visit the nodes in any order, provided that all children are visited before their parent. In particular, we can use a straightforward post-order traversal.

Here is the resulting dynamic programming algorithm. Yes, it's still recursive. I've swapped the evaluation of the with-v and without-v cases; we need to visit the kids first anyway, so why not consider the subproblem that depends directly on the kids first?

```
MaximumIndSetSize(v):
  withoutv ← 0
  for each child w of v
      withoutv ← withoutv + MaximumIndSetSize(w)
  withv ← 1
  for each grandchild x of v
      withv ← withv + x.MIS
  v.MIS ← max{withv, withoutv}
  return v.MIS
```

Another option is to store *two* values for each rooted subtree: the size of the largest independent set *that includes the root*, and the size of the largest independent set *that excludes the root*. This gives us an even simpler algorithm, with the same $O(n)$ running time.

```
MaximumIndSetSize(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
      v.MISno ← v.MISno + MaximumIndSetSize(w)
      v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}
```

Exercises

Sequences/Arrays

1. In a previous life, you worked as a cashier in the lost Antarctican colony of Nadira, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource in Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadira, called Dream Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365.[10]

   (a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream Dollar bills than the minimum possible.

   (b) Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream Dollars. (Don't worry about making your algorithm fast; just make sure it's correct.)

   (c) Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream Dollars. (This one needs to be fast.)

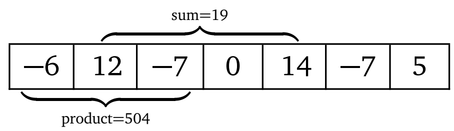
2. Suppose you are given an array $A[1..n]$ of numbers, which may be positive, negative, or zero, and which are ***not*** necessarily integers.

[10]For more details on the history and culture of Nadira, including images of the various denominations of Dream Dollars, see http://moneyart.biz/dd/.

(a) Describe and analyze an algorithm that finds the largest sum of of elements in a contiguous subarray $A[i \,..\, j]$.

(b) Describe and analyze an algorithm that finds the largest *product* of of elements in a contiguous subarray $A[i \,..\, j]$.

For example, given the array $[-6, 12, -7, 0, 14, -7, 5]$ as input, your first algorithm should return the integer 19, and your second algorithm should return the integer 504.



For the sake of analysis, assume that comparing, adding, or multiplying any pair of numbers takes $O(1)$ time.

[Hint: Problem (a) has been a standard computer science interview question since at least the mid-1980s. You can find many correct solutions on the web; the problem even has its own Wikipedia page! But at least in 2013, the few solutions I found on the web for problem (b) were all either slower than necessary or incorrect.]

3. This series of exercises asks you to develop efficient algorithms to find optimal *subsequences* of various kinds. A subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings `C`, `DAMN`, `YAIOAI`, and `DYNAMICPROGRAMMING` are all subsequences of the string `DYNAMICPROGRAMMING`.

(a) Let $A[1 \,..\, m]$ and $B[1 \,..\, n]$ be two arbitrary arrays. A ***common subsequence*** of A and B is another sequence that is a subsequence of both A and B. Describe an efficient algorithm to compute the length of the *longest* common subsequence of A and B.

(b) Let $A[1 \,..\, m]$ and $B[1 \,..\, n]$ be two arbitrary arrays. A ***common supersequence*** of A and B is another sequence that contains both A and B as subsequences. Describe an efficient algorithm to compute the length of the *shortest* common supersequence of A and B.

(c) Call a sequence $X[1 \,..\, n]$ of numbers ***bitonic*** if there is an index i with $1 < i < n$, such that the prefix $X[1 \,..\, i]$ is increasing and the suffix $X[i \,..\, n]$ is decreasing. Describe an efficient algorithm to compute the length of the longest bitonic subsequence of an arbitrary array A of integers.

(d) Call a sequence $X[1 \,..\, n]$ of numbers ***oscillating*** if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Describe an efficient algorithm to compute the length of the longest oscillating subsequence of an arbitrary array A of integers.

(e) Describe an efficient algorithm to compute the length of the shortest oscillating supersequence of an arbitrary array A of integers.

(f) Call a sequence $X[1 \,..\, n]$ of numbers ***convex*** if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Describe an efficient algorithm to compute the length of the longest convex subsequence of an arbitrary array A of integers.

(g) Call a sequence $X[1 \,..\, n]$ of numbers ***weakly increasing*** if each element is larger than the average of the two previous elements; that is, $2 \cdot X[i] > X[i-1] + X[i-2]$ for all

$i > 2$. Describe an efficient algorithm to compute the length of the longest weakly increasing subsequence of an arbitrary array A of integers.

(h) Call a sequence $X[1..n]$ of numbers ***double-increasing*** if $X[i] > X[i-2]$ for all $i > 2$. (In other words, a semi-increasing sequence is a perfect shuffle of two increasing sequences.) Describe an efficient algorithm to compute the length of the longest double-increasing subsequence of an arbitrary array A of integers.

*(i) Recall that a sequence $X[1..n]$ of numbers is *increasing* if $X[i] < X[i+1]$ for all i. Describe an efficient algorithm to compute the length of the *longest common increasing subsequence* of two given arrays of integers. For example, $\langle 1,4,5,6,7,9\rangle$ is the longest common increasing subsequence of the sequences $\langle 3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3\rangle$ and $\langle 1,4,1,4,2,1,3,5,6,2,3,7,3,0,9,5\rangle$.

4. Describe an algorithm to compute the number of times that one given array $X[1..k]$ appears as a subsequence of another given array $Y[1..n]$. For example, if all characters in X and Y are equal, your algorithm should return $\binom{n}{k}$. For purposes of analysis, assume that adding two ℓ-bit integers requires $\Theta(\ell)$ time.

5. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

   Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

   (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.

   (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

   (c) Five years later, Elmo has become a *much* stronger player. Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a *perfect* opponent.

6. It's almost time to show off your flippin' sweet dancing skills! Tomorrow is the big dance contest you've been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You've obtained an advance copy of the the list of n songs that the judges will play during the contest, in chronological order.

   You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly *Score*$[k]$ points, but then you will be physically unable to dance for the next *Wait*$[k]$ songs (that is, you cannot dance to songs $k+1$ through $k+$*Wait*$[k]$). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.

7. You are driving a bus along a highway, full of rowdy, hyper, thirsty students and a soda fountain machine. Each minute that a student is on your bus, that student drinks one ounce of soda. Your goal is to drop the students off quickly, so that the total amount of soda consumed by all students is as small as possible.

   You know how many students will get off of the bus at each exit. Your bus begins somewhere along the highway (probably not at either end) and move s at a constant speed of 37.4 miles per hour. You must drive the bus along the highway; however, you may drive forward to one exit then backward to an exit in the opposite direction, switching as often as you like. (You can stop the bus, drop off students, and turn around instantaneously.)

   Describe an efficient algorithm to drop the students off so that they drink as little soda as possible. Your input consists of the bus route (a list of the exits, together with the travel time between successive exits), the number of students you will drop off at each exit, and the current location of your bus (which you may assume is an exit).

8. A palindrome is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`.

   (a) Describe and analyze an algorithm to find the length of the *longest subsequence* of a given string that is also a palindrome. For example, the longest palindrome subsequence of `MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM` is `MHYMRORMYHM`, so given that string as input, your algorithm should output the number 11.

   (b) Describe and analyze an algorithm to find the length of the *shortest supersequence* of a given string that is also a palindrome. For example, the shortest palindrome supersequence of `TWENTYONE` is `TWENTOYOTNEWT`, so given the string `TWENTYONE` as input, your algorithm should output the number 13.

   (c) Any string can be decomposed into a sequence of palindromes. For example, the string `BUBBASEESABANANA` ("Bubba sees a banana.") can be broken into palindromes in the following ways (and many others):

   `BUB` • `BASEESAB` • `ANANA`

   `B` • `U` • `BB` • `A` • `SEES` • `ABA` • `NAN` • `A`

   `B` • `U` • `BB` • `A` • `SEES` • `A` • `B` • `ANANA`

   `B` • `U` • `B` • `B` • `A` • `S` • `E` • `E` • `S` • `A` • `B` • `A` • `N` • `ANA`

   Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string `BUBBASEESABANANA`, your algorithm would return the integer 3.

9. Suppose you have a black-box subroutine QUALITY that can compute the 'quality' of any given string $A[1..k]$ in $O(k)$ time. For example, the quality of a string might be 1 if the string is a Québecois curse word, and 0 otherwise.

   Given an arbitrary input string $T[1..n]$, we would like to break it into contiguous substrings, such that the total quality of all the substrings is as large as possible. For example, the string `SAINTCIBOIREDESACRAMENTDECRISSE` can be decomposed into the substrings `SAINT` • `CIBOIRE` • `DE` • `SACRAMENT` • `DE` • `CRISSE`, of which three (or possibly four) are *sacres*.

   Describe an algorithm that breaks a string into substrings of maximum total quality, using the QUALITY subroutine.

10. (a) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

    (b) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

    (c) Suppose we are given a set L of n line segments in the plane, where the endpoints of each segment lie on the unit circle $x^2 + y^2 = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

    (d) Suppose we are given a set L of n line segments in the plane, where the endpoints of each segment lie on the unit circle $x^2 + y^2 = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

11. Let P be a set of n points evenly distributed on the unit circle, and let S be a set of m line segments with endpoints in P. The endpoints of the m segments are *not* necessarily distinct; n could be significantly smaller than $2m$.

    (a) Describe an algorithm to find the size of the largest subset of segments in S such that every pair is disjoint. Two segments are disjoint if they do not intersect even at their endpoints.

    (b) Describe an algorithm to find the size of the largest subset of segments in S such that every pair is interior-disjoint. Two segments are interior-disjoint if their intersection is either empty or an endpoint of both segments.

    (c) Describe an algorithm to find the size of the largest subset of segments in S such that every pair intersects.

    (d) Describe an algorithm to find the size of the largest subset of segments in S such that every pair crosses. Two segments cross if they intersect but not at their endpoints.

    For full credit, all four algorithms should run in $O(mn)$ time.

12. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string `BANANAANANAS` is a shuffle of the strings `BANANA` and `ANANAS` in several different ways.

$$\mathtt{BANANA}_{\mathtt{ANANAS}} \qquad \mathtt{BAN}_{\mathtt{ANA}}{}^{\mathtt{ANA}}{}_{\mathtt{NAS}} \qquad \mathtt{B}_{\mathtt{AN}}{}^{\mathtt{AN}}{}_{\mathtt{A}}{}^{\mathtt{A}}{}_{\mathtt{NA}}{}^{\mathtt{NA}}{}_{\mathtt{S}}$$

Similarly, the strings `PRODGYRNAMAMMIINCG` and `DYPRONGARMAMMICING` are both shuffles of `DYNAMIC` and `PROGRAMMING`:

$$\mathtt{PRO}^{\mathtt{D}}\mathtt{G}^{\mathtt{Y}}\mathtt{R}^{\mathtt{NAM}}\mathtt{AMMI}^{\mathtt{I}}\mathtt{N}^{\mathtt{C}}\mathtt{G} \qquad {}^{\mathtt{DY}}\mathtt{PRO}^{\mathtt{N}}\mathtt{G}^{\mathtt{A}}\mathtt{R}^{\mathtt{M}}\mathtt{AMM}^{\mathtt{IC}}\mathtt{ING}$$

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

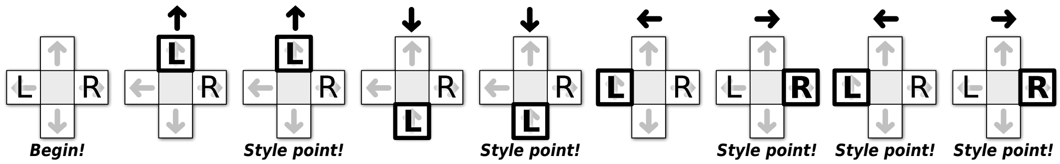
13. Describe and analyze an efficient algorithm to find the length of the longest contiguous substring that appears both forward and backward in an input string $T[1..n]$. The forward and backward substrings must not overlap. Here are several examples:

   - Given the input string `ALGORITHM`, your algorithm should return 0.
   - Given the input string $\mathtt{\underline{R}ECU\underline{R}SION}$, your algorithm should return 1, for the substring `R`.
   - Given the input string $\mathtt{R\underline{EDI}V\underline{IDE}}$, your algorithm should return 3, for the substring `EDI`. (The forward and backward substrings must not overlap!)
   - Given the input string $\mathtt{D\underline{YNAM}ICPROGRAMMING\underline{MANY}TIMES}$, your algorithm should return 4, for the substring `YNAM`. (In particular, it should *not* return 6, for the subsequence `YNAMIR`).

14. ***Dance Dance Revolution*** is a dance video game, first introduced in Japan by Konami in 1998. Players stand on a platform marked with four arrows, pointing forward, back, left, and right, arranged in a cross pattern. During play, the game plays a song and scrolls a sequence of n arrows (←, ↑, ↓, or →) from the bottom to the top of the screen. At the precise moment each arrow reaches the top of the screen, the player must step on the corresponding arrow on the dance platform. (The arrows are timed so that you'll step with the beat of the song.)

   You are playing a variant of this game called "Vogue Vogue Revolution", where the goal is to play perfectly but move as little as possible. When an arrow reaches the top of the screen, if one of your feet is already on the correct arrow, you are awarded one style point for maintaining your current pose. If neither foot is on the right arrow, you must move one (and *only* one) of your feet from its current location to the correct arrow on the platform. If you ever step on the wrong arrow, or fail to step on the correct arrow, or move more than one foot at a time, or move either foot when you are already standing on the correct arrow, all your style points are taken away and you lose the game.

   How should you move your feet to maximize your total number of style points? For purposes of this problem, assume you always start with you left foot on ← and you right foot on →, and that you've memorized the entire sequence of arrows. For example, if the sequence is ↑↑↓↓←→←→, you can earn 5 style points by moving you feet as shown below:



(a) ***Prove*** that for *any* sequence of n arrows, it is possible to earn at least $n/4 - 1$ style points.

(b) Describe an efficient algorithm to find the maximum number of style points you can earn during a given VVR routine. The input to your algorithm is an array $Arrow[1..n]$ containing the sequence of arrows.

15. Consider the following solitaire form of Scrabble. We begin with a fixed, finite sequence of tiles; each tile contains a letter and a numerical value. At the start of the game, we draw the seven tiles from the sequence and put them into our hand. In each turn, we form an English word from some or all of the tiles in our hand, place those tiles on the table, and receive the total value of those tiles as points. If no English word can be formed from the tiles in our hand, the game immediately ends. Then we repeatedly draw the next tile from the start of the sequence until either (a) we have seven tiles in our hand, or (b) the sequence is empty. (Sorry, no double/triple word/letter scores, bingos, blanks, or passing.) Our goal is to obtain as many points as possible.

For example, suppose we are given the tile sequence

$\boxed{\texttt{I}_2}\boxed{\texttt{N}_2}\boxed{\texttt{X}_8}\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}\boxed{\texttt{A}_1}\boxed{\texttt{D}_3}\boxed{\texttt{U}_5}\boxed{\texttt{D}_3}\boxed{\texttt{I}_2}\boxed{\texttt{D}_3}\boxed{\texttt{K}_8}\boxed{\texttt{U}_5}\boxed{\texttt{B}_4}\boxed{\texttt{L}_2}\boxed{\texttt{A}_1}\boxed{\texttt{K}_8}\boxed{\texttt{H}_5}\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}$.

Then we can earn 68 points as follows:

- We initially draw $\boxed{\texttt{I}_2}\boxed{\texttt{N}_2}\boxed{\texttt{X}_8}\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}\boxed{\texttt{A}_1}\boxed{\texttt{D}_3}$.
- Play the word $\boxed{\texttt{N}_2}\boxed{\texttt{A}_1}\boxed{\texttt{I}_2}\boxed{\texttt{A}_1}\boxed{\texttt{D}_3}$ for 9 points, leaving $\boxed{\texttt{N}_2}\boxed{\texttt{X}_8}$ in our hand.
- Draw the next five tiles $\boxed{\texttt{U}_5}\boxed{\texttt{D}_3}\boxed{\texttt{I}_2}\boxed{\texttt{D}_3}\boxed{\texttt{K}_8}$.
- Play the word $\boxed{\texttt{U}_5}\boxed{\texttt{N}_2}\boxed{\texttt{D}_3}\boxed{\texttt{I}_2}\boxed{\texttt{D}_3}$ for 15 points, leaving $\boxed{\texttt{K}_8}\boxed{\texttt{X}_8}$ in our hand.
- Draw the next five tiles $\boxed{\texttt{U}_5}\boxed{\texttt{B}_4}\boxed{\texttt{L}_2}\boxed{\texttt{A}_1}\boxed{\texttt{K}_8}$.
- Play the word $\boxed{\texttt{B}_4}\boxed{\texttt{U}_5}\boxed{\texttt{L}_2}\boxed{\texttt{K}_8}$ for 19 points, leaving $\boxed{\texttt{K}_8}\boxed{\texttt{X}_8}\boxed{\texttt{A}_1}$ in our hand.
- Draw the next three tiles $\boxed{\texttt{H}_5}\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}$, emptying the list.
- Play the word $\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}\boxed{\texttt{K}_8}\boxed{\texttt{H}_5}$ for 16 points, leaving $\boxed{\texttt{X}_8}\boxed{\texttt{A}_1}$ in our hand.
- Play the word $\boxed{\texttt{A}_1}\boxed{\texttt{X}_8}$ for 9 points, emptying our hand and ending the game.

(a) Suppose you are given as input two arrays $Letter[1..n]$, containing a sequence of letters between $\texttt{A}$ and $\texttt{Z}$, and $Value[\texttt{A}..\texttt{Z}]$, where $Value[\ell]$ is the value of letter ℓ. Design and analyze an efficient algorithm to compute the maximum number of points that can be earned from the given sequence of tiles.

(b) Now suppose two tiles with the same letter can have different values; you are given two arrays $Letter[1..n]$ and $Value[1..n]$. Design and analyze an efficient algorithm to compute the maximum number of points that can be earned from the given sequence of tiles.

In both problems, the output is a single number: the maximum possible score. Assume that you can find all English words that can be made from any set of at most seven tiles, along with the point values of those words, in $O(1)$ time.

16. Suppose you are given a DFA $M = (\{0, 1\}, Q, s, A, \delta)$ and a binary string $w \in \{0, 1\}^*$.

(a) Describe and analyze an algorithm that computes the longest subsequence of w that is accepted by M, or correctly reports that M does not accept any subsequence of w.

*(b) Describe and analyze an algorithm that computes the *shortest supersequence* of w that is accepted by M, or correctly reports that M does not accept any supersequence of w. *[Hint: Careful!]*

Analyze both of your algorithms in terms of the parameters $n = |w|$ and $k = |Q|$.

17. *Vankin's Mile* is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

For example, given the grid below, the player can score $8 - 6 + 7 - 3 + 4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is *not* the best possible score for these values.)

−1	7	−8	10	−5
−4	−9	8	−6	0
5	−2	−6	−6	7
−7	4	7	−3	−3
7	1	−6	4	−9

(a) Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Mile, given the $n \times n$ array of values as input.

(b) In the European version of this game, appropriately called *Vankin's Kilometer*, the player can move the token either one square down, one square right, *or one square left* in each turn. However, to prevent infinite scores, the token cannot land on the same square more than once. Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Kilometer, given the $n \times n$ array of values as input.[11]

[11] If we also allowed upward movement, the resulting game (Vankin's Fathom?) would be Ebay-hard.

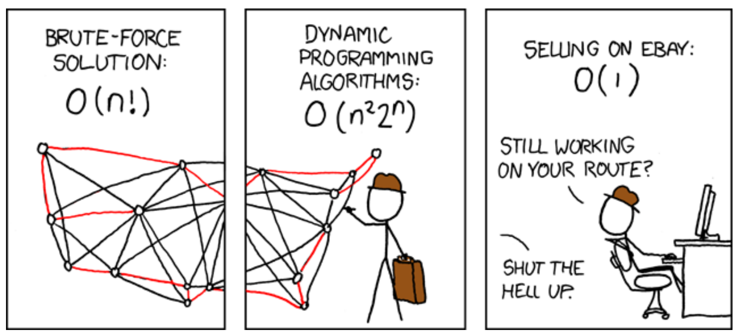
18. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..n, 1..n]$ of 0s and 1s. A *solid block* in M is a subarray of the form $M[i..i', j..j']$ containing only 1-bits. A solid block is square if it has the same number of rows and columns.

    (a) Describe an algorithm to find the maximum area of a solid *square* block in M in $O(n^2)$ time.

    (b) Describe an algorithm to find the maximum area of a solid block in M in $O(n^3)$ time.

    *(c) Describe an algorithm to find the maximum area of a solid block in M in $O(n^2)$ time.

*19. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \operatorname{poly}(n))$ time. Given an undirected n-vertex graph G with weighted edges, your algorithm should return the weight of the lightest cycle in G that visits every vertex exactly once, or ∞ if G has no such cycles. *[Hint: The obvious recursive algorithm takes $O(n!)$ time.]*



— Randall Munroe, *xkcd* (http://xkcd.com/399/)

*20. Let $\mathscr{A} = \{A_1, A_2, \ldots, A_n\}$ be a finite set of strings over some fixed alphabet Σ. An *edit center* for $\mathscr{A}$ is a string $C \in \Sigma^*$ such that the maximum edit distance from C to any string in $\mathscr{A}$ is as small as possible. The *edit radius* of $\mathscr{A}$ is the maximum edit distance from an edit center to a string in $\mathscr{A}$. A set of strings may have several edit centers, but its edit radius is unique.

$$\mathit{EditRadius}(\mathscr{A}) = \min_{C \in \Sigma^*} \max_{A \in \mathscr{A}} \mathit{Edit}(A, C) \qquad \mathit{EditCenter}(\mathscr{A}) = \arg\min_{C \in \Sigma^*} \max_{A \in \mathscr{A}} \mathit{Edit}(A, C)$$

   (a) Describe and analyze an efficient algorithm to compute the edit radius of three given strings.

   (b) Describe and analyze an efficient algorithm to approximate the edit radius of an arbitrary set of strings within a factor of 2. (Computing the *exact* edit radius is NP-hard unless the number of strings is fixed.)

$^\bigstar$21. Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. An ***digital subsequence*** of D is a sequence of positive integers composed in the usual way from disjoint substrings of D. For example, 3, 4, 5, 6, 8, 9, 32, 38, 46, 64, 83, 279 is a digital subsequence of the first several digits of π:

$$\underbrace{3},1,\underbrace{4},1,\underbrace{5},9,2,\underbrace{6},5,3,5,\underbrace{8},\underbrace{9},7,9,\underbrace{3,2},\underbrace{3,8},\underbrace{4,6},2,\underbrace{6,4},3,3,\underbrace{8,3},\underbrace{2,7,9}$$

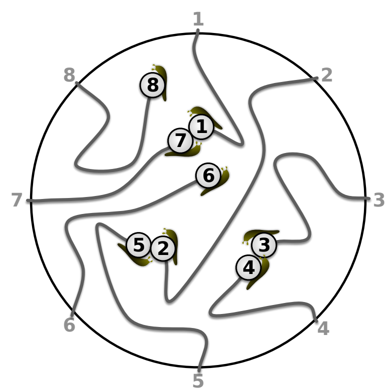
The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the preceding example has length 12. As usual, a digital subsequence is ***increasing*** if each number is larger than its predecessor.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D. *[Hint: Be careful about your computational assumptions. How long does it take to compare two k-digit numbers?]*

For full credit, your algorithm should run in $O(n^4)$ time; faster algorithms are worth extra credit. The fastest algorithm I know for this problem runs in $O(n^2 \log n)$ time; achieving this bound requires several tricks, both in the algorithm and in its analysis.

Splitting Sequences/Arrays

22. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.



The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates.
The organizers must pay $M[3,4] + M[2,5] + M[1,7]$.

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i, j] = M[j, i]$ is the reward to be paid if snails i and j meet.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

23. Suppose you are given a sequence of integers separated by + and × signs; for example:

$$1 + 3 \times 2 \times 0 + 1 \times 6 + 7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$(1 + (3 \times 2)) \times 0 + (1 \times 6) + 7 = 13$$
$$((1 + (3 \times 2 \times 0) + 1) \times 6) + 7 = 19$$
$$(1 + 3) \times 2 \times (0 + 1) \times (6 + 7) = 208$$

(a) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, assuming all integers in the input are *positive*. *[Hint: This is easy.]*

(b) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, assuming all integers in the input are *non-negative*.

(c) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, with no further restrictions on the input.

Assume any arithmetic operation takes $O(1)$ time.

24. Suppose you are given a sequence of integers separated by $+$ and $-$ signs; for example:

$$1 + 3 - 2 - 5 + 1 - 6 + 7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$1 + 3 - 2 - 5 + 1 - 6 + 7 = -1$$
$$(1 + 3 - (2 - 5)) + (1 - 6) + 7 = 9$$
$$(1 + (3 - 2)) - (5 + 1) - (6 + 7) = -17$$

Describe and analyze an algorithm to compute, given a list of integers separated by $+$ and $-$ signs, the maximum possible value the expression can take by adding parentheses.

You may only use parentheses to group additions and subtractions; in particular, you are not allowed to create implicit multiplication as in $1 + 3(-2)(-5) + 1 - 6 + 7 = 33$.

25. A ***basic arithmetic expression*** is composed of characters from the set $\{1, +, \times\}$ and parentheses. Almost every integer can be represented by more than one basic arithmetic expression. For example, all of the following basic arithmetic expression represent the integer 14:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1 + 1) \times (1 + 1 + 1 + 1 + 1)) + ((1 + 1) \times (1 + 1))$$
$$(1 + 1) \times (1 + 1 + 1 + 1 + 1 + 1 + 1)$$
$$(1 + 1) \times (((1 + 1 + 1) \times (1 + 1)) + 1)$$

Describe and analyze an algorithm to compute, given an integer n as input, the minimum number of 1's in a basic arithmetic expression whose value is n. The number of parentheses doesn't matter, just the number of 1's. For example, when $n = 14$, your algorithm should return 8, for the final expression above. For full credit, the running time of your algorithm should be bounded by a small polynomial function of n.

26. After graduating from UIUC, you have decided to join the Wall Street Bank ***Long Live Boole***. The managing director of the bank, Eloob Egroeg, is a genius mathematician who worships George Boole (the inventor of Boolean Logic) every morning before leaving for the office. The first day of every hired employee is a 'solve-or-die' day where s/he has to solve one of the problems posed by Eloob within 24 hours. Those who fail to solve the problem are fired immediately!

    Entering into the bank for the first time, you notice that the offices of the employees are organized in a straight row, with a large T or F printed on the door of each office. Furthermore, between each adjacent pair of offices, there is a board marked by one of the symbols $\wedge, \vee$, or $\oplus$. When you ask about these arcane symbols, Eloob confirms that T and F represent the boolean values TRUE and FALSE, and the symbols on the boards represent the standard boolean operators AND, OR, and XOR. He also explains that these letters and symbols describe whether certain combinations of employees can work together successfully. At the start of any new project, Eloob hierarchically clusters his employees by adding parentheses to the sequence of symbols, to obtain an unambiguous boolean expression. The project is successful if this parenthesized boolean expression evaluates to T.

    For example, if the bank has three employees, and the sequence of symbols on and between their doors is $T \wedge F \oplus T$, there is exactly one successful parenthesization scheme: $(T \wedge (F \oplus T))$. However, if the list of door symbols is $F \wedge T \oplus F$, there is no way to add parentheses to make the project successful.

    Eloob finally poses your solve-or-die question: Describe and algorithm to decide whether a given sequence of symbols can be parenthesized so that the resulting boolean expression evaluates to T. The input to your algorithm is an array $S[0\,..\,2n]$, where $S[i] \in \{T, F\}$ when i is even, and $S[i] \in \{\vee, \wedge, \oplus\}$ when i is odd.

27. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the ith word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we must insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of white-space between any two words on the same line. For example, the width of *the paragraph you are reading right now* is exactly $6\frac{4}{33}$ inches or (assuming a display resolution of 600 pixels per inch) exactly $3672\frac{8}{11}$ pixels. (Sometimes TEX is weird. But thanks to anti-aliasing, fractional pixel widths are fine.)

    Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line, not counting the one pixel required between every adjacent pair of words. Specifically, if a line contains words i through j, then the slop of that line is $\left(P - j + i - \sum_{k=i}^{j} p_k\right)^3$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

28. You have mined a large slab of marble from your quarry. For simplicity, suppose the marble slab is a rectangle measuring n inches in height and m inches in width. You want to cut the slab into smaller rectangles of various sizes—some for kitchen countertops, some for large

sculpture projects, others for memorial headstones. You have a marble saw that can make either horizontal or vertical cuts across any rectangular slab. At any time, you can query the spot price $P[x, y]$ of an x-inch by y-inch marble rectangle, for any positive integers x and y. These prices will vary with demand, so do not make any assumptions about them; in particular, larger rectangles may have much smaller spot prices. Given the spot prices, describe an algorithm to compute how to subdivide an $n \times m$ marble slab to maximize your profit.

29. A string w of parentheses **(** and **)** and brackets **[** and **]** is ***balanced*** if it satisfies one of the following conditions:

    - w is the empty string.
    - $w = \mathbf{(}x\mathbf{)}$ for some balanced string x
    - $w = \mathbf{[}x\mathbf{]}$ for some balanced string x
    - $w = xy$ for some balanced strings x and y

    For example, the string

    $$w = \mathbf{([()][]())[()()]()}$$

    is balanced, because $w = xy$, where

    $$x = \mathbf{(\ [()]\ []\ ()\ )} \qquad \text{and} \qquad y = \mathbf{[\ ()\ ()\ ]\ ()}.$$

    (a) Describe and analyze an algorithm to determine whether a given string of parentheses and brackets is balanced.

    (b) Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets.

    (c) Describe and analyze an algorithm to compute the length of a shortest balanced supersequence of a given string of parentheses and brackets.

    (d) Describe and analyze an algorithm to compute the minimum edit distance from a given string of parentheses and brackets to a balanced string of parentheses and brackets.

    For each problem, your input is an array $w[1..n]$, where $w[i] \in \{\mathbf{(}, \mathbf{)}, \mathbf{[}, \mathbf{]}\}$ for every index i.

30. Congratulations! Your research team has just been awarded a \$50M multi-year project, jointly funded by DARPA, Google, and McDonald's, to produce DWIM: The first compiler to read programmers' minds! Your proposal and your numerous press releases all promise that DWIM will automatically correct errors in any given piece of code, while modifying that code as little as possible. Unfortunately, now it's time to start actually making the damn thing work.

    As a warmup exercise, you decide to tackle the following necessary subproblem. Recall that the *edit distance* between two strings is the minimum number of single-character insertions, deletions, and replacements required to transform one string into the other. An *arithmetic expression* is a string w such that

    - w is a string of one or more decimal digits,

- $w = (x)$ for some arithmetic expression x, or
- $w = x \diamond y$ for some arithmetic expressions x and y and some binary operator $\diamond$.

Suppose you are given a string of tokens from the alphabet $\{\#, \diamond, (,)\}$, where # represents a decimal digit and $\diamond$ represents a binary operator. Describe an algorithm to compute the minimum edit distance from the given string to an arithmetic expression.

31. Let P be a set of points in the plane in *convex position*. Intuitively, if a rubber band were wrapped around the points, then every point would touch the rubber band. More formally, for any point p in P, there is a line that separates p from the other points in P. Moreover, suppose the points are indexed $P[1], P[2], \ldots, P[n]$ in counterclockwise order around the 'rubber band', starting with the leftmost point $P[1]$.

    This problem asks you to solve a special case of the traveling salesman problem, where the salesman must visit every point in P, and the cost of moving from one point $p \in P$ to another point $q \in P$ is the Euclidean distance $|pq|$.

    (a) Describe a simple algorithm to compute the shortest *cyclic* tour of P.

    (b) A *simple* tour is one that never crosses itself. Prove that the shortest tour of P must be simple.

    (c) Describe and analyze an efficient algorithm to compute the shortest tour of P that starts at the leftmost point $P[1]$ and ends at the rightmost point $P[r]$.

    (d) Describe and analyze an efficient algorithm to compute the shortest tour of P, with no restrictions on the endpoints.

32. (a) Describe and analyze an efficient algorithm to determine, given a string w and a regular expression R, whether $w \in L(R)$.

    (b) *Generalized* regular expressions allow the binary operator $\cap$ (intersection) and the unary operator $\neg$ (complement), in addition to the usual concatenation, $+$ (or), and * (Kleene closure) operators. NFA constructions and Kleene's theorem imply that any generalized regular expression E represents a regular language $L(E)$.

    Describe and analyze an efficient algorithm to determine, given a string w and a generalized regular expression E, whether $w \in L(E)$.

    In both problems, assume that you are actually given a parse tree for the (generalized) regular expression, not just a string.

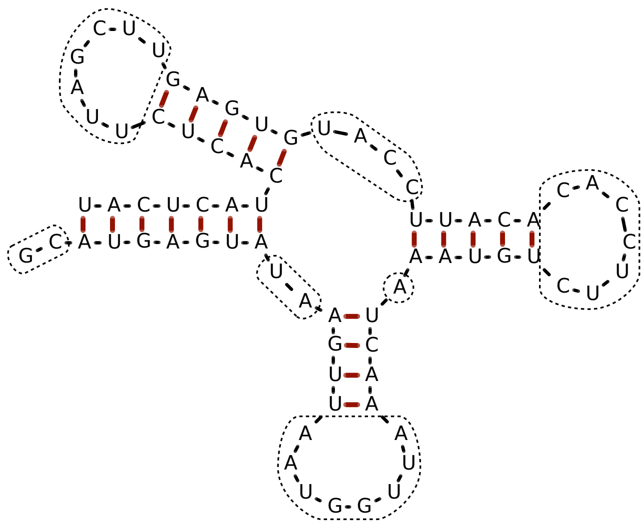
33. Ribonucleic acid (RNA) molecules are long chains of millions of nucleotides or *bases* of four different types: adenine (A), cytosine (C), guanine (G), and uracil (U). The *sequence* of an RNA molecule is a string $b[1..n]$, where each character $b[i] \in \{A, C, G, U\}$ corresponds to a base. In addition to the chemical bonds between adjacent bases in the sequence, hydrogen bonds can form between certain pairs of bases. The set of bonded base pairs is called the *secondary structure* of the RNA molecule.

    We say that two base pairs (i, j) and (i', j') with $i < j$ and $i' < j'$ ***overlap*** if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most base pairs are non-overlapping. Overlapping base pairs create so-called *pseudoknots* in the secondary structure, which are essential for some RNA functions, but are more difficult to predict.

Suppose we want to predict the best possible secondary structure for a given RNA sequence. We will adopt a drastically simplified model of secondary structure:

- Each base can be paired with at most one other base.
- Only A-U pairs and C-G pairs can bond.
- Pairs of the form $(i, i+1)$ and $(i, i+2)$ cannot bond.
- Overlapping base pairs cannot bond.

The last restriction allows us to visualize RNA secondary structure as a sort of fat tree.



Example RNA secondary structure with 21 base pairs, indicated by heavy red lines.
Gaps are indicated by dotted curves. This structure has score $2^2 + 2^2 + 8^2 + 1^2 + 7^2 + 4^2 + 7^2 = 187$

(a) Describe and analyze an algorithm that computes the maximum possible *number* of bonded base pairs in a secondary structure for a given RNA sequence.

(b) A *gap* in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the *score* of a secondary structure to be the sum of the *squares* of the gap lengths. (This score function is utterly fictional; real RNA structure prediction requires *much* more complicated scoring functions.) Describe and analyze an algorithm that computes the minimum possible score of a secondary structure for a given RNA sequence.

34. A standard method to improve the cache performance of search trees is to pack more search keys and subtrees into each node. A ***B*-tree** is a rooted tree in which each internal node stores up to B keys and pointers to up to $B+1$ children, each the root of a smaller B-tree. Specifically, each node v stores three fields:

- a positive integer $v.d \le B$,
- a *sorted* array $v.key[1\,..\,v.d]$, and
- an array $v.child[0\,..\,v.d]$ of child pointers.

In particular, the number of child pointers is always exactly one more than the number of keys.

Each pointer $v.child[i]$ is either Null or a pointer to the root of a B-tree whose keys are all larger than $v.key[i]$ and smaller than $v.key[i+1]$. In particular, all keys in the leftmost subtree $v.child[0]$ are smaller than $v.key[1]$, and all keys in the rightmost subtree $v.child[v.d]$ are larger than $v.key[v.d]$.

Intuitively, you should have the following picture in mind:

$$
\begin{array}{ccccccccc}
[\ \bullet & < key[1] < & \bullet & < key[2] < & \bullet & \cdots & \bullet & < key[d] < & \bullet\ ] \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
T_0 & & T_1 & & T_2 & \cdots & T_{d-1} & & T_d
\end{array}
$$

Here T_i is the subtree pointed to by $child[i]$.

The ***cost*** of searching for a key x in a B-tree is the number of nodes in the path from the root to the node containing x as one of its keys. A 1-tree is just a standard binary search tree.

Fix an arbitrary positive integer $B > 0$. (I suggest $B = 8$.) Suppose your are given a sorted array $A[1,\ldots,n]$ of search keys and a corresponding array $F[1,\ldots,n]$ of frequency counts, where $F[i]$ is the number of times that we will search for $A[i]$. Your task is to describe and analyze an efficient algorithm to find a B-tree that minimizes the total cost of searching for the given keys with the given frequencies.

(a) Describe a polynomial-time algorithm for the special case $B = 2$.

(b) Describe an algorithm for arbitrary B that runs in $O(n^{B+c})$ time for some fixed integer c.

(c) Describe an algorithm for arbitrary B that runs in $O(n^c)$ time for some fixed integer c that does *not* depend on B.

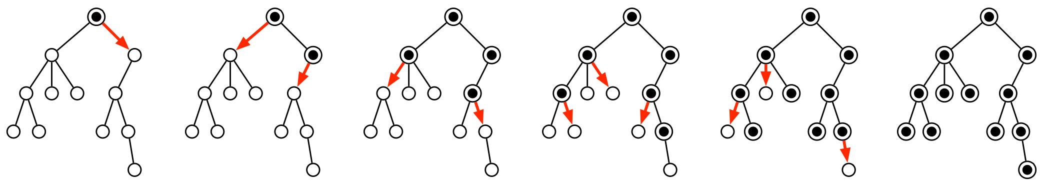
A few comments about *B*-trees. Normally, B-trees are required to satisfy two additional constraints, which guarantee a worst-case search cost of $O(\log_B n)$: Every leaf must have exactly the same depth, and every node except possibly the root must contain at least $B/2$ keys. However, in this problem, we are not interested in optimizing the *worst-case* search cost, but rather the *total* cost of a sequence of searches, so we will not impose these additional constraints.

In most large database systems, the parameter B is chosen so that each node exactly fits in a cache line. Since the entire cache line is loaded into cache anyway, and the cost of loading a cache line exceeds the cost of searching within the cache, the running time is dominated by the number of cache faults. This effect is even more noticeable if the data is too big to fit in RAM; then the cost is dominated by the number of *page faults*, and B should be roughly the size of a page. In extreme cases, the data is too large even to fit on disk (or flash-memory "disk") and is instead distributed on a bank of magnetic tape cartridges, in which case the cost is dominated by the number of *tape faults*. (I invite anyone who thinks tape is dead to visit a supercomputing center like Blue Waters.) In principle, your data might be so large that the cost of searching is actually dominated by the number of *FedEx faults*. (See https://what-if.xkcd.com/31/.)

Don't worry about the cache/disk/tape/FedEx performance in your solutions; just analyze the CPU time as usual. Designing algorithms with few cache misses or page faults is a interesting pastime; simultaneously optimizing CPU time *and* cache misses *and* page faults *and* FedEx faults is a topic of active research. Sadly, this kind of design and analysis requires tools we won't see in this class.

Trees and Subtrees

35. Suppose we need to distribute a message to all the nodes in a rooted tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. Design an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in a given tree.



A message being distributed through a tree in five rounds.

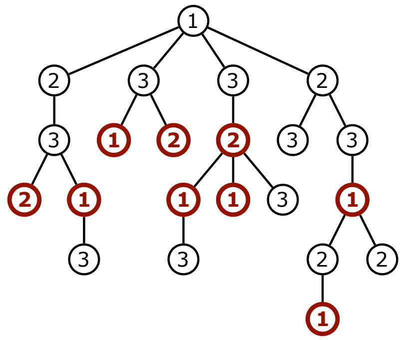
36. Oh, no! You have been appointed as the organizer of Giggle, Inc.'s annual mandatory holiday party! The employees at Giggle are organized into a strict hierarchy, that is, a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how "fun" the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the "fun" ratings of the guests.

37. Since so few people came to last year's holiday party, the president of Giggle, Inc. decides to give each employee a present instead this year. Specifically, each employee must receive on the three gifts: (1) an all-expenses-paid six-week vacation anywhere in the world, (2) an all-the-pancakes-you-can-eat breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. Corporate regulations prohibit any employee from receiving exactly the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy.

    As Giggle, Inc.'s official party czar, it's *your* job to decide which gift each employee receives. Describe an algorithm to distribute gifts so that the minimum number of people are fired. Yes, you may send the president a flaming bag of dog poop.

    More formally, you are given a rooted tree T, representing the company hierarchy, and you want to label each node in T with an integer 1, 2, or 3, so that every node has a different label from its parent. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T.

38. After losing so many employees to last year's Flaming Dog Poop Holiday Debacle, the president of Giggle, Inc. has declared that once again there will be a holiday party this year. Recall that the employees are organized into a strict hierarchy, that is, a tree with the company president at the root. The president demands that you invite exactly k employees,



A tree labeling with cost 9.
Bold nodes have smaller labels than their parents.
This is *not* the optimal labeling for this tree.

including the president herself. Moreover, everyone who is invited is required to attend. Yeah, that'll be fun.

The all-knowing oracles in Human Resources have assigned a real number to each employee indicating the *awkwardness* of inviting both that employee and their immediate supervisor; a negative value indicates that the employee and their supervisor actually like each other. Your goal is to choose a subset k employees to invite, so that the total awkwardness of the resulting party is as small as possible. For example, if the guest list does not include both an employee and their immediate supervisor, the total awkwardness is zero.

(a) Describe an algorithm that computes the total awkwardness of the least awkward subset of k employees, assuming the company hierarchy is described by a *binary* tree. That is, assume that each employee directly supervises at most two others.

*(b) Describe an algorithm that computes the total awkwardness of the least awkward subset of k employees, with no restrictions on the company hierarchy.

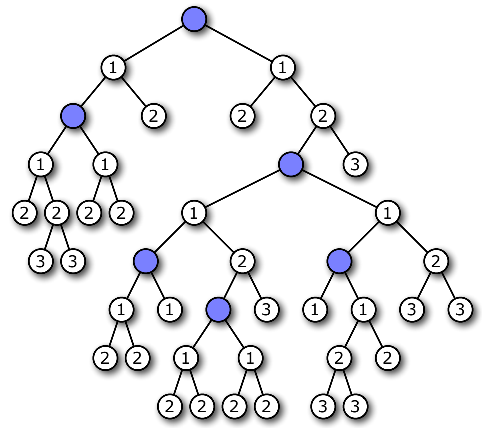
39. Let T be a rooted binary tree with n vertices, and let $k \leq n$ be a positive integer. We would like to mark k vertices in T so that every vertex has a nearby marked ancestor. More formally, we define the *clustering cost* of any subset K of vertices as

$$cost(K) = \max_{v} cost(v, K),$$

where the maximum is taken over all vertices v in the tree, and

$$cost(v, K) = \begin{cases} 0 & \text{if } v \in K \\ \infty & \text{if } v \text{ is the root of } T \text{ and } v \notin K \\ 1 + cost(parent(v)) & \text{otherwise} \end{cases}$$

Describe and analyze a dynamic-programming algorithm to compute the minimum clustering cost of any subset of k vertices in T. For full credit, your algorithm should run in $O(n^2k^2)$ time.



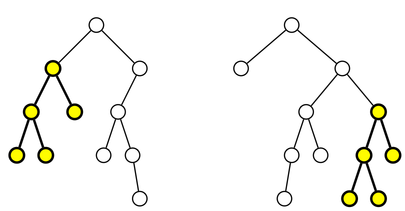
A subset of 5 vertices with clustering cost 3

The next several questions ask for algorithms to find various optimal subtrees in trees. To make the problem statements precise, we must distinguish between several different types of trees and subtrees:

- By default, a *tree* is just a connected, acyclic, undirected graph.
- A *rooted* tree has a distinguished vertex, called the *root*. A tree without a distinguished root vertex is called an *unrooted* tree or a *free* tree.
- In an *ordered* tree, the neighbors of every vertex have a well-defined cyclic order. A tree without these orders is called an *unordered* tree.
- A *binary* tree is a rooted tree in which every node has a (possibly empty) *left* subtree and a (possibly empty) *right* subtree. Two binary trees are isomorphic if they are both empty, or if their left subtrees are isomorphic and their right subtrees are isomorphic.
- A (rooted) subtree of a *rooted* tree consists of a node and all its descendants. A (free) subtree of an *unrooted* tree is any connected subgraph. Subtrees of ordered rooted trees are themselves ordered trees.

40. This question asks you to find efficient algorithms to compute the ***largest common rooted subtree*** of two given rooted trees. A rooted subtree consists of an arbitrary node and *all* its descendants. However, the precise definition of "common" depends on which rooted trees we consider to be isomorphic.

    (a) Describe an algorithm to find the largest common *binary* subtree of two given *binary* trees.

    Two binary trees, with their largest common (rooted) subtree emphasized

    (b) An *ordered* tree is either empty or a node with a *sequence* of children, which are themselves the roots of (possibly empty) ordered trees. Two ordered trees are isomorphic if they are both empty, or if their ith subtrees are isomorphic for all i. Describe an algorithm to find the largest common ordered subtree of two ordered trees T_1 and T_2.

(c) An *unordered* tree is either empty or a node with a *set* of children, which are themselves the roots of (possibly empty) ordered trees. Two unordered trees are isomorphic if they are both empty, or the subtrees or each tree *can be ordered so that* their ith subtrees are isomorphic for all i. Describe an algorithm to find the largest common unordered subtree of two unordered trees T_1 and T_2.

41. This question asks you to find efficient algorithms to compute optimal subtrees in *unrooted* trees. A *subtree* of an unrooted tree is any connected subgraph.

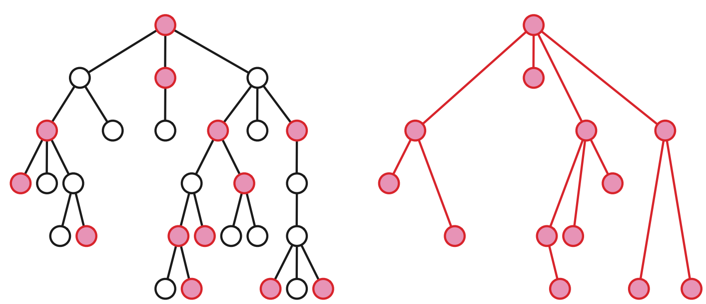
(a) Suppose you are given an unrooted tree T with weights on its *edges*, which may be positive, negative, or zero. Describe an algorithm to find a *path* in T with maximum total weight.

(b) Suppose you are given an unrooted tree T with weights on its vertices, which may be positive, negative, or zero. Describe an algorithm to find a *subtree* of T with maximum total weight.

(c) Let T_1 and T_2 be *ordered* trees, meaning that the neighbors of every node have a well-defined cyclic order. Describe an algorithm to find the largest common *ordered* subtree of T_1 and T_2.

*(d) Let T_1 and T_2 be *unordered* trees. Describe an algorithm to find the largest common *unordered* subtree of T_1 and T_2.

42. ***Sub-branchings*** of a rooted tree are a generalization of subsequences of a sequence. A sub-branching of a tree is a subset S of the nodes such that *exactly* one node in S that does not have a proper ancestor in S. Any sub-branching S implicitly defines a tree $T(S)$, in which the parent of a node $x \in S$ is the closest proper ancestor (in T) of x that is also in S.



A sub-branching S and its associated tree $T(S)$.

(a) Let T be a rooted tree with labeled nodes. We say that T is *boring* if, for each node x, all children of x have the same label; children of different nodes may have different labels. A sub-branching S of a labeled rooted tree T is boring if its associated tree $T(S)$ is boring; nodes in $T(S)$ inherit their labels from T. Describe an algorithm to find the largest boring sub-branching S of a given labeled rooted tree.

(b) Suppose we are given a rooted tree T whose nodes are labeled with numbers. Describe an algorithm to find the largest *heap-ordered sub-branching* of T. That is, your algorithm should return the largest sub-branching S such that every node in $T(S)$ has a smaller label than its children in $T(S)$.

(c) Suppose we are given a *binary* tree T whose nodes are labeled with numbers. Describe an algorithm to find the largest *binary-search-ordered sub-branching* of T. That is, your algorithm should return a sub-branching S such that every node in $T(S)$ has at most two children, and an inorder traversal of $T(S)$ is an increasing subsequence of an inorder traversal of T.

(d) Recall that a rooted tree is *ordered* if the children of each node have a well-defined left-to-right order. Describe an algorithm to find the largest binary-search-ordered sub-branching S of an *arbitrary* ordered tree T whose nodes are labeled with numbers. Again, the order of nodes in $T(S)$ should be consistent with their order in T.

*(e) Describe an algorithm to find the largest common *ordered* sub-branching of two *ordered* labeled rooted trees.

★(f) Describe an algorithm to find the largest common *unordered* sub-branching of two *unordered* labeled rooted trees. *[Hint: This problem will be much easier after you've seen flows.]*

It is a very sad thing that nowadays there is so little useless information.

— Oscar Wilde, "A Few Maxims for the Instruction Of The Over-Educated" (1894)

Ninety percent of science fiction is crud.
But then, ninety percent of everything is crud,
and it's the ten percent that isn't crud that is important.

— [Theodore] Sturgeon's Law (1953)

*6 Advanced Dynamic Programming

Dynamic programming is a powerful technique for efficiently solving recursive problems, but it's hardly the end of the story. In many cases, once we have a basic dynamic programming algorithm in place, we can make further improvements to bring down the running time or the space usage. We saw one example in the Fibonacci number algorithm. Buried inside the naïve iterative Fibonacci algorithm is a recursive problem—computing a power of a matrix—that can be solved more efficiently by dynamic programming techniques—in this case, repeated squaring.

6.1 Saving Space: Divide and Conquer

Just as we did for the Fibonacci recurrence, we can reduce the space complexity of our edit distance algorithm from $O(mn)$ to $O(m+n)$ by only storing the current and previous rows of the memoization table. This 'sliding window' technique provides an easy space improvement for most (but *not* all) dynamic programming algorithm.

Unfortunately, this technique seems to be useful only if we are interested in the *cost* of the optimal edit sequence, not if we want the optimal edit sequence itself. By throwing away most of the table, we apparently lose the ability to walk backward through the table to recover the optimal sequence.

Fortunately for memory-misers, in 1975 Dan Hirshberg discovered a simple divide-and-conquer strategy that allows us to compute the optimal edit sequence in $O(mn)$ time, using just $O(m+n)$ space. The trick is to record not just the edit distance for each pair of prefixes, but also a single position in the middle of the optimal editing sequence for that prefix. Specifically, any optimal editing sequence that transforms $A[1..m]$ into $B[1..n]$ can be split into two smaller editing sequences, one transforming $A[1..m/2]$ into $B[1..h]$ for some integer h, the other transforming $A[m/2+1..m]$ into $B[h+1..n]$.

To compute this breakpoint h, we define a second function $Half(i,j)$ such that some optimal edit sequence from $A[1..i]$ into $B[1..j]$ contains an optimal edit sequence from $A[1..m/2]$ to $B[1..Half(i,j)]$. We can define this function recursively as follows:

$$Half(i,j) = \begin{cases} \infty & \text{if } i < m/2 \\ j & \text{if } i = m/2 \\ Half(i-1,j) & \text{if } i > m/2 \text{ and } Edit(i,j) = Edit(i-1,j)+1 \\ Half(i,j-1) & \text{if } i > m/2 \text{ and } Edit(i,j) = Edit(i,j-1)+1 \\ Half(i-1,j-1) & \text{otherwise} \end{cases}$$

(Because there there may be more than one optimal edit sequence, this is not the only correct definition.) A simple inductive argument implies that $Half(m,n)$ is indeed the correct value of h. We can easily modify our earlier algorithm so that it computes $Half(m,n)$ at the same time as the edit distance $Edit(m,n)$, all in $O(mn)$ time, using only $O(m)$ space.

Edit		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Half		A	L	G	O	R	I	T	H	M
	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
A	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
L	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
T	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
R	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
U	0	1	2	3	4	5	6	7	8	9
I	0	1	2	3	4	5	**5**	5	5	5
S	0	1	2	3	4	5	5	5	5	5
T	0	1	2	3	4	5	5	**5**	5	5
I	0	1	2	3	4	5	**5**	5	5	5
C	0	1	2	3	4	5	5	5	5	5

Finally, to compute the optimal editing sequence that transforms A into B, we recursively compute the optimal sequences transforming $A[1..m/2]$ into $B[1..Half(m,n)]$ and transforming $A[m/2+1..m]$ into $B[Half(m,n)+1..n]$. The recursion bottoms out when one string has only constant length, in which case we can determine the optimal editing sequence in linear time using our old dynamic programming algorithm. The running time of the resulting algorithm satisfies the following recurrence:

$$T(m,n) = \begin{cases} O(n) & \text{if } m \le 1 \\ O(m) & \text{if } n \le 1 \\ O(mn) + T(m/2,h) + T(m/2,n-h) & \text{otherwise} \end{cases}$$

It's easy to prove inductively that $T(m,n) = O(mn)$, no matter what the value of h is. Specifically, the entire algorithm's running time is at most twice the time for the initial dynamic programming phase.

$$\begin{aligned} T(m,n) &\le \alpha mn + T(m/2,h) + T(m/2,n-h) \\ &\le \alpha mn + 2\alpha mh/2 + 2\alpha m(n-h)/2 && \text{[inductive hypothesis]} \\ &= 2\alpha mn \end{aligned}$$

A similar inductive argument implies that the algorithm uses only $O(n+m)$ space.

Hirschberg's divide-and-conquer trick can be applied to almost any dynamic programming problem to obtain an algorithm to construct an optimal *structure* (in this case, the cheapest edit sequence) within the same space and time bounds as computing the *cost* of that optimal structure (in this case, edit distance). For this reason, we will almost always ask you for algorithms to compute the cost of some optimal structure, not the optimal structure itself.

6.2 Saving Time: Sparseness

In many applications of dynamic programming, we are faced with instances where almost every recursive subproblem will be resolved exactly the same way. We call such instances *sparse*. For example, we might want to compute the edit distance between two strings that have few characters in common, which means there are few "free" substitutions anywhere in the table.

Most of the table has exactly the same structure. If we can reconstruct the entire table from just a few key entries, then why compute the entire table?

To better illustrate how to exploit sparseness, let's consider a simplification of the edit distance problem, in which substitutions are not allowed (or equivalently, where a substitution counts as two operations instead of one). Now our goal is to maximize the number of "free" substitutions, or equivalently, to find the *longest common subsequence* of the two input strings.

Fix the two input strings $A[1\,..\,n]$ and $B[1\,..\,m]$. For any indices i and j, let $LCS(i, j)$ denote the length of the longest common subsequence of the prefixes $A[1\,..\,i]$ and $B[1\,..\,j]$. This function can be defined recursively as follows:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ LCS(i-1, j-1) + 1 & \text{if } A[i] = B[j] \\ \max\{LCS(i, j-1),\ LCS(i-1, j)\} & \text{otherwise} \end{cases}$$

This recursive definition directly translates into an $O(mn)$-time dynamic programming algorithm.

Call an index pair (i, j) a *match point* if $A[i] = B[j]$. In some sense, match points are the only 'interesting' locations in the memoization table; given a list of the match points, we could easily reconstruct the entire table.

	«	A	L	G	O	R	I	T	H	M	S	»
«	**0**	0	0	0	0	0	0	0	0	0	0	0
A	0	**1**	1	1	1	1	1	1	1	1	1	1
L	0	1	**2**	2	2	2	2	2	2	2	2	2
T	0	1	2	2	2	2	2	**3**	3	3	3	3
R	0	1	2	2	2	**3**	3	3	3	3	3	3
U	0	1	2	2	2	3	3	3	3	3	3	3
I	0	1	2	2	2	3	**4**	4	4	4	4	4
S	0	1	2	2	2	3	4	4	4	4	**5**	5
T	0	1	2	2	2	3	4	**5**	5	5	5	5
I	0	1	2	2	2	3	**4**	5	5	5	5	5
C	0	1	2	2	2	3	4	5	5	5	5	5
»	0	1	2	2	2	3	4	5	5	5	5	**6**

The *LCS* memoization table for `ALGORITHMS` and `ALTRUISTIC`; the brackets « and » are sentinel characters.

More importantly, we can compute the *LCS* function directly from the list of match points using the following recurrence:

$$LCS(i,j) = \begin{cases} 0 & \text{if } i = j = 0 \\ \max\left\{LCS(i',j') \mid A[i'] = B[j'] \text{ and } i' < i \text{ and } j' < j\right\} + 1 & \text{if } A[i] = B[j] \\ \max\left\{LCS(i',j') \mid A[i'] = B[j'] \text{ and } i' \le i \text{ and } j' \le j\right\} & \text{otherwise} \end{cases}$$

(Notice that the inequalities are strict in the second case, but not in the third.) To simplify boundary issues, we add unique sentinel characters $A[0] = B[0]$ and $A[m+1] = B[n+1]$ to both strings. This ensures that the sets on the right side of the recurrence equation are non-empty, and that we only have to consider match points to compute $LCS(m, n) = LCS(m+1, n+1) - 1$.

If there are K match points, we can actually compute them all in $O(m \log m + n \log n + K)$ time. Sort the characters in each input string, but remembering the original index of each character, and then essentially merge the two sorted arrays, as follows:

```
FindMatches(A[1..m], B[1..n]):
  for i ← 1 to m:  I[i] ← i
  for j ← 1 to n:  J[j] ← j

  sort A and permute I to match
  sort B and permute J to match

  i ← 1;  j ← 1
  while i < m and j < n
      if A[i] < B[j]
          i ← i + 1
      else if A[i] > B[j]
          j ← j + 1
      else                        ⟨⟨Found a match!⟩⟩
          ii ← i
          while A[ii] = A[i]
              jj ← j
              while B[jj] = B[j]
                  report (I[ii], J[jj])
                  jj ← jj + 1
              ii ← i + 1
          i ← ii;  j ← jj
```

To efficiently evaluate our modified recurrence, we once again turn to dynamic programming. We consider the match points in lexicographic order—the order they would be encountered in a standard row-major traversal of the $m \times n$ table—so that when we need to evaluate $LCS(i, j)$, all match points (i', j') with $i' < i$ and $j' < j$ have already been evaluated.

```
SparseLCS(A[1..m], B[1..n]):
  Match[1..K] ← FindMatches(A, B)
  Match[K + 1] ← (m + 1, n + 1)       ⟨⟨Add end sentinel⟩⟩
  Sort M lexicographically
  for k ← 1 to K
      (i, j) ← Match[k]
      LCS[k] ← 1                      ⟨⟨From start sentinel⟩⟩
      for ℓ ← 1 to k − 1
          (i', j') ← Match[ℓ]
          if i' < i and j' < j
              LCS[k] ← min{LCS[k], 1 + LCS[ℓ]}
  return LCS[K + 1] − 1
```

The overall running time of this algorithm is $O(m \log m + n \log n + K^2)$. So as long as $K = o(\sqrt{mn})$, this algorithm is actually faster than naïve dynamic programming.

6.3 Saving Time: Monotonicity

> The SMAWK matrix-searching algorithm is a better example here; the problem is more general, the algorithm is simpler, and the proof is self-contained. Next time!

Recall the optimal binary search tree problem from the previous lecture. Given an array $F[1..n]$ of access frequencies for n items, the problem it to compute the binary search tree that minimizes the cost of all accesses. A relatively straightforward dynamic programming algorithm solves this problem in $O(n^3)$ time.

As for longest common subsequence problem, the algorithm can be improved by exploiting some structure in the memoization table. In this case, however, the relevant structure isn't in the table of costs, but rather in the table used to reconstruct the actual optimal tree. Let $OptRoot[i, j]$ denote the index of the root of the optimal search tree for the frequencies $F[i\,..\,j]$; this is always an integer between i and j. Donald Knuth proved the following nice monotonicity property for optimal subtrees: If we move either end of the subarray, the optimal root moves in the same direction or not at all. More formally:

$$\boxed{OptRoot[i, j-1] \le OptRoot[i, j] \le OptRoot[i+1, j] \text{ for all } i \text{ and } j.}$$

This (nontrivial!) observation leads to the following more efficient algorithm:

```
FASTEROPTIMALSEARCHTREE(f[1..n]):
  INITF(f[1..n])
  for i ← 1 downto n
      OptCost[i, i−1] ← 0
      OptRoot[i, i−1] ← i
  for d ← 0 to n
      for i ← 1 to n
          COMPUTECOSTANDROOT(i, i+d)
  return OptCost[1, n]
```

```
COMPUTECOSTANDROOT(i, j):
  OptCost[i, j] ← ∞

  for r ← OptRoot[i, j−1] to OptRoot[i+1, j]
      tmp ← OptCost[i, r−1] + OptCost[r+1, j]
      if OptCost[i, j] > tmp
          OptCost[i, j] ← tmp
          OptRoot[i, j] ← r

  OptCost[i, j] ← OptCost[i, j] + F[i, j]
```

It's not hard to see that the loop index r increases monotonically from 1 to n during each iteration of the *outermost* for loop of FASTEROPTIMALSEARCHTREE. Consequently, the total cost of all calls to COMPUTECOSTANDROOT is only $O(n^2)$.

If we formulate the problem slightly differently, this algorithm can be improved even further. Suppose we require the optimum *external* binary tree, where the keys $A[1\,..\,n]$ are all stored at the leaves, and intermediate pivot values are stored at the internal nodes. An algorithm discovered by Ching Hu and Alan Tucker[1] computes the optimal binary search tree in this setting in only $O(n \log n)$ time!

6.4 Saving Time: Four Russians

Some day.

Exercises

1. Describe an algorithm to compute the edit distance between two strings $A[1\,..\,m]$ and $B[1 \ldots n]$ in $O(m \log m + n \log n + K^2)$ time, where K is the number of match points. *[Hint: Use the FINDMATCHES algorithm on page 3 as a subroutine.]*

2. (a) Describe an algorithm to compute the *longest increasing subsequence* of a string $X[1\,..\,n]$ in $O(n \log n)$ time.

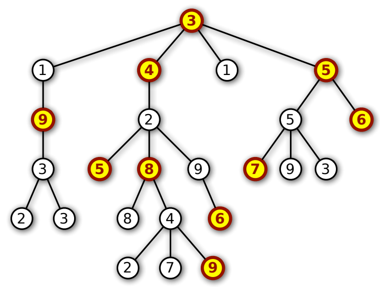
   (b) Using your solution to part (a) as a subroutine, describe an algorithm to compute the longest common subsequence of two strings $A[1\,..\,m]$ and $B[1 \ldots n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points.

[1] T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM J. Applied Math.* 21:514–532, 1971. For a slightly simpler algorithm with the same running time, see A. M. Garsia and M. L. Wachs, A new algorithms for minimal binary search trees, *SIAM J. Comput.* 6:622–642, 1977. The original correctness proofs for both algorithms are rather intricate; for simpler proofs, see Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter, Correctness of constructing optimal alphabetic trees revisited, *Theoretical Computer Science*, 180:309-324, 1997.

3. Describe an algorithm to compute the edit distance between two strings $A[1..m]$ and $B[1...n]$ in $O(m \log m + n \log n + K \log K)$ time, where K is the number of match points. *[Hint: Combine your answers for problems 1 and 2(b).]*

4. Let T be an arbitrary rooted tree, where each vertex is labeled with a positive integer. A subset S of the nodes of T is *heap-ordered* if it satisfies two properties:

   - S contains a node that is an ancestor of every other node in S.
   - For any node v in S, the label of v is larger than the labels of any ancestor of v in S.



A heap-ordered subset of nodes in a tree.

   (a) Describe an algorithm to find the largest heap-ordered subset S of nodes in T that has the heap property in $O(n^2)$ time.

   (b) Modify your algorithm from part (a) so that it runs in $O(n \log n)$ time when T is a linked list. *[Hint: This special case is equivalent to a problem you've seen before.]*

   *(c) Describe an algorithm to find the largest subset S of nodes in T that has the heap property, in $O(n \log n)$ time. *[Hint: Find an algorithm to merge two sorted lists of lengths k and ℓ in $O(\log \binom{k+\ell}{k})$ time.]*

The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed in all its forms, greed for life, money, love, knowledge has marked the upward surge in mankind. And greed—mark my words—will save not only Teldar Paper but the other malfunctioning corporation called the USA.

— Gordon Gekko [Michael Douglas], *Wall Street* (1987)

There is always an easy solution to every human problem—
neat, plausible, and wrong.

— H. L. Mencken, "The Divine Afflatus",
New York Evening Mail (November 16, 1917)

7 Greedy Algorithms

7.1 Storing Files on Tape

Suppose we have a set of n files that we want to store on a tape. In the future, users will want to read those files from the tape. Reading a file from tape isn't like reading a file from disk; first we have to fast-forward past all the other files, and that takes a significant amount of time. Let $L[1..n]$ be an array listing the lengths of each file; specifically, file i has length $L[i]$. If the files are stored in order from 1 to n, then the cost of accessing the kth file is

$$cost(k) = \sum_{i=1}^{k} L[i].$$

The cost reflects the fact that before we read file k we must first scan past all the earlier files on the tape. If we assume for the moment that each file is equally likely to be accessed, then the *expected* cost of searching for a random file is

$$\mathrm{E}[cost] = \sum_{k=1}^{n} \frac{cost(k)}{n} = \sum_{k=1}^{n} \sum_{i=1}^{k} \frac{L[i]}{n}.$$

If we change the order of the files on the tape, we change the cost of accessing the files; some files become more expensive to read, but others become cheaper. Different file orders are likely to result in different expected costs. Specifically, let $\pi(i)$ denote the index of the file stored at position i on the tape. Then the expected cost of the permutation π is

$$\mathrm{E}[cost(\pi)] = \sum_{k=1}^{n} \sum_{i=1}^{k} \frac{L[\pi(i)]}{n}.$$

Which order should we use if we want the expected cost to be as small as possible? The answer is intuitively clear; we should store the files in order from shortest to longest. So let's prove this.

Lemma 1. $\mathrm{E}[cost(\pi)]$ *is minimized when* $L[\pi(i)] \le L[\pi(i+1)]$ *for all* i.

Proof: Suppose $L[\pi(i)] > L[\pi(i+1)]$ for some i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the expected cost by $(L[b]-L[a])/n$. But this change is an improvement, because $L[b] < L[a]$. Thus, if the files are out of order, we can improve the expected cost by swapping some mis-ordered adjacent pair. □

This example gives us our first *greedy algorithm*. To minimize the *total* expected cost of accessing the files, we put the file that is cheapest to access first, and then recursively write everything else; no backtracking, no dynamic programming, just make the best local choice and blindly plow ahead. If we use an efficient sorting algorithm, the running time is clearly $O(n \log n)$, plus the time required to actually write the files. To prove the greedy algorithm is actually correct, we simply prove that the output of any other algorithm can be improved by some sort of swap.

Let's generalize this idea further. Suppose we are also given an array $F[1..n]$ of *access frequencies* for each file; file i will be accessed exactly $F[i]$ times over the lifetime of the tape. Now the *total* cost of accessing all the files on the tape is

$$\Sigma cost(\pi) = \sum_{k=1}^{n}\left(F[\pi(k)] \cdot \sum_{i=1}^{k} L[\pi(i)]\right) = \sum_{k=1}^{n}\sum_{i=1}^{k}\big(F[\pi(k)] \cdot L[\pi(i)]\big).$$

Now what order should store the files if we want to minimize the total cost?

We've already proved that if all the frequencies are equal, then we should sort the files by increasing size. If the frequencies are all different but the file lengths $L[i]$ are all equal, then intuitively, we should sort the files by *decreasing* access frequency, with the most-accessed file first. In fact, this is not hard to prove by modifying the proof of Lemma 1. But what if the sizes and the frequencies are both different? In this case, we should sort the files by the ratio L/F.

Lemma 2. *$\Sigma cost(\pi)$ is minimized when* $\dfrac{L[\pi(i)]}{F[\pi(i)]} \le \dfrac{L[\pi(i+1)]}{F[\pi(i+1)]}$ *for all i.*

Proof: Suppose $L[\pi(i)]/F[\pi(i)] > L[\pi(i+1)]/F[\pi(i+i)]$ for some i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the total cost by $L[b]F[a] - L[a]F[b]$. But this change is an improvement, since

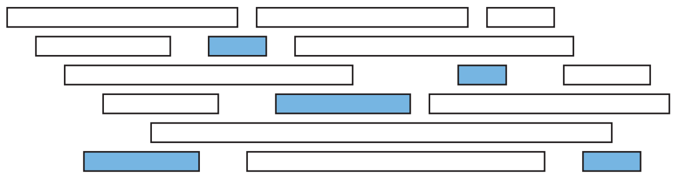
$$\frac{L[a]}{F[a]} > \frac{L[b]}{F[b]} \implies L[b]F[a] - L[a]F[b] < 0.$$

Thus, if two adjacent files are out of order, we can improve the total cost by swapping them. □

7.2 Scheduling Classes

The next example is slightly less trivial. Suppose you decide to drop out of computer science at the last minute and change your major to Applied Chaos. The Applied Chaos department offers all of its classes on the same day every week, called 'Soberday' by the students (but interestingly, *not* by the faculty). Every class has a different start time and a different ending time: AC 101 ('Toilet Paper Landscape Architecture') starts at 10:27pm and ends at 11:51pm; AC 666 ('Immanentizing the Eschaton') starts at 4:18pm and ends at 7:06pm, and so on. In the interest of graduating as quickly as possible, you want to register for as many classes as you can. (Applied Chaos classes don't require any actual *work*.) The university's registration computer won't let you register for overlapping classes, and no one in the department knows how to override this 'feature'. Which classes should you take?

More formally, suppose you are given two arrays $S[1..n]$ and $F[1..n]$ listing the start and finish times of each class; to be concrete, we can assume that $0 \leq S[i] < F[i] \leq M$ for each i, for some value M (for example, the number of picoseconds in Soberday). Your task is to choose the largest possible subset $X \in \{1, 2, \ldots, n\}$ so that for any pair $i, j \in X$, either $S[i] > F[j]$ or $S[j] > F[i]$. We can illustrate the problem by drawing each class as a rectangle whose left and right x-coordinates show the start and finish times. The goal is to find a largest subset of rectangles that do not overlap vertically.



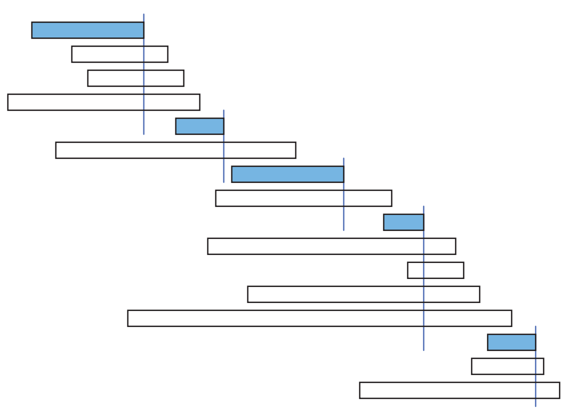
A maximal conflict-free schedule for a set of classes.

This problem has a fairly simple recursive solution, based on the observation that either you take class 1 or you don't. Let B_4 denote the set of classes that end *before* class 1 starts, and let L_8 denote the set of classes that start *late*r than class 1 ends:

$$B_4 = \{i \mid 2 \leq i \leq n \text{ and } F[i] < S[1]\} \qquad L_8 = \{i \mid 2 \leq i \leq n \text{ and } S[i] > F[1]\}$$

If class 1 is in the optimal schedule, then so are the optimal schedules for B_4 and L_8, which we can find recursively. If not, we can find the optimal schedule for $\{2, 3, \ldots, n\}$ recursively. So we should try both choices and take whichever one gives the better schedule. Evaluating this recursive algorithm from the bottom up gives us a dynamic programming algorithm that runs in $O(n^2)$ time. I won't bother to go through the details, because we can do better.[1]

Intuitively, we'd like the first class to finish as early as possible, because that leaves us with the most remaining classes. If this greedy strategy works, it suggests the following very simple algorithm. Scan through the classes in order of finish time; whenever you encounter a class that doesn't conflict with your latest class so far, take it!



The same classes sorted by finish times and the greedy schedule.

We can write the greedy algorithm somewhat more formally as follows. (Hopefully the first line is understandable.) The algorithm clearly runs in $O(n \log n)$ time.

[1]But you should still work out the details yourself. The dynamic programming algorithm can be used to find the "best" schedule for several different definitions of "best", but the greedy algorithm I'm about to describe only works when "best" means "biggest". Also, you need the practice.

```
GREEDYSCHEDULE(S[1..n], F[1..n]):
  sort F and permute S to match
  count ← 1
  X[count] ← 1
  for i ← 2 to n
      if S[i] > F[X[count]]
          count ← count + 1
          X[count] ← i
  return X[1..count]
```

To prove that this algorithm actually gives us a maximal conflict-free schedule, we use an exchange argument, similar to the one we used for tape sorting. We are not claiming that the greedy schedule is the *only* maximal schedule; there could be others. (See the figures on the previous page.) All we can claim is that at least one of the maximal schedules is the one that the greedy algorithm produces.

Lemma 3. *At least one maximal conflict-free schedule includes the class that finishes first.*

Proof: Let f be the class that finishes first. Suppose we have a maximal conflict-free schedule X that does not include f. Let g be the first class in X to finish. Since f finishes before g does, f cannot conflict with any class in the set $S \setminus \{g\}$. Thus, the schedule $X' = X \cup \{f\} \setminus \{g\}$ is also conflict-free. Since X' has the same size as X, it is also maximal. □

To finish the proof, we call on our old friend, induction.

Theorem 4. *The greedy schedule is an optimal schedule.*

Proof: Let f be the class that finishes first, and let L be the subset of classes the start after f finishes. The previous lemma implies that some optimal schedule contains f, so the best schedule that contains f is an optimal schedule. The best schedule that includes f must contain an optimal schedule for the classes that do not conflict with f, that is, an optimal schedule for L. The greedy algorithm chooses f and then, by the inductive hypothesis, computes an optimal schedule of classes from L. □

The proof might be easier to understand if we unroll the induction slightly.

Proof: Let $\langle g_1, g_2, \ldots, g_k \rangle$ be the sequence of classes chosen by the greedy algorithm. Suppose we have a maximal conflict-free schedule of the form

$$\langle g_1, g_2, \ldots, g_{j-1}, c_j, c_{j+1}, \ldots, c_m \rangle,$$

where class c_j is different from the class g_j that would be chosen by the greedy algorithm. (We may have $j = 1$, in which case this schedule starts with a non-greedy choice c_1.) By construction, the jth greedy choice g_j does not conflict with any earlier class $g_1, g_2, \ldots, g_{j-1}$, and since our schedule is conflict-free, neither does c_j. Moreover, g_j has the *earliest* finish time among all classes that don't conflict with the earlier classes; in particular, g_j finishes before c_j. This implies that g_j does not conflict with any of the later classes $c_{j+1}, \ldots, c_m$. Thus, the schedule

$$\langle g_1, g_2, \ldots, g_{j-1}, g_j, c_{j+1}, \ldots, c_m \rangle,$$

is conflict-free. (This is just a generalization of Lemma 3, which considers the case $j = 1$.)

By induction, it now follows that there is an optimal schedule $\langle g_1, g_2, \ldots, g_k, c_{k+1}, \ldots, c_m \rangle$ that includes *every* class chosen by the greedy algorithm. But this is impossible unless $k = m$; if there were a class c_{k+1} that does not conflict with g_k, the greedy algorithm would choose more than k classes. □

7.3 General Structure

The basic structure of this correctness proof is exactly the same as for the tape-sorting problem: an inductive exchange argument.

- Assume that there is an optimal solution that is different from the greedy solution.
- Find the "first" difference between the two solutions.
- Argue that we can exchange the optimal choice for the greedy choice without degrading the solution.

This argument implies by induction that some optimal solution that *contains* the entire greedy solution, and therefore *equals* the greedy solution. Sometimes, as in the scheduling problem, an additional step is required to show no optimal solution *strictly* improves the greedy solution.

7.4 Huffman Codes

A *binary code* assigns a string of 0s and 1s to each character in the alphabet. A binary code is *prefix-free* if no code is a prefix of any other. 7-bit ASCII and Unicode's UTF-8 are both prefix-free binary codes. Morse code is a binary code, but it is not prefix-free; for example, the code for S ($\cdot\cdot\cdot$) includes the code for E ($\cdot$) as a prefix. Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf; 0 for left, 1 for right. The length of a codeword for a symbol is the depth of the corresponding leaf.

Let me emphasize that binary code trees are *not* binary search trees; we don't care at all about the order of symbols at the leaves.

Suppose we want to encode messages in an n-character alphabet so that the encoded message is as short as possible. Specifically, given an array frequency counts $f[1..n]$, we want to compute a prefix-free binary code that minimizes the total encoded length of the message:[2]

$$\sum_{i=1}^{n} f[i] \cdot depth(i).$$

In 1951, as a PhD student at MIT, David Huffman developed the following greedy algorithm to produce such an optimal code:[3]

$\boxed{\text{H\textsc{uffman}: Merge the two least frequent letters and recurse.}}$

For example, suppose we want to encode the following helpfully self-descriptive sentence, discovered by Lee Sallows:[4]

---

[2]This looks almost exactly like the cost of a binary search tree, but the optimization problem is very different: code trees are not required to keep the keys in any particular order.

[3]Huffman was a student in an information theory class taught by Robert Fano, who was a close colleague of Claude Shannon, the father of information theory. Fano and Shannon had previously developed a different greedy algorithm for producing prefix codes—split the frequency array into two subarrays as evenly as possible, and then recursively build a code for each subarray—but these Fano-Shannon codes were known not to be optimal. Fano posed the (then open) problem of finding an optimal encoding to his class; Huffman solved the problem as a class project, in lieu of taking a final exam.

[4]A. K. Dewdney. Computer recreations. *Scientific American*, October 1984. Douglas Hofstadter published a few earlier examples of Lee Sallows' self-descriptive sentences in his *Scientific American* column in January 1982.

> This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

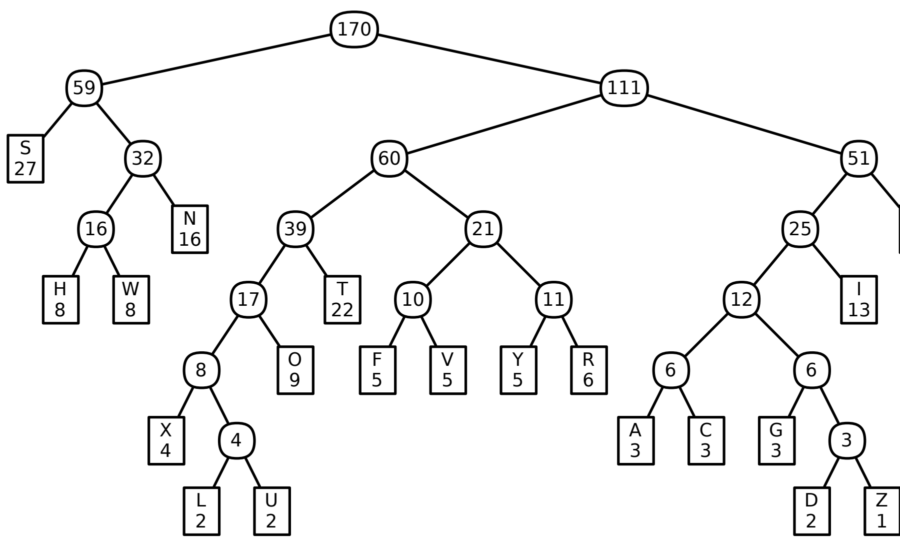
To keep things simple, let's forget about the forty-four spaces, nineteen apostrophes, nineteen commas, three hyphens, and only one period, and just encode the letters. Here's the frequency table:

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

Huffman's algorithm picks out the two least frequent letters, breaking ties arbitrarily—in this case, say, Z and D—and merges them together into a single new character DZ with frequency 3. This new character becomes an internal node in the code tree we are constructing, with Z and D as its children; it doesn't matter which child is which. The algorithm then recursively constructs a Huffman code for the new frequency table

A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	DZ
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

After 19 merges, all 20 characters have been merged together. The record of merges gives us our code tree. The algorithm makes a number of arbitrary choices; as a result, there are actually several different Huffman codes. One such code is shown below. For example, the code for A is 110000, and the code for S is 00.



A Huffman code for Lee Sallows' self-descriptive sentence; the numbers are frequencies for merged characters

If we use this code, the encoded message starts like this:

1001	0100	1101	00	00	111	011	1001	111	011	110001	111	110001	10001	011	1001	110000	...
T	H	I	S	S	E	N	T	E	N	C	E	C	O	N	T	A	

Here is the list of costs for encoding each character in the example message, along with that character's contribution to the total length of the encoded message:

char.	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
freq.	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
depth	6	6	7	3	5	6	4	4	7	3	4	4	2	4	7	5	4	6	5	7
total	18	18	14	78	25	18	32	52	14	48	36	24	54	88	14	25	32	24	25	7

Altogether, the encoded message is 646 bits long. Different Huffman codes would assign different codes, possibly with different lengths, to various characters, but the overall length of the encoded message is the same for any Huffman code: 646 bits.

Given the simple structure of Huffman's algorithm, it's rather surprising that it produces an *optimal* prefix-free binary code. Encoding Lee Sallows' sentence using *any* prefix-free code requires at least 646 bits! Fortunately, the recursive structure makes this claim easy to prove using an exchange argument, similar to our earlier optimality proofs. We start by proving that the algorithm's very first choice is correct.

Lemma 5. *Let x and y be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which x and y are siblings.*

Proof: I'll actually prove a stronger statement: There is an optimal code in which x and y are siblings *and* have the largest depth of any leaf.

Let T be an optimal code tree, and suppose this tree has depth d. Since T is a full binary tree, it has at least two leaves at depth d that are siblings. (Verify this by induction!) Suppose those two leaves are *not* x and y, but some other characters a and b.

Let T' be the code tree obtained by swapping x and a. The depth of x increases by some amount Δ, and the depth of a decreases by the same amount. Thus,

$$cost(T') = cost(T) - (f[a] - f[x])\Delta.$$

By assumption, x is one of the two least frequent characters, but a is not, which implies that $f[a] \geq f[x]$. Thus, swapping x and a does not increase the total cost of the code. Since T was an optimal code tree, swapping x and a does not decrease the cost, either. Thus, T' is also an optimal code tree (and incidentally, $f[a]$ actually equals $f[x]$).

Similarly, swapping y and b must give yet another optimal code tree. In this final optimal code tree, x and y are maximum-depth siblings, as required. □

Now optimality is guaranteed by our dear friend the Recursion Fairy! Essentially we're relying on the following recursive definition for a full binary tree: either a single node, or a full binary tree where some leaf has been replaced by an internal node with two leaf children.

Theorem 6. *Huffman codes are optimal prefix-free binary codes.*

Proof: If the message has only one or two different characters, the theorem is trivial.

Otherwise, let $f[1..n]$ be the original input frequencies, where without loss of generality, $f[1]$ and $f[2]$ are the two smallest. To keep things simple, let $f[n+1] = f[1] + f[2]$. By the previous lemma, we know that some optimal code for $f[1..n]$ has characters 1 and 2 as siblings.

Let T' be the Huffman code tree for $f[3..n+1]$; the inductive hypothesis implies that T' is an optimal code tree for the smaller set of frequencies. To obtain the final code tree T, we replace the leaf labeled $n+1$ with an internal node with two children, labelled 1 and 2. I claim that T is optimal for the original frequency array $f[1..n]$.

To prove this claim, we can express the cost of T in terms of the cost of T' as follows. (In these equations, $depth(i)$ denotes the depth of the leaf labelled i in either T or T'; if the leaf appears in both T and T', it has the same depth in both trees.)

$$\begin{aligned}
cost(T) &= \sum_{i=1}^{n} f[i] \cdot depth(i) \\
&= \sum_{i=3}^{n+1} f[i] \cdot depth(i) + f[1] \cdot depth(1) + f[2] \cdot depth(2) - f[n+1] \cdot depth(n+1) \\
&= cost(T') + f[1] \cdot depth(1) + f[2] \cdot depth(2) - f[n+1] \cdot depth(n+1) \\
&= cost(T') + (f[1] + f[2]) \cdot depth(T) - f[n+1] \cdot (depth(T) - 1) \\
&= cost(T') + f[1] + f[2]
\end{aligned}$$

This equation implies that minimizing the cost of T is equivalent to minimizing the cost of T'; in particular, attaching leaves labeled 1 and 2 to the leaf in T' labeled $n+1$ gives an optimal code tree for the original frequencies. □

To actually implement Huffman codes efficiently, we keep the characters in a min-heap, where the priority of each character is its frequency. We can construct the code tree by keeping three arrays of indices, listing the left and right children and the parent of each node. The root of the tree is the node with index $2n-1$.

```
BuildHuffman(f[1..n]):
  for i ← 1 to n
      L[i] ← 0; R[i] ← 0
      Insert(i, f[i])
  for i ← n to 2n − 1
      x ← ExtractMin()
      y ← ExtractMin()
      f[i] ← f[x] + f[y]
      L[i] ← x; R[i] ← y
      P[x] ← i; P[y] ← i
      Insert(i, f[i])
  P[2n − 1] ← 0
```

The algorithm performs $O(n)$ min-heap operations. If we use a balanced binary tree as the heap, each operation requires $O(\log n)$ time, so the total running time of BuildHuffman is $O(n \log n)$.

Finally, here are simple algorithms to encode and decode messages:

```
HuffmanEncode(A[1..k]):
  m ← 1
  for i ← 1 to k
      HuffmanEncodeOne(A[i])

HuffmanEncodeOne(x):
  if x < 2n − 1
      HuffmanEncodeOne(P[x])
      if x = L[P[x]]
          B[m] ← 0
      else
          B[m] ← 1
      m ← m + 1
```

```
HuffmanDecode(B[1..m]):
  k ← 1
  v ← 2n − 1
  for i ← 1 to m
      if B[i] = 0
          v ← L[v]
      else
          v ← R[v]
      if L[v] = 0
          A[k] ← v
          k ← k + 1
          v ← 2n − 1
```

Exercises

1. For each of the following alternative greedy algorithms for the class scheduling problem, either prove that the algorithm always constructs an optimal schedule, or describe a small input example for which the algorithm does not produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control).

   (a) Choose the course x that *ends last*, discard classes that conflict with x, and recurse.

   (b) Choose the course x that *starts first*, discard all classes that conflict with x, and recurse.

   (c) Choose the course x that *starts last*, discard all classes that conflict with x, and recurse.

   (d) Choose the course x with *shortest duration*, discard all classes that conflict with x, and recurse.

   (e) Choose a course x that *conflicts with the fewest other courses*, discard all classes that conflict with x, and recurse.

   (f) If no classes conflict, choose them all. Otherwise, discard the course with *longest duration* and recurse.

   (g) If no classes conflict, choose them all. Otherwise, discard a course that *conflicts with the most other courses* and recurse.

   (h) Let x be the class with the *earliest start time*, and let y be the class with the *second earliest start time*.
   - If x and y are disjoint, choose x and recurse on everything but x.
   - If x completely contains y, discard x and recurse.
   - Otherwise, discard y and recurse.

   (i) If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that *ends last*, discard all classes that conflict with y, and recurse.

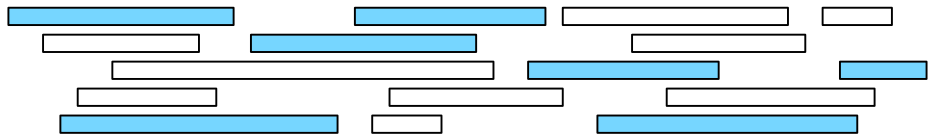
2. Now consider a weighted version of the class scheduling problem, where different classes offer different number of credit hours (totally unrelated to the duration of the class lectures). Your goal is now to choose a set of non-conflicting classes that give you the largest possible number of credit hours, given an array of start times, end times, and credit hours as input.

   (a) Prove that the greedy algorithm described in the notes — Choose the class that ends first and recurse — does *not* always return an optimal schedule.

   (b) Describe an algorithm to compute the optimal schedule in $O(n^2)$ time.

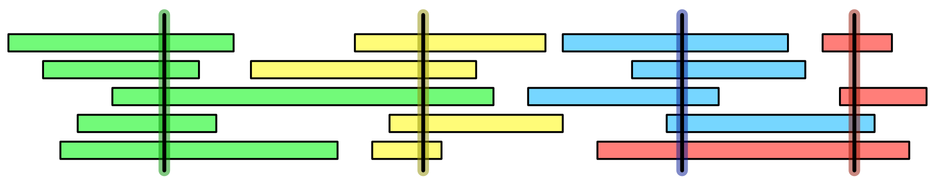
3. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The *size* of a tiling cover is just the number of intervals.

   Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X. If you use a greedy algorithm, you must prove that it is correct.



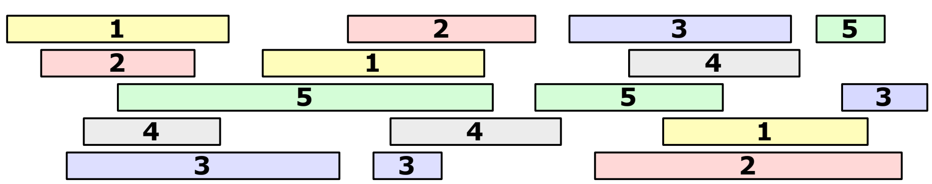
A set of intervals. The seven shaded intervals form a tiling path.

4. Let X be a set of n intervals on the real line. We say that a set P of points *stabs* X if every interval in X contains at least one point in P. Describe and analyze an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $X_L[1\,..\,n]$ and $X_R[1\,..\,n]$, representing the left and right endpoints of the intervals in X. As usual, If you use a greedy algorithm, you must prove that it is correct.



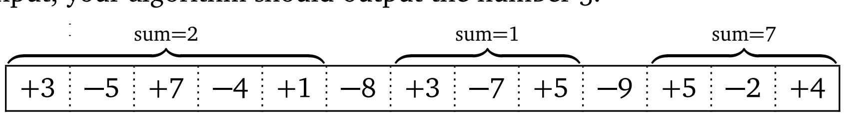
A set of intervals stabbed by four points (shown here as vertical segments)

5. Let X be a set of n intervals on the real line. A *proper coloring* of X assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Describe and analyze an efficient algorithm to compute the minimum number of colors needed to properly color X. Assume that your input consists of two arrays $L[1\,..\,n]$ and $R[1\,..\,n]$, where $L[i]$ and $R[i]$ are the left and right endpoints of the ith interval. As usual, if you use a greedy algorithm, you must prove that it is correct.



A proper coloring of a set of intervals using five colors.

6. Suppose you are given an array $A[1\,..\,n]$ of integers, each of which may be positive, negative, or zero. A contiguous subarray $A[i\,..\,j]$ is called a ***positive interval*** if the sum of its entries is greater than zero. Describe and analyze an algorithm to compute the minimum number of positive intervals that cover every positive entry in A. For example, given the following array as input, your algorithm should output the number 3.



7. Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \cdots < c[n]$. (In the U.S., for example, $n = 6$ and the values

are 1, 5, 10, 25, 50 and 100 cents.) Your beloved and benevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.

(a) In the United States, there is a simple greedy algorithm that always results in the smallest number of coins: subtract the largest coin and recursively give change for the remainder. El Generalissimo does not approve of American capitalist greed. Show that there is a set of coin values for which the greedy algorithm does *not* always give the smallest possible of coins.

(b) Now suppose El Generalissimo decides to impose a currency system where the coin denominations are consecutive powers $b^0, b^1, b^2, \ldots, b^k$ of some integer $b \geq 2$. Prove that despite El Generalissimo's disapproval, the greedy algorithm described in part (a) does make optimal change in this currency system.

(c) Describe and analyze an efficient algorithm to determine, given a target amount A and a sorted array $c[1..n]$ of coin denominations, the smallest number of coins needed to make A cents in change. Assume that $c[1] = 1$, so that it is possible to make change for any amount A.

8. Suppose you have just purchased a new type of hybrid car that uses fuel extremely efficiently, but can only travel 100 miles on a single battery. The car's fuel is stored in a single-use battery, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and can only be installed by licensed battery-replacement technicians. Thus, even if you decide to replace your battery early, you must still pay full price for the new battery to be installed. Moreover, because these batteries are in high demand, no one can afford to own more than one battery at a time.

Suppose you are trying to get from San Francisco to New York City on the new Inter-Continental Super-Highway, which runs in a direct line between these two cities. There are several fueling stations along the way; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every fueling station on the ICSH. Given this information, how do you decide the best places to stop for fuel?

More formally, suppose you are given two arrays $D[1..n]$ and $C[1..n]$, where $D[i]$ is the distance from the start of the highway to the ith station, and $C[i]$ is the cost to replace your battery at the ith station. Assume that your trip starts and ends at fueling stations (so $D[1] = 0$ and $D[n]$ is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

(a) Describe and analyze a greedy algorithm to find the minimum number of refueling stops needed to complete your trip. Don't forget to prove that your algorithm is correct.

(b) But what you really want to minimize is the total *cost* of travel. Show that your greedy algorithm in part (a) does *not* produce an optimal solution when extended to this setting.

(c) Describe an efficient algorithm to compute the locations of the fuel stations you should stop at to minimize the total cost of travel.

9. Recall that a string w of parentheses **(** and **)** is ***balanced*** if it satisfies one of the following conditions:

   - w is the empty string.
   - $w = (x)$ for some balanced string x
   - $w = xy$ for some balanced strings x and y

   For example, the string
   $$w = ((())()())(()())()$$
   is balanced, because $w = xy$, where
   $$x = (\ (())\ ()\ ()\ ) \qquad \text{and} \qquad y = (\ ()\ ()\ )\ ().$$

   (a) Describe and analyze an algorithm to determine whether a given string of parentheses is balanced.

   (b) Describe and analyze a greedy algorithm to compute the length of a longest balanced subsequence of a given string of parentheses. As usual, don't forget to prove your algorithm is correct.

   For both problems, your input is an array $w[1..n]$, where for each i, either $w[i] = ($ or $w[i] =)$. Both of your algorithms should run in $O(n)$ time.

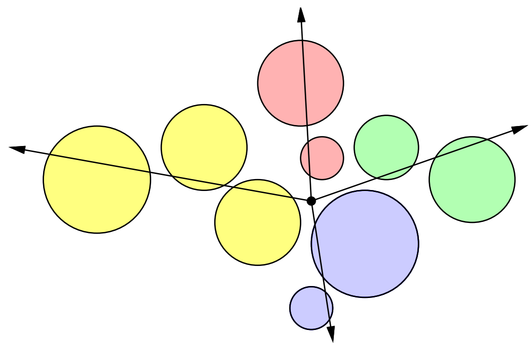
10. Congratulations! You have successfully conquered Camelot, transforming the former battle-scarred kingdom with an anarcho-syndicalist commune, where citizens take turns to act as a sort of executive-officer-for-the-week, but with all the decisions of that officer ratified at a special bi-weekly meeting, by a simple majority in the case of purely internal affairs, but by a two-thirds majority in the case of more major....

    As a final symbolic act, you order the Round Table (surprisingly, an actual circular table) to be split into pizza-like wedges and distributed to the citizens of Camelot as trophies. Each citizen has submitted a request for an angular wedge of the table, specified by two angles—for example: Sir Robin the Brave might request the wedge from $17.23°$ to $42°$, and Sir Lancelot the Pure might request the $2°$ wedge from $359°$ to $1°$. Each citizen will be happy if and only if they receive *precisely* the wedge that they requested. Unfortunately, some of these ranges overlap, so satisfying *all* the citizens' requests is simply impossible. Welcome to politics.

    Describe and analyze an algorithm to find the maximum number of requests that can be satisfied. *[Hint: Careful! The output of your algorithm must not change if you rotate the table. Do not assume that angles are integers.]*

11. Suppose you are standing in a field surrounded by several large balloons. You want to use your brand new Acme Brand Zap-O-Matic™ to pop all the balloons, without moving from your current location. The Zap-O-Matic™ shoots a high-powered laser beam, which pops all the balloons it hits. Since each shot requires enough energy to power a small country for a year, you want to fire as few shots as possible.

    The *minimum zap* problem can be stated more formally as follows. Given a set C of n circles in the plane, each specified by its radius and the (x, y) coordinates of its center, compute the minimum number of rays from the origin that intersect every circle in C. Your goal is to find an efficient algorithm for this problem.



Nine balloons popped by 4 shots of the Zap-O-Matic™

(a) Suppose it is possible to shoot a ray that does not intersect any balloons. Describe and analyze a greedy algorithm that solves the minimum zap problem in this special case. *[Hint: See Exercise 2.]*

(b) Describe and analyze a greedy algorithm whose output is within 1 of optimal. That is, if m is the minimum number of rays required to hit every balloon, then your greedy algorithm must output either m or $m+1$. (Of course, you must prove this fact.)

(c) Describe an algorithm that solves the minimum zap problem in $O(n^2)$ time.

*(d) Describe an algorithm that solves the minimum zap problem in $O(n \log n)$ time.

Assume you have a subroutine INTERSECTS(r, c) that determines whether an arbitrary ray r intersects an arbitrary circle c in $O(1)$ time. This subroutine is not difficult to write, but it's not the interesting part of the problem.

The problem is that we attempt to solve the simplest questions cleverly,
thereby rendering them unusually complex.
One should seek the simple solution.

— Anton Pavlovich Chekhov (c. 1890)

I love deadlines. I like the whooshing sound they make as they fly by.

— Douglas Adams

*8 Matroids

8.1 Definitions

Many problems that can be correctly solved by greedy algorithms can be described in terms of an abstract combinatorial object called a *matroid*. Matroids were first described in 1935 by the mathematician Hassler Whitney as a combinatorial generalization of linear independence of vectors—'matroid' means 'something sort of like a matrix'.

A matroid $\mathcal{M}$ is a finite collection of finite sets that satisfies three axioms:

- **Non-emptiness:** The empty set is in $\mathcal{M}$. (Thus, $\mathcal{M}$ is not itself empty.)
- **Heredity:** If a set X is an element of $\mathcal{M}$, then every subset of X is also in $\mathcal{M}$.
- **Exchange:** If X and Y are two sets in $\mathcal{M}$ where $|X| > |Y|$, then there is an element $x \in X \setminus Y$ such that $Y \cup \{x\}$ is in $\mathcal{M}$.

The sets in $\mathcal{M}$ are typically called ***independent sets***; for example, we would say that any subset of an independent set is independent. The union of all sets in $\mathcal{M}$ is called the ***ground set***. An independent set is called a ***basis*** if it is not a proper subset of another independent set. The exchange property implies that every basis of a matroid has the same cardinality. The ***rank*** of a subset X of the ground set is the size of the largest independent subset of X. A subset of the ground set that is not in $\mathcal{M}$ is called ***dependent*** (surprise, surprise). Finally, a dependent set is called a ***circuit*** if every proper subset is independent.

Most of this terminology is justified by Whitney's original example:

- ***Linear matroid:*** Let A be any $n \times m$ matrix. A subset $I \subseteq \{1, 2, \ldots, n\}$ is independent if and only if the corresponding subset of columns of A is linearly independent.

The heredity property follows directly from the definition of linear independence; the exchange property is implied by an easy dimensionality argument. A basis in any linear matroid is also a basis (in the linear-algebra sense) of the vector space spanned by the columns of A. Similarly, the rank of a set of indices is precisely the rank (in the linear-algebra sense) of the corresponding set of column vectors.

Here are several other examples of matroids; some of these we will see again later. I will leave the proofs that these are actually matroids as exercises for the reader.

- ***Uniform matroid $U_{k,n}$:*** A subset $X \subseteq \{1, 2, \ldots, n\}$ is independent if and only if $|X| \leq k$. Any subset of $\{1, 2, \ldots, n\}$ of size k is a basis; any subset of size $k+1$ is a circuit.

- ***Graphic/cycle matroid*** $\mathcal{M}(G)$: Let $G = (V, E)$ be an arbitrary undirected graph. A subset of E is independent if it defines an *acyclic* subgraph of G. A basis in the graphic matroid is a spanning tree of G; a circuit in this matroid is a cycle in G.
- ***Cographic/cocycle matroid*** $\mathcal{M}^*(G)$: Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq E$ is independent if the complementary subgraph $(V, E \setminus I)$ of G is *connected*. A basis in this matroid is the complement of a spanning tree; a circuit in this matroid is a *cocycle*—a minimal set of edges that disconnects the graph.
- ***Matching matroid:*** Let $G = (V, E)$ be an arbitrary undirected graph. A subset $I \subseteq V$ is independent if there is a matching in G that covers I.
- ***Disjoint path matroid:*** Let $G = (V, E)$ be an arbitrary *directed* graph, and let s be a fixed vertex of G. A subset $I \subseteq V$ is independent if and only if there are edge-disjoint paths from s to each vertex in I.

Now suppose each element of the ground set of a matroid $\mathcal{M}$ is given an arbitrary non-negative weight. The ***matroid optimization problem*** is to compute a basis with maximum total weight. For example, if $\mathcal{M}$ is the cycle matroid for a graph G, the matroid optimization problem asks us to find the maximum spanning tree of G. Similarly, if $\mathcal{M}$ is the cocycle matroid for G, the matroid optimization problem seeks (the complement of) the *minimum* spanning tree.

The following natural greedy strategy computes a basis for any weighted matroid:

```
GreedyBasis(M, w):
  X[1..n] ← ⋃M   ⟨⟨the ground set⟩⟩
  sort X in decreasing order of weight w
  G ← ∅
  for i ← 1 to n
      if G ∪ {X[i]} ∈ M
          add X[i] to G
  return G
```

Suppose we can test in $F(n)$ whether a given subset of the ground set is independent. Then this algorithm runs in $O(n \log n + n \cdot F(n))$ time.

Theorem 1. *For any matroid $\mathcal{M}$ and any weight function w, $\textsc{GreedyBasis}(\mathcal{M}, w)$ returns a maximum-weight basis of $\mathcal{M}$.*

Proof: We use a standard exchange argument. Let $G = \{g_1, g_2, \ldots, g_k\}$ be the independent set returned by $\textsc{GreedyBasis}(\mathcal{M}, w)$. If any other element could be added to G to obtain a larger independent set, the greedy algorithm would have added it. Thus, G is a basis.

For purposes of deriving a contradiction, suppose there is an independent set $H = \{h_1, h_2, \ldots, h_\ell\}$ such that

$$\sum_{i=1}^{k} w(g_i) < \sum_{j=1}^{\ell} w(h_i).$$

Without loss of generality, we assume that H is a basis. The exchange property now implies that $k = \ell$.

Now suppose the elements of G and H are indexed in order of decreasing weight. Let i be the smallest index such that $w(g_i) < w(h_i)$, and consider the independent sets

$$G_{i-1} = \{g_1, g_2, \ldots, g_{i-1}\} \quad \text{and} \quad H_i = \{h_1, h_2, \ldots, h_{i-1}, h_i\}.$$

By the exchange property, there is some element $h_j \in H_i$ such that $G_{i-1} \cup \{h_j\}$ is an independent set. We have $w(h_j) \geq w(h_i) > w(g_i)$. Thus, the greedy algorithm considers *and rejects* the heavier element h_j before it considers the lighter element g_i. But this is impossible—the greedy algorithm accepts elements in decreasing order of weight. $\square$

We now immediately have a correct greedy optimization algorithm for *any* matroid. Returning to our examples:

- Linear matroid: Given a matrix A, compute a subset of vectors of maximum total weight that span the column space of A.
- Uniform matroid: Given a set of weighted objects, compute its k largest elements.
- Cycle matroid: Given a graph with weighted edges, compute its maximum spanning tree. In this setting, the greedy algorithm is better known as *Kruskal's algorithm*.
- Cocycle matroid: Given a graph with weighted edges, compute its minimum spanning tree.
- Matching matroid: Given a graph, determine whether it has a perfect matching.
- Disjoint path matroid: Given a directed graph with a special vertex s, find the largest set of edge-disjoint paths from s to other vertices.

The exchange condition for matroids turns out to be crucial for the success of this algorithm. A *subset system* is a finite collection $\mathcal{S}$ of finite sets that satisfies the heredity condition—If $X \in \mathcal{S}$ and $Y \subseteq X$, then $Y \in \mathcal{S}$—but not necessarily the exchange condition.

Theorem 2. *For any subset system $\mathcal{S}$ that is* ***not*** *a matroid, there is a weight function w such that* GreedyBasis$(\mathcal{S}, w)$ *does* ***not*** *return a maximum-weight set in $\mathcal{S}$.*

Proof: Let X and Y be two sets in $\mathcal{S}$ that violate the exchange property—$|X| > |Y|$, but for any element $x \in X \setminus Y$, the set $Y \cup \{x\}$ is not in $\mathcal{S}$. Let $m = |Y|$. We define a weight function as follows:

- Every element of Y has weight $m + 2$.
- Every element of $X \setminus Y$ has weight $m + 1$.
- Every other element of the ground set has weight zero.

With these weights, the greedy algorithm will consider and accept every element of Y, then consider and reject every element of X, and finally consider all the other elements. The algorithm returns a set with total weight $m(m+2) = m^2 + 2m$. But the total weight of X is at least $(m+1)^2 = m^2 + 2m + 1$. Thus, the output of the greedy algorithm is not the maximum-weight set in $\mathcal{S}$. $\square$

Recall the Applied Chaos scheduling problem considered in the previous lecture note. There is a natural subset system associated with this problem: A set of classes is independent if and only if not two classes overlap. (This is just the graph-theory notion of 'independent set'!) This subset system is *not* a matroid, because there can be maximal independent sets of different sizes, which violates the exchange property. If we consider a *weighted* version of the class scheduling problem, say where each class is worth a different number of hours, Theorem 2 implies that the greedy algorithm will *not* always find the optimal schedule. (In fact, there's an easy counterexample with only two classes!) However, Theorem 2 does *not* contradict the correctness of the greedy algorithm for the original *unweighted* problem, however; that problem uses a particularly lucky choice of weights (all equal).

8.2 Scheduling with Deadlines

Suppose you have n tasks to complete in n days; each task requires your attention for a full day. Each task comes with a *deadline*, the last day by which the job should be completed, and a *penalty* that you must pay if you do not complete each task by its assigned deadline. What order should you perform your tasks in to minimize the total penalty you must pay?

More formally, you are given an array $D[1..n]$ of deadlines an array $P[1..n]$ of penalties. Each deadline $D[i]$ is an integer between 1 and n, and each penalty $P[i]$ is a non-negative real number. A *schedule* is a permutation of the integers $\{1,2,\ldots,n\}$. The scheduling problem asks you to find a schedule π that minimizes the following cost:

$$cost(\pi) := \sum_{i=1}^{n} P[i]\cdot[\pi(i) > D[i]].$$

This doesn't look anything like a matroid optimization problem. For one thing, matroid optimization problems ask us to find an optimal *set*; this problem asks us to find an optimal *permutation*. Surprisingly, however, this scheduling problem is actually a matroid optimization in disguise! For any schedule π, call tasks i such that $\pi(i) > D[i]$ *late*, and all other tasks *on time*. The following trivial observation is the key to revealing the underlying matroid structure.

The cost of a schedule is determined by the subset of tasks that are on time.

Call a subset X of the tasks *realistic* if there is a schedule π in which every task in X is on time. We can precisely characterize the realistic subsets as follows. Let $X(t)$ denote the subset of tasks in X whose deadline is on or before t:

$$X(t) := \{i \in X \mid D[i] \le t\}.$$

In particular, $X(0) = \varnothing$ and $X(n) = X$.

Lemma 3. *Let $X \subseteq \{1,2,\ldots,n\}$ be an arbitrary subset of the n tasks. X is realistic if and only if $|X(t)| \le t$ for every integer t.*

Proof: Let π be a schedule in which every task in X is on time. Let i_t be the tth task in X to be completed. On the one hand, we have $\pi(i_t) \ge t$, since otherwise, we could not have completed $t-1$ other jobs in X before i_t. On the other hand, $\pi(i_t) \le D[i]$, because i_t is on time. We conclude that $D[i_t] \ge t$, which immediately implies that $|X(t)| \le t$.

Now suppose $|X(t)| \le t$ for every integer t. If we perform the tasks in X in increasing order of deadline, then we complete all tasks in X with deadlines t or less by day t. In particular, for any $i \in X$, we perform task i on or before its deadline $D[i]$. Thus, X is realistic. □

We can define a *canonical schedule* for any set X as follows: execute the tasks in X in increasing deadline order, and then execute the remaining tasks in any order. The previous proof implies that a set X is realistic if and only if every task in X is on time in the canonical schedule for X. Thus, our scheduling problem can be rephrased as follows:

Find a realistic subset X such that $\sum_{i\in X} P[i]$ is maximized.

So we're looking for optimal subsets after all.

Lemma 4. *The collection of realistic sets of jobs forms a matroid.*

Proof: The empty set is vacuously realistic, and any subset of a realistic set is clearly realistic. Thus, to prove the lemma, it suffices to show that the exchange property holds. Let X and Y be realistic sets of jobs with $|X| > |Y|$.

Let t^* be the largest integer such that $|X(t^*)| \leq |Y(t^*)|$. This integer must exist, because $|X(0)| = 0 \leq 0 = |Y(0)|$ and $|X(n)| = |X| > |Y| = |Y(n)|$. By definition of t^*, there are more tasks with deadline $t^* + 1$ in X than in Y. Thus, we can choose a task j in $X \setminus Y$ with deadline $t^* + 1$; let $Z = Y \cup \{j\}$.

Let t be an arbitrary integer. If $t \leq t^*$, then $|Z(t)| = |Y(t)| \leq t$, because Y is realistic. On the other hand, if $t > t^*$, then $|Z(t)| = |Y(t)| + 1 \leq |X(t)| < t$ by definition of t^* and because X is realistic. The previous lemma now implies that Z is realistic. This completes the proof of the exchange property. $\square$

This lemma implies that our scheduling problem is a matroid optimization problem, so the greedy algorithm finds the optimal schedule.

```
GreedySchedule(D[1..n], P[1..n]):
  Sort P in increasing order, and permute D to match
  j ← 0
  for i ← 1 to n
      X[j+1] ← i
      if X[1..j+1] is realistic
          j ← j+1
  return the canonical schedule for X[1..j]
```

To turn this outline into a real algorithm, we need a procedure to test whether a given subset of jobs is realistic. Lemma 9 immediately suggests the following strategy to answer this question in $O(n)$ time.

```
Realistic?(X[1..m], D[1..n]):
  ⟨⟨X is sorted by increasing deadline: i ≤ j ⟹ D[X[i]] ≤ D[X[j]]⟩⟩
  N ← 0
  j ← 0
  for t ← 1 to n
      if D[X[j]] = t
          N ← N+1;  j ← j+1
      ⟨⟨Now N = |X(t)|⟩⟩
      if N > t
          return False
  return True
```

If we use this subroutine, GreedySchedule runs in $O(n^2)$ time. By using some appropriate data structures, the running time can be reduced to $O(n \log n)$; details are left as an exercise for the reader.

Exercises

1. Prove that for any graph G, the 'graphic matroid' $\mathcal{M}(G)$ is in fact a matroid. (This problem is really asking you to prove that Kruskal's algorithm is correct!)

2. Prove that for any graph G, the 'cographic matroid' $\mathcal{M}^*(G)$ is in fact a matroid.

3. Prove that for any graph G, the 'matching matroid' of G is in fact a matroid. *[Hint: What is the symmetric difference of two matchings?]*

4. Prove that for any directed graph G and any vertex s of G, the resulting 'disjoint path matroid' of G is in fact a matroid. *[Hint: This question is **much** easier if you're already familiar with maximum flows.]*

5. Let G be an undirected graph. A set of cycles $\{c_1, c_2, \ldots, c_k\}$ in G is called *redundant* if every edge in G appears in an even number of c_i's. A set of cycles is *independent* if it contains no redundant subset. A maximal independent set of cycles is called a *cycle basis* for G.

   (a) Let C be any cycle basis for G. Prove that for any cycle γ in G, there is a subset $A \subseteq C$ such that $A \cap \{\gamma\}$ is redundant. In other words, γ is the 'exclusive or' of the cycles in A.

   (b) Prove that the set of independent cycle sets form a matroid.

   *(c) Now suppose each edge of G has a weight. Define the weight of a cycle to be the total weight of its edges, and the weight of a *set* of cycles to be the total weight of all cycles in the set. (Thus, each edge is counted once for every cycle in which it appears.) Describe and analyze an efficient algorithm to compute the minimum-weight cycle basis in G.

6. Describe a modification of GreedySchedule that runs in $O(n \log n)$ time. *[Hint: Store X in an appropriate data structure that supports the operations "Is $X \cup \{i\}$ realistic?" and "Add i to X" in $O(\log n)$ time each.]*

Randomization

A	A	D	I	N		N	O	M	I	O	T		R	Z	
I	T	A	O	A	R	N	O				N	I	M	D	Z
M	T	I	Z	I			A	R		O	N	N	D	O	A
O	A	D	N	I	Z	T	I	O	N	M			A		R
D	A	N	Z	I	I	T			M		O	O	R	A	N
Z	R		T			A	I	O	N	M	A	O	D	I	N
		Z	T	N	A	D	N	I	R	A	I	O		O	M
	I	N	M	D	Z	O	O		R	T	A	A	N		I
I	T		M	N	R	D	A	O	I	A			O	N	Z
Z		I	N	D	I		T		M	O	R	A	O	A	N
	I		A	O	N	I	T	M	N	Z		A	O	R	D
A	N	R		O	I	T	I	A	D		N	Z	M		O
T		O	N	Z		N	O	D	R	M	A		I	I	A
A	O	O	I	A	N	Z	T	N	D			I		M	R
A		I	O	D	M	N	T	R	O	N		Z	A	I	
N		A	I		Z	O	R	D	O	T	I		A	N	M

The first nuts and bolts appeared in the middle 1400's. The bolts were just screws with straight sides and a blunt end. The nuts were hand-made, and very crude. When a match was found between a nut and a bolt, they were kept together until they were finally assembled.

In the Industrial Revolution, it soon became obvious that threaded fasteners made it easier to assemble products, and they also meant more reliable products. But the next big step came in 1801, with Eli Whitney, the inventor of the cotton gin. The lathe had been recently improved. Batches of bolts could now be cut on different lathes, and they would all fit the same nut.

Whitney set up a demonstration for President Adams, and Vice-President Jefferson. He had piles of musket parts on a table. There were 10 similar parts in each pile. He went from pile to pile, picking up a part at random. Using these completely random parts, he quickly put together a working musket.

— Karl S. Kruszelnicki ("Dr. Karl"), *Karl Trek*, December 1997

Dr [John von] Neumann in his Theory of Games and Economic Behavior *introduces the cut-up method of random action into game and military strategy: Assume that the worst has happened and act accordingly. If your strategy is at some point determined. . . by random factor your opponent will gain no advantage from knowing your strategy since he cannot predict the move. The cut-up method could be used to advantage in processing scientific data. How many discoveries have been made by accident? We cannot produce accidents to order.*

— William S. Burroughs, "The Cut-Up Method of Brion Gysin"
in *The Third Mind* by William S. Burroughs and Brion Gysin (1978)

9 Randomized Algorithms

9.1 Nuts and Bolts

Suppose we are given n nuts and n bolts of different sizes. Each nut matches exactly one bolt and vice versa. The nuts and bolts are all almost exactly the same size, so we can't tell if one bolt is bigger than the other, or if one nut is bigger than the other. If we try to match a nut witch a bolt, however, the nut will be either too big, too small, or just right for the bolt.

Our task is to match each nut to its corresponding bolt. But before we do this, let's try to solve some simpler problems, just to get a feel for what we can and can't do.

Suppose we want to find the nut that matches a particular bolt. The obvious algorithm — test every nut until we find a match — requires exactly $n-1$ tests in the worst case. We might have to check every bolt except one; if we get down the the last bolt without finding a match, we know that the last nut is the one we're looking for.[1]

Intuitively, in the 'average' case, this algorithm will look at approximately $n/2$ nuts. But what exactly does 'average case' mean?

9.2 Deterministic vs. Randomized Algorithms

Normally, when we talk about the running time of an algorithm, we mean the *worst-case* running time. This is the maximum, over all problems of a certain size, of the running time of that algorithm on that input:

$$T_{\text{worst-case}}(n) = \max_{|X|=n} T(X).$$

On extremely rare occasions, we will also be interested in the *best-case* running time:

$$T_{\text{best-case}}(n) = \min_{|X|=n} T(X).$$

[1]"Whenever you lose something, it's always in the last place you look. So why not just look there first?"

The *average-case* running time is best defined by the *expected value*, over all inputs X of a certain size, of the algorithm's running time for X:[2]

$$T_{\text{average-case}}(n) = \mathop{\mathrm{E}}_{|X|=n}[T(X)] = \sum_{|X|=n} T(x) \cdot \Pr[X].$$

The problem with this definition is that we rarely, if ever, know what the probability of getting any particular input X is. We could compute average-case running times by assuming a particular probability distribution—for example, every possible input is equally likely—but this assumption doesn't describe reality very well. Most real-life data is decidedly non-random (or at least random in some unpredictable way).

Instead of considering this rather questionable notion of average case running time, we will make a distinction between two kinds of algorithms: *deterministic* and *randomized*. A deterministic algorithm is one that always behaves the same way given the same input; the input completely *determines* the sequence of computations performed by the algorithm. Randomized algorithms, on the other hand, base their behavior not only on the input but also on several *random* choices. The same randomized algorithm, given the same input multiple times, may perform different computations in each invocation.

This means, among other things, that the running time of a randomized algorithm on a given input is no longer fixed, but is itself a random variable. When we analyze randomized algorithms, we are typically interested in the *worst-case expected* running time. That is, we look at the average running time for each input, and then choose the maximum over all inputs of a certain size:

$$T_{\text{worst-case expected}}(n) = \max_{|X|=n} \mathrm{E}[T(X)].$$

It's important to note here that we are making *no* assumptions about the probability distribution of possible inputs. All the randomness is inside the algorithm, where we can control it!

9.3 Back to Nuts and Bolts

Let's go back to the problem of finding the nut that matches a given bolt. Suppose we use the same algorithm as before, but at each step we choose a nut *uniformly at random* from the untested nuts. 'Uniformly' is a technical term meaning that each nut has exactly the same probability of being chosen.[3] So if there are k nuts left to test, each one will be chosen with probability $1/k$. Now what's the expected number of comparisons we have to perform? Intuitively, it should be about $n/2$, but let's formalize our intuition.

Let $T(n)$ denote the number of comparisons our algorithm uses to find a match for a single bolt out of n nuts.[4] We still have some simple base cases $T(1) = 0$ and $T(2) = 1$, but when $n > 2$, $T(n)$ is a random variable. $T(n)$ is always between 1 and $n-1$; it's actual value depends on our algorithm's random choices. We are interested in the *expected value* or *expectation* of $T(n)$, which is defined as follows:

$$\mathrm{E}[T(n)] = \sum_{k=1}^{n-1} k \cdot \Pr[T(n) = k]$$

[2] The notation E[ ] for expectation has nothing to do with the shift operator **E** used in the annihilator method for solving recurrences!

[3] This is what most people think 'random' means, but they're wrong.

[4] Note that for this algorithm, the input is completely specified by the number n. Since we're choosing the nuts to test at random, even the order in which the nuts and bolts are presented doesn't matter. That's why I'm using the simpler notation $T(n)$ instead of $T(X)$.

If the target nut is the kth nut tested, our algorithm performs $\min\{k, n-1\}$ comparisons. In particular, if the target nut is the last nut chosen, we don't actually test it. Because we choose the next nut to test uniformly at random, the target nut is equally likely—with probability exactly $1/n$—to be the first, second, third, or kth bolt tested, for any k. Thus:

$$\Pr[T(n) = k] = \begin{cases} 1/n & \text{if } k < n-1, \\ 2/n & \text{if } k = n-1. \end{cases}$$

Plugging this into the definition of expectation gives us our answer.

$$\begin{aligned} \mathrm{E}[T(n)] &= \sum_{k=1}^{n-2} \frac{k}{n} + \frac{2(n-1)}{n} \\ &= \sum_{k=1}^{n-1} \frac{k}{n} + \frac{n-1}{n} \\ &= \frac{n(n-1)}{2n} + 1 - \frac{1}{n} \\ &= \frac{n+1}{2} - \frac{1}{n} \end{aligned}$$

We can get exactly the same answer by thinking of this algorithm recursively. We always have to perform at least one test. With probability $1/n$, we successfully find the matching nut and halt. With the remaining probability $1 - 1/n$, we recursively solve the same problem but with one fewer nut. We get the following recurrence for the expected number of tests:

$$T(1) = 0, \qquad \mathrm{E}[T(n)] = 1 + \frac{n-1}{n}\,\mathrm{E}[T(n-1)]$$

To get the solution, we define a new function $t(n) = n\,\mathrm{E}[T(n)]$ and rewrite:

$$t(1) = 0, \qquad t(n) = n + t(n-1)$$

This recurrence translates into a simple summation, which we can easily solve.

$$\begin{aligned} t(n) &= \sum_{k=2}^{n} k = \frac{n(n+1)}{2} - 1 \\ \Longrightarrow\ \mathrm{E}[T(n)] &= \frac{t(n)}{n} = \frac{n+1}{2} - \frac{1}{n} \end{aligned}$$

9.4 Finding All Matches

Not let's go back to the problem introduced at the beginning of the lecture: finding the matching nut for every bolt. The simplest algorithm simply compares every nut with every bolt, for a total of n^2 comparisons. The next thing we might try is repeatedly finding an arbitrary matched pair, using our very first nuts and bolts algorithm. This requires

$$\sum_{i=1}^{n} (i-1) = \frac{n^2 - n}{2}$$

comparisons in the worst case. So we save roughly a factor of two over the really stupid algorithm. Not very exciting.

Here's another possibility. Choose a *pivot* bolt, and test it against every nut. Then test the matching pivot nut against every other bolt. After these $2n-1$ tests, we have one matched pair, and the remaining nuts and bolts are partitioned into two subsets: those smaller than the pivot pair and those larger than the pivot pair. Finally, recursively match up the two subsets. The worst-case number of tests made by this algorithm is given by the recurrence

$$\begin{aligned} T(n) &= 2n-1+\max_{1\le k\le n}\{T(k-1)+T(n-k)\} \\ &= 2n-1+T(n-1) \end{aligned}$$

Along with the trivial base case $T(0)=0$, this recurrence solves to

$$T(n)=\sum_{i=1}^{n}(2n-1)=n^2.$$

In the worst case, this algorithm tests *every* nut-bolt pair! We could have been a little more clever—for example, if the pivot bolt is the smallest bolt, we only need $n-1$ tests to partition everything, not $2n-1$—but cleverness doesn't actually help that much. We still end up with about $n^2/2$ tests in the worst case.

However, since this recursive algorithm looks almost exactly like quicksort, and everybody 'knows' that the 'average-case' running time of quicksort is $\Theta(n\log n)$, it seems reasonable to guess that the average number of nut-bolt comparisons is also $\Theta(n\log n)$. As we shall see shortly, if the pivot bolt is always chosen *uniformly at random*, this intuition is exactly right.

9.5 Reductions to and from Sorting

The second algorithm for mathing up the nuts and bolts looks exactly like quicksort. The algorithm not only matches up the nuts and bolts, but also sorts them by size.

In fact, the problems of sorting and matching nuts and bolts are equivalent, in the following sense. If the bolts were sorted, we could match the nuts and bolts in $O(n\log n)$ time by performing a binary search with each nut. Thus, if we had an algorithm to sort the bolts in $O(n\log n)$ time, we would immediately have an algorithm to match the nuts and bolts, starting from scratch, in $O(n\log n)$ time. This process of *assuming* a solution to one problem and using it to solve another is called *reduction*—we can *reduce* the matching problem to the sorting problem in $O(n\log n)$ time.

There is a reduction in the other direction, too. If the nuts and bolts were matched, we could sort them in $O(n\log n)$ time using, for example, merge sort. Thus, if we have an $O(n\log n)$ time algorithm for either sorting or matching nuts and bolts, we automatically have an $O(n\log n)$ time algorithm for the other problem.

Unfortunately, since we aren't allowed to directly compare two bolts or two nuts, we can't use heapsort or mergesort to sort the nuts and bolts in $O(n\log n)$ worst case time. In fact, the problem of sorting nuts and bolts *deterministically* in $O(n\log n)$ time was only 'solved' in 1995[5], but both the algorithms and their analysis are incredibly technical and the constant hidden in the $O(\cdot)$ notation is quite large.

Reductions will come up again later in the course when we start talking about lower bounds and NP-completeness.

[5]János Komlós, Yuan Ma, and Endre Szemerédi, Sorting nuts and bolts in $O(n\log n)$ time, *SIAM J. Discrete Math* 11(3):347–372, 1998. See also Phillip G. Bradford, Matching nuts and bolts optimally, Technical Report MPI-I-95-1-025, Max-Planck-Institut für Informatik, September 1995. Bradford's algorithm is *slightly* simpler.

9.6 Recursive Analysis

Intuitively, we can argue that our quicksort-like algorithm will usually choose a bolt of approximately median size, and so the average numbers of tests should be $O(n \log n)$. We can now finally formalize this intuition. To simplify the notation slightly, I'll write $\overline{T}(n)$ in place of $\mathrm{E}[T(n)]$ everywhere.

Our randomized matching/sorting algorithm chooses its pivot bolt *uniformly at random* from the set of unmatched bolts. Since the pivot bolt is equally likely to be the smallest, second smallest, or kth smallest for any k, the expected number of tests performed by our algorithm is given by the following recurrence:

$$\begin{aligned}\overline{T}(n) &= 2n - 1 + \mathrm{E}_k\left[\overline{T}(k-1) + \overline{T}(n-k)\right]\\ &= \boxed{2n - 1 + \frac{1}{n}\sum_{k=1}^{n}\left(\overline{T}(k-1) + \overline{T}(n-k)\right)}\end{aligned}$$

The base case is $T(0) = 0$. (We can save a few tests by setting $T(1) = 0$ instead of 1, but the analysis will be easier if we're a little stupid.)

Yuck. At this point, we could simply *guess* the solution, based on the incessant rumors that quicksort runs in $O(n \log n)$ time in the average case, and prove our guess correct by induction. (See Section 9.8 below for details.)

However, if we're only interested in asymptotic bounds, we can afford to be a little conservative. What we'd *really* like is for the pivot bolt to be the median bolt, so that half the bolts are bigger and half the bolts are smaller. This isn't very likely, but there is a good chance that the pivot bolt is close to the median bolt. Let's say that a pivot bolt is *good* if it's in the middle half of the final sorted set of bolts, that is, bigger than at least $n/4$ bolts and smaller than at least $n/4$ bolts. If the pivot bolt is good, then the *worst* split we can have is into one set of $3n/4$ pairs and one set of $n/4$ pairs. If the pivot bolt is bad, then our algorithm is still better than starting over from scratch. Finally, a randomly chosen pivot bolt is good with probability $1/2$.

These simple observations give us the following simple recursive *upper bound* for the expected running time of our algorithm:

$$\overline{T}(n) \le 2n - 1 + \frac{1}{2}\left(\overline{T}\left(\frac{3n}{4}\right) + \overline{T}\left(\frac{n}{4}\right)\right) + \frac{1}{2} \cdot \overline{T}(n)$$

A little algebra simplifies this even further:

$$\overline{T}(n) \le 4n - 2 + \overline{T}\left(\frac{3n}{4}\right) + \overline{T}\left(\frac{n}{4}\right)$$

We can solve this recurrence using the recursion tree method, giving us the unsurprising upper bound $\overline{T}(n) = O(n \log n)$. A similar argument gives us the matching lower bound $\overline{T}(n) = \Omega(n \log n)$.

Unfortunately, while this argument is convincing, it is *not* a formal proof, because it relies on the unproven assumption that $\overline{T}(n)$ is a *convex* function, which means that $\overline{T}(n+1) + \overline{T}(n-1) \ge 2\overline{T}(n)$ for all n. $\overline{T}(n)$ is actually convex, but we never proved it. Convexity follows form the closed-form solution of the recurrence, but using that fact would be circular logic. Sadly, formally proving convexity seems to be almost as hard as solving the recurrence. If we want a *proof* of the expected cost of our algorithm, we need another way to proceed.

9.7 Iterative Analysis

By making a simple change to our algorithm, which has no effect on the number of tests, we can analyze it much more directly and exactly, without solving a recurrence or relying on hand-wavy intuition.

The recursive subproblems solved by quicksort can be laid out in a binary tree, where each node corresponds to a subset of the nuts and bolts. In the usual recursive formulation, the algorithm partitions the nuts and bolts at the root, then the left child of the root, then the leftmost grandchild, and so forth, recursively sorting everything on the left before starting on the right subproblem.

But we don't have to solve the subproblems in this order. In fact, we can visit the nodes in the recursion tree in any order we like, as long as the root is visited first, and any other node is visited after its parent. Thus, we can recast quicksort in the following iterative form. Choose a pivot bolt, find its match, and partition the remaining nuts and bolts into two subsets. Then pick a second pivot bolt and partition whichever of the two subsets contains it. At this point, we have two matched pairs and three subsets of nuts and bolts. Continue choosing new pivot bolts and partitioning subsets, each time finding one match and increasing the number of subsets by one, until every bolt has been chosen as the pivot. At the end, every bolt has been matched, and the nuts and bolts are sorted.

Suppose we always choose the next pivot bolt *uniformly at random* from the bolts that haven't been pivots yet. Then no matter which subset contains this bolt, the pivot bolt is equally likely to be any bolt *in that subset*. That implies (by induction) that our randomized iterative algorithm performs *exactly* the same set of tests as our randomized recursive algorithm, but possibly in a different order.

Now let B_i denote the ith smallest bolt, and N_j denote the jth smallest nut. For each i and j, define an indicator variable X_{ij} that equals 1 if our algorithm compares B_i with N_j and zero otherwise. Then the total number of nut/bolt comparisons is exactly

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} X_{ij}.$$

We are interested in the expected value of this double summation:

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathrm{E}[X_{ij}].$$

This equation uses a crucial property of random variables called *linearity of expectation*: for any random variables X and Y, the sum of their expectations is equal to the expectation of their sum: $E[X+Y] = E[X] + E[Y]$.

To analyze our algorithm, we only need to compute the expected value of each X_{ij}. By definition of expectation,

$$\mathrm{E}[X_{ij}] = 0 \cdot \Pr[X_{ij} = 0] + 1 \cdot \Pr[X_{ij} = 1] = \Pr[X_{ij} = 1],$$

so we just need to calculate $\Pr[X_{ij} = 1]$ for all i and j.

First let's assume that $i < j$. The only comparisons our algorithm performs are between some pivot bolt (or its partner) and a nut (or bolt) in the same subset. The only event that can prevent a comparison between B_i and N_j is choosing some intermediate pivot bolt B_k, with $i < k < j$, before B_i or B_j. In other words:

Our algorithm compares B_i and N_j if and only if the first pivot chosen from the set $\{B_i, B_{i+1}, \ldots, B_j\}$ is either B_i or B_j.

Since the set $\{B_i, B_{i+1}, \ldots, B_j\}$ contains $j-i+1$ bolts, each of which is equally likely to be chosen first, we immediately have

$$\mathrm{E}[X_{ij}] = \frac{2}{j-i+1} \qquad \text{for all } i<j.$$

Symmetric arguments give us $\mathrm{E}[X_{ij}] = \frac{2}{i-j+1}$ for all $i>j$. Since our algorithm is a little stupid, every bolt is compared with its partner, so $X_{ii} = 1$ for all i. (In fact, if a pivot bolt is the only bolt in its subset, we don't need to compare it against its partner, but this improvement complicates the analysis.)

Putting everything together, we get the following summation.

$$\begin{aligned}
\mathrm{E}[T(n)] &= \sum_{i=1}^{n}\sum_{j=1}^{n} \mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n} \mathrm{E}[X_{ii}] + 2\sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\
&= \boxed{n + 4\sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{j-i+1}}
\end{aligned}$$

This is quite a bit simpler than the recurrence we got before. With just a few more lines of algebra, we can turn it into an exact, closed-form expression for the expected number of comparisons.

$$\begin{aligned}
\mathrm{E}[T(n)] &= n + 4\sum_{i=1}^{n}\sum_{k=2}^{n-i+1} \frac{1}{k} && [\text{substitute } k = j-i+1] \\
&= n + 4\sum_{k=2}^{n}\sum_{i=1}^{n-k+1} \frac{1}{k} && [\text{reorder summations}] \\
&= n + 4\sum_{k=2}^{n} \frac{n-k+1}{k} \\
&= n + 4\left((n+1)\sum_{k=2}^{n}\frac{1}{k} - \sum_{k=2}^{n} 1\right) \\
&= n + 4((n+1)(H_n - 1) - (n-1)) \\
&= n + 4(nH_n - 2n + H_n) \\
&= \boxed{4nH_n - 7n + 4H_n}
\end{aligned}$$

Sure enough, it's $\Theta(n \log n)$.

*9.8 Masochistic Analysis

If we're feeling particularly masochistic, we can actually solve the recurrence directly, all the way to an exact closed-form solution. I'm including this only to show you it can be done; this won't be on the test.

First we simplify the recurrence slightly by combining symmetric terms.

$$\overline{T}(n) \;=\; \frac{1}{n}\sum_{k=1}^{n}\bigl(\overline{T}(k-1)+\overline{T}(n-k)\bigr) \;+\; 2n-1 \;=\; \frac{2}{n}\sum_{k=0}^{n-1}\overline{T}(k) \;+\; 2n-1$$

We then convert this 'full history' recurrence into a 'limited history' recurrence by shifting and subtracting away common terms. (I call this "Magic step #1".) To make this step slightly easier, we first multiply both sides of the recurrence by n to get rid of the fractions.

$$\begin{aligned}
n\overline{T}(n) &= 2\sum_{k=0}^{n-1}\overline{T}(k) \;+\; 2n^2-n\\
(n-1)\overline{T}(n-1) &= 2\sum_{k=0}^{n-2}\overline{T}(k) \;+\; 2(n-1)^2-(n-1)\\
&= 2\sum_{k=0}^{n-2}\overline{T}(k) \;+\; 2n^2-5n+3\\
n\overline{T}(n)-(n-1)\overline{T}(n-1) &= 2T(n-1)+4n-3\\
\overline{T}(n) &= \frac{n+1}{n}\overline{T}(n-1)+4-\frac{3}{n}
\end{aligned}$$

To solve this limited-history recurrence, we define a new function $t(n)=\overline{T}(n)/(n+1)$. (I call this "Magic step #2".) This gives us an even simpler recurrence for $t(n)$ in terms of $t(n-1)$:

$$\begin{aligned}
t(n) &= \frac{\overline{T}(n)}{n+1}\\
&= \frac{1}{n+1}\left((n+1)\frac{T(n-1)}{n}+4-\frac{3}{n}\right)\\
&= t(n-1)+\frac{4}{n+1}-\frac{3}{n(n+1)}\\
&= t(n-1)+\frac{7}{n+1}-\frac{3}{n}
\end{aligned}$$

I used the technique of partial fractions (remember calculus?) to replace $\frac{1}{n(n+1)}$ with $\frac{1}{n}-\frac{1}{n+1}$ in the last step. The base case for this recurrence is $t(0)=0$. Once again, we have a recurrence that translates directly into a summation, which we can solve with just a few lines of algebra.

$$\begin{aligned}
t(n) &= \sum_{i=1}^{n}\left(\frac{7}{i+1}-\frac{3}{i}\right)\\
&= 7\sum_{i=1}^{n}\frac{1}{i+1}-3\sum_{i=1}^{n}\frac{1}{i}\\
&= 7(H_{n+1}-1)-3H_n\\
&= 4H_n-7+\frac{7}{n+1}
\end{aligned}$$

The last step uses the recursive definition of the harmonic numbers: $H_n=H_n+\frac{1}{n+1}$. Finally, substituting $\overline{T}(n)=(n+1)t(n)$ and simplifying gives us the exact solution to the original recurrence.

$$\overline{T}(n)=4(n+1)H_n-7(n+1)+7=\boxed{4nH_n-7n+4H_n}$$

Surprise, surprise, we get exactly the same solution!

Exercises

Probability

Several of these problems refer to decks of playing cards. A standard (Anglo-American) deck of 52 playing cards contains 13 cards in each of four suits: spades (♠), hearts (♥), diamonds (♦), and clubs (♣). Within each suit, the 13 cards have distinct *ranks*: 2, 3, 4, 5, 6, 7, 8, 9, 10, jack (J), queen (Q), king (K), and ace (A). For purposes of these problems, the ranks are ordered $A < 2 < 3 < \cdots < 9 < 10 < J < Q < K$; thus, for example, the jack of spades has higher rank thank the eight of diamonds.

1. Clock Solitaire is played with a standard deck of playing cards. To set up the game, deal the cards face down into 13 piles of four cards each, one in each of the 'hour' positions of a clock and one in the center. Each pile corresponds to a particular rank—A through Q in clockwise order for the hour positions, and K for the center. To start the game, turn over a card in the center pile. Then repeatedly turn over a card in the pile corresponding to the value of the previous card. The game ends when you try to turn over a card from a pile whose four cards are already face up. (This is always the center pile—why?) You win if and only if every card is face up when the game ends.

   What is the *exact* probability that you win a game of Clock Solitaire, assuming that the cards are permuted uniformly at random before they are dealt into their piles?

2. Professor Jay is about to perform a public demonstration with two decks of cards, one with red backs ('the red deck') and one with blue backs ('the blue deck'). Both decks lie face-down on a table in front of Professor Jay, shuffled so that every permutation of each deck is equally likely.

   To begin the demonstration, Professor Jay turns over the top card from each deck. If one of these two cards is the three of clubs (3♣), the demonstration ends immediately. Otherwise, the good Professor repeatedly hurls the cards he just turned over into the thick, pachydermatous outer melon layer of a nearby watermelon, and then turns over the next card from the top of each deck. The demonstration ends the first time a 3♣ is turned over. Thus, if 3♣ is the last card in both decks, the demonstration ends with 102 cards embedded in the watermelon, that most prodigious of household fruits.

   (a) What is the *exact* expected number of cards that Professor Jay hurls into the watermelon?

   (b) For each of the statements below, give the *exact* probability that the statement is true of the ***first*** pair of cards Professor Jay turns over.

      i. Both cards are threes.
      ii. One card is a three, and the other card is a club.
      iii. If (at least) one card is a heart, then (at least) one card is a diamond.
      iv. The card from the red deck has higher rank than the card from the blue deck.

   (c) For each of the statements below, give the *exact* probability that the statement is true of the ***last*** pair of cards Professor Jay turns over.

      i. Both cards are threes.
      ii. One card is a three, and the other card is a club.

iii. If (at least) one card is a heart, then (at least) one card is a diamond.
iv. The card from the red deck has higher rank than the card from the blue deck.

3. Penn and Teller agree to play the following game. Penn shuffles a standard deck of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3♣), at which point the remaining undrawn cards instantly burst into flames.

   The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.[6] To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

   (a) What is the expected number of cards that Teller draws?
   (b) What is the expected *maximum* value among the cards Teller gives to Penn?
   (c) What is the expected *minimum* value among the cards Teller gives to Penn?
   (d) What is the expected number of cards that Teller gives to Penn? *[Hint: Let* $13 = n$*.]*

4. Suppose n lights labeled $0, \ldots, n-1$ are placed clockwise around a circle. Initially, every light is off. Consider the following random process.

```
LightTheCircle(n):
  k ← 0
  turn on light 0
  while at least one light is off
      with probability 1/2
          k ← (k + 1) mod n
      else
          k ← (k − 1) mod n
      if light k is off, turn it on
```

   (a) Let $p(i, n)$ be the probability that light i is the last to be turned on by LightTheCircle$(n, 0)$. For example, $p(0, 2) = 0$ and $p(1, 2) = 1$. Find an exact closed-form expression for $p(i, n)$ in terms of n and i. Prove your answer is correct.
   (b) Give the tightest upper bound you can on the expected running time of this algorithm.

5. Consider a random walk on a path with vertices numbered $1, 2, \ldots, n$ from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n.

   (a) Prove that the probability that the walk ends by falling off the *right* end of the path is exactly $1/(n + 1)$.
   (b) Prove that if we start at vertex k, the probability that we fall off the *right* end of the path is exactly $k/(n + 1)$.

[6]Specifically, he hurls it directly into the back of Penn's right hand.

(c) Prove that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n.

(d) Suppose we start at vertex $n/2$ instead. State and prove a tight Θ-bound on the expected length of the random walk in this case.

Randomized Algorithms

6. Consider the following randomized algorithm for generating biased random bits. The subroutine FairCoin returns either 0 or 1 with equal probability; the random bits returned by FairCoin are mutually independent.

```
OneInThree:
  if FairCoin = 0
      return 0
  else
      return 1 − OneInThree
```

(a) Prove that OneInThree returns 1 with probability 1/3.

(b) What is the *exact* expected number of times that this algorithm calls FairCoin?

(c) Now suppose you are *given* a subroutine OneInThree that generates a random bit that is equal to 1 with probability 1/3. Describe a FairCoin algorithm that returns either 0 or 1 with equal probability, using OneInThree as your only source of randomness.

(d) What is the *exact* expected number of times that your FairCoin algorithm calls OneInThree?

7. (a) Suppose you have access to a function FairCoin that returns a single random bit, chosen uniformly and independently from the set $\{0, 1\}$, in $O(1)$ time. Describe and analyze an algorithm Random(n), which returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$.

(b) Suppose you have access to a function FairCoins(k) that returns k random bits (or equivalently, a random integer chosen uniformly and independently from the set $\{0, 1, \ldots, 2^k - 1\}$) in $O(1)$ time, given any non-negative integer k as input. Describe and analyze an algorithm Random(n), which returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$.

For each of the remaining problems, you may assume a function Random(k) that returns, given any positive integer k, an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, k\}$, in $O(1)$ time. For example, to perform a fair coin flip, one could call Random(2).

8. Consider the following algorithm for finding the smallest element in an unsorted array:

```
RandomMin(A[1 .. n]):
  min ← ∞
  for i ← 1 to n in random order
      if A[i] < min
          min ← A[i]   (⋆)
  return min
```

(a) In the worst case, how many times does RANDOMMIN execute line ($\star$)?

(b) What is the probability that line ($\star$) is executed during the nth iteration of the for loop?

(c) What is the *exact* expected number of executions of line ($\star$)?

9. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

   What is the *exact* expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. *[Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?]*

10. Let S be a set of n points in the plane. A point p in S is called *Pareto-optimal* if no other point in S is both above and to the right of p.

    (a) Describe and analyze a deterministic algorithm that computes the Pareto-optimal points in S in $O(n \log n)$ time.

    (b) Suppose each point in S is chosen independently and uniformly at random from the unit square $[0,1] \times [0,1]$. What is the *exact* expected number of Pareto-optimal points in S?

11. Suppose we want to write an efficient function RANDOMPERMUTATION(n) that returns a permutation of the integers $\langle 1, \ldots, n \rangle$ chosen uniformly at random.

    (a) Prove that the following algorithm is ***not*** correct. *[Hint: Consider the case $n = 3$.]*

```
RandomPermutation(n):
  for i ← 1 to n
      π[i] ← i
  for i ← 1 to n
      swap π[i] ↔ π[Random(n)]
```

    (b) Consider the following implementation of RANDOMPERMUTATION.

```
RandomPermutation(n):
  for i ← 1 to n
      π[i] ← NULL
  for i ← 1 to n
      j ← Random(n)
      while (π[j] != NULL)
          j ← Random(n)
      π[j] ← i
  return π
```

    Prove that this algorithm is correct. Analyze its expected runtime.

    (c) Consider the following partial implementation of RANDOMPERMUTATION.

```
RandomPermutation(n):
  for i ← 1 to n
      A[i] ← Random(n)
  π ← SomeFunction(A)
  return π
```

Prove that if the subroutine SomeFunction is deterministic, then this algorithm cannot be correct. *[Hint: There is a one-line proof.]*

(d) Describe and analyze an implementation of RandomPermutation that runs in expected worst-case time $O(n)$.

(e) Describe and analyze an implementation of RandomPermutation that runs in expected worst-case time $O(n \log n)$, using fair coin flips (instead of Random) as the only source of randomness.

*(f) Consider a correct implementation of RandomPermutation(n) with the following property: whenever it calls Random(k), the argument k is at most m. Prove that this algorithm ***always*** calls Random at least $\Omega(\frac{n \log n}{\log m})$ times.

12. A *data stream* is an extremely long sequence of items that you can only read only once, in order. A good example of a data stream is the sequence of packets that pass through a router. Data stream algorithms must process each item in the stream quickly, using very little memory; there is simply too much data to store, and it arrives too quickly for any complex computations. Every data stream algorithm looks roughly like this:

```
DoSomethingInteresting(stream S):
  repeat
      x ← next item in S
      ⟨⟨do something fast with x⟩⟩
  until S ends
  return ⟨⟨something⟩⟩
```

Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, *without knowing the length of the stream in advance*. Your algorithm should spend $O(1)$ time per stream element and use $O(1)$ space (not counting the stream itself).

13. Consider the following randomized variant of one-armed quicksort, which selects the kth smallest element in an unsorted array $A[1..n]$. As usual, Partition($A[1..n], p$) partitions the array A into three parts by comparing the pivot element $A[p]$ to every other element, using $n-1$ comparisons, and returns the new index of the pivot element.

```
QuickSelect(A[1..n], k):
  r ← Partition(A[1..n], Random(n))
  if k < r
      return QuickSelect(A[1..r−1], k)
  else if k > r
      return QuickSelect(A[r+1..n], k−r)
  else
      return A[k]
```

(a) State a recurrence for the expected running time of QuickSelect, as a function of n and k.

(b) What is the *exact* probability that QuickSelect compares the ith smallest and jth smallest elements in the input array? The correct answer is a simple function of i, j, and k. *[Hint: Check your answer by trying a few small examples.]*

(c) What is the *exact* probability that in one of the recursive calls to QuickSelect, the first argument is the subarray $A[i\,..\,j]$? The correct answer is a simple function of i, j, and k. *[Hint: Check your answer by trying a few small examples.]*

(d) Show that for any n and k, the expected running time of QuickSelect is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b) or (c). For extra credit, find the *exact* expected number of comparisons, as a function of n and k.

(e) What is the expected number of times that QuickSelect calls itself recursively?

14. Let $M[1\,..\,n, 1\,..\,n]$ be an $n \times n$ matrix in which every row and every column is sorted. Such an array is called *totally monotone*. No two elements of M are equal.

(a) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, compute the number of elements of M smaller than $M[i, j]$ and larger than $M[i', j']$.

(b) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, return an element of M chosen uniformly at random from the elements smaller than $M[i, j]$ and larger than $M[i', j']$. Assume the requested range is always non-empty.

(c) Describe and analyze a randomized algorithm to compute the median element of M in $O(n \log n)$ expected time.

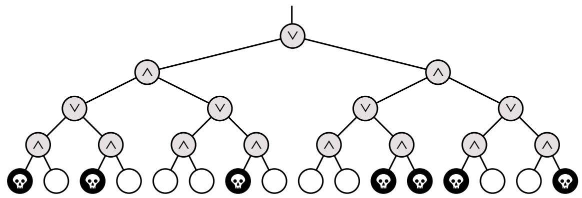
15. Suppose we have a circular linked list of numbers, implemented as a pair of arrays, one storing the actual numbers and the other storing successor pointers. Specifically, let $X[1\,..\,n]$ be an array of n distinct real numbers, and let $N[1\,..\,n]$ be an array of indices with the following property: If $X[i]$ is the largest element of X, then $X[N[i]]$ is the smallest element of X; otherwise, $X[N[i]]$ is the smallest element of X. For example:

i	1	2	3	4	5	6	7	8	9
$X[i]$	83	54	16	31	45	99	78	62	27
$N[i]$	6	8	9	5	2	3	1	7	4

Describe and analyze a randomized algorithm that determines whether a given number x appears in the array X in $O(\sqrt{n})$ expected time. ***Your algorithm may not modify the arrays X and N.***

16. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will

take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



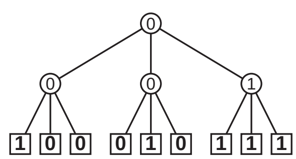
You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

(b) Unfortunately, Death won't give you enough time to look at every node in the tree. Describe a *randomized* algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*

*(c) Describe and analyze a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. *[Hint: You may not need to change your algorithm from part (b) at all!]*

17. A *majority tree* is a complete binary tree with depth n, where every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. Consider the problem of computing the value of the root of a majority tree, given the sequence of 3^n leaf labels as input. For example, if $n = 2$ and the leaves are labeled $1, 0, 0, 0, 1, 0, 1, 1, 1$, the root has value 0.



A majority tree with depth $n = 2$.

(a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. *[Hint: Consider the special case $n = 1$. Recurse.]*

(b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. *[Hint: Consider the special case $n = 1$. Recurse.]*

I thought the following four [rules] would be enough, provided that I made a firm and constant resolution not to fail even once in the observance of them. The first was never to accept anything as true if I had not evident knowledge of its being so.... The second, to divide each problem I examined into as many parts as was feasible, and as was requisite for its better solution. The third, to direct my thoughts in an orderly way... establishing an order in thought even when the objects had no natural priority one to another. And the last, to make throughout such complete enumerations and such general surveys that I might be sure of leaving nothing out.

— René Descartes, *Discours de la Méthode* (1637)

What is luck?
Luck is probability taken personally.
It is the excitement of bad math.

— Penn Jillette (2001), quoting Chip Denman (1998)

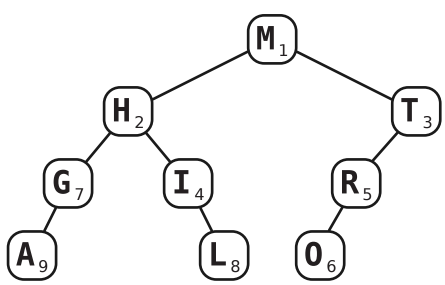
10 Randomized Binary Search Trees

In this lecture, we consider two randomized alternatives to balanced binary search tree structures such as AVL trees, red-black trees, B-trees, or splay trees, which are arguably simpler than any of these deterministic structures.

10.1 Treaps

10.1.1 Definitions

A *treap* is a binary tree in which every node has both a *search key* and a *priority*, where the inorder sequence of search keys is sorted and each node's priority is smaller than the priorities of its children.[1] In other words, a treap is simultaneously a binary search tree for the search keys and a (min-)heap for the priorities. In our examples, we will use letters for the search keys and numbers for the priorities.



A treap. Letters are search keys; numbers are priorities.

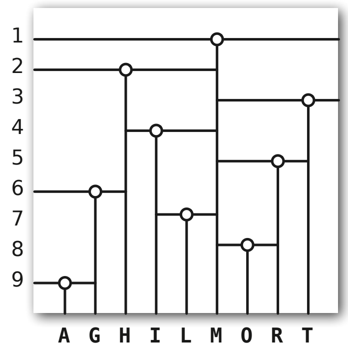
I'll assume from now on that all the keys and priorities are distinct. Under this assumption, we can easily prove by induction that the structure of a treap is completely determined by the search keys and priorities of its nodes. Since it's a heap, the node v with highest priority must be the root. Since it's also a binary search tree, any node u with $key(u) < key(v)$ must be in the left

[1]Sometimes I hate English. Normally, 'higher priority' means 'more important', but 'first priority' is also more important than 'second priority'. Maybe 'posteriority' would be better; one student suggested 'unimportance'.

subtree, and any node w with $key(w) > key(v)$ must be in the right subtree. Finally, since the subtrees are treaps, by induction, their structures are completely determined. The base case is the trivial empty treap.

Another way to describe the structure is that a treap is exactly the binary search tree that results by inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm. This characterization is also easy to prove by induction.

A third description interprets the keys and priorities as the coordinates of a set of points in the plane. The root corresponds to a T whose joint lies on the topmost point. The T splits the plane into three parts. The top part is (by definition) empty; the left and right parts are split recursively. This interpretation has some interesting applications in computational geometry, which (unfortunately) we won't have time to talk about.



A geometric interpretation of the same treap.

Treaps were first discovered by Jean Vuillemin in 1980, but he called them *Cartesian trees*.[2] The word 'treap' was first used by Edward McCreight around 1980 to describe a slightly different data structure, but he later switched to the more prosaic name *priority search trees*.[3] Treaps were rediscovered and used to build randomized search trees by Cecilia Aragon and Raimund Seidel in 1989.[4] A different kind of randomized binary search tree, which uses random rebalancing instead of random priorities, was later discovered and analyzed by Conrado Martínez and Salvador Roura in 1996.[5]

10.1.2 Treap Operations

The search algorithm is the usual one for binary search trees. The time for a successful search is proportional to the depth of the node. The time for an unsuccessful search is proportional to the depth of either its successor or its predecessor.

To insert a new node z, we start by using the standard binary search tree insertion algorithm to insert it at the bottom of the tree. At the point, the search keys still form a search tree, but the priorities may no longer form a heap. To fix the heap property, as long as z has smaller priority than its parent, perform a *rotation* at z, a local operation that decreases the depth of z by one

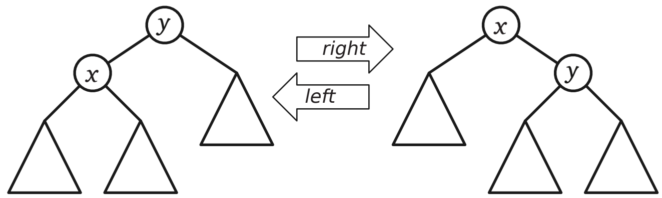
[2] J. Vuillemin, A unifying look at data structures. *Commun. ACM* 23:229–239, 1980.

[3] E. M. McCreight. Priority search trees. *SIAM J. Comput.* 14(2):257–276, 1985.

[4] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica* 16:464–497, 1996.

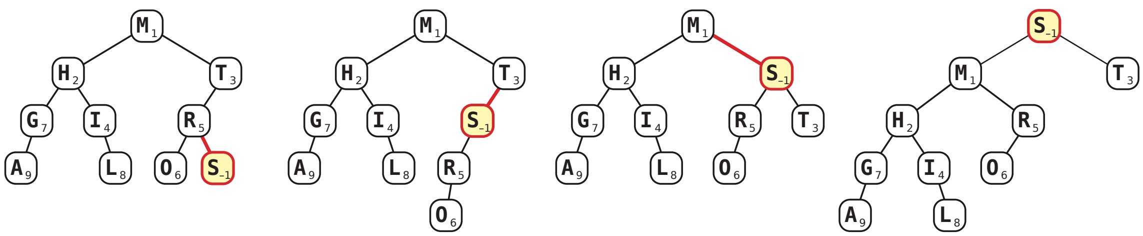
[5] C. Martínez and S. Roura. Randomized binary search trees. *J. ACM* 45(2):288-323, 1998. The results in this paper are virtually identical (including the constant factors!) to the corresponding results for treaps, although the analysis techniques are quite different.

and increases its parent's depth by one, while maintaining the search tree property. Rotations can be performed in constant time, since they only involve simple pointer manipulation.



A right rotation at x and a left rotation at y are inverses.

The overall time to insert z is proportional to the depth of z before the rotations—we have to walk down the treap to insert z, and then walk back up the treap doing rotations. Another way to say this is that the time to insert z is roughly twice the time to perform an unsuccessful search for $key(z)$.



Left to right: After inserting S with priority -1, rotate it up to fix the heap property.
Right to left: Before deleting S, rotate it down to make it a leaf.

To delete a node, we just run the insertion algorithm backward in time. Suppose we want to delete node z. As long as z is not a leaf, perform a rotation at the child of z with smaller priority. This moves z down a level and its smaller-priority child up a level. The choice of which child to rotate preserves the heap property everywhere except at z. When z becomes a leaf, chop it off.

We sometimes also want to *split* a treap T into two treaps $T_<$ and $T_>$ along some pivot key π, so that all the nodes in $T_<$ have keys less than π and all the nodes in $T_>$ have keys bigger then π. A simple way to do this is to insert a new node z with $key(z) = \pi$ and $priority(z) = -\infty$. After the insertion, the new node is the root of the treap. If we delete the root, the left and right sub-treaps are exactly the trees we want. The time to split at π is roughly twice the time to (unsuccessfully) search for π.

Similarly, we may want to *join* two treaps $T_<$ and $T_>$, where every node in $T_<$ has a smaller search key than any node in $T_>$, into one super-treap. Merging is just splitting in reverse—create a dummy root whose left sub-treap is $T_<$ and whose right sub-treap is $T_>$, rotate the dummy node down to a leaf, and then cut it off.

The cost of each of these operations is proportional to the depth of some node v in the treap.

- **Search:** A successful search for key k takes $O(depth(v))$ time, where v is the node with $key(v) = k$. For an unsuccessful search, let v^- be the inorder *predecessor* of k (the node whose key is just barely smaller than k), and let v^+ be the inorder *successor* of k (the node whose key is just barely larger than k). Since the last node examined by the binary search is either v^- or v^+, the time for an unsuccessful search is either $O(depth(v^+))$ or $O(depth(v^-))$.

- **Insert/Delete:** Inserting a new node with key k takes either $O(depth(v^+))$ time or $O(depth(v^-))$ time, where v^+ and v^- are the predecessor and successor of the new node. Deletion is just insertion in reverse.

- **Split/Join:** Splitting a treap at pivot value k takes either $O(depth(v^+))$ time or $O(depth(v^-))$ time, since it costs the same as inserting a new dummy root with search key k and priority $-\infty$. Merging is just splitting in reverse.

Since the depth of a node in a treap is $\Theta(n)$ in the worst case, each of these operations has a worst-case running time of $\Theta(n)$.

10.1.3 Random Priorities

A *randomized treap* is a treap in which the priorities are *independently and uniformly distributed continuous random variables*. That means that whenever we insert a new search key into the treap, we generate a random real number between (say) 0 and 1 and use that number as the priority of the new node. The only reason we're using real numbers is so that the probability of two nodes having the same priority is zero, since equal priorities make the analysis slightly messier. In practice, we could just choose random integers from a large range, like 0 to $2^{31}-1$, or random floating point numbers. Also, since the priorities are independent, each node is equally likely to have the smallest priority.

The cost of all the operations we discussed—search, insert, delete, split, join—is proportional to the depth of some node in the tree. Here we'll see that the *expected* depth of *any* node is $O(\log n)$, which implies that the expected running time for any of those operations is also $O(\log n)$.

Let x_k denote the node with the kth smallest search key. To simplify notation, let us write $\boldsymbol{i \uparrow k}$ (read "i above k") to mean that x_i is a proper ancestor of x_k. Since the depth of v is just the number of proper ancestors of v, we have the following identity:

$$depth(x_k) = \sum_{i=1}^{n} [i \uparrow k].$$

(Again, we're using Iverson bracket notation.) Now we can express the *expected* depth of a node in terms of these indicator variables as follows.

$$\mathrm{E}[depth(x_k)] = \sum_{i=1}^{n} E\big[[i \uparrow k]\big] = \sum_{i=1}^{n} \Pr[i \uparrow k]$$

(Just as in our analysis of matching nuts and bolts, we're using linearity of expectation and the fact that $\mathrm{E}[X] = \Pr[X = 1]$ for any zero-one variable X; in this case, $X = [i \uparrow k]$.) So to compute the expected depth of a node, we just have to compute the probability that some node is a proper ancestor of some other node.

Fortunately, we can do this easily once we prove a simple structural lemma. Let $X(i,k)$ denote either the subset of treap nodes $\{x_i, x_{i+1}, \ldots, x_k\}$ or the subset $\{x_k, x_{k+1}, \ldots, x_i\}$, depending on whether $i < k$ or $i > k$. The order of the arguments is unimportant; the subsets $X(i,k)$ and $X(k,i)$ are identical. The subset $X(1,n) = X(n,1)$ contains all n nodes in the treap.

Lemma 1. *For all $i \neq k$, we have $i \uparrow k$ if and only if x_i has the smallest priority among all nodes in $X(i,k)$.*

Proof: There are four cases to consider.

If x_i is the root, then $i \uparrow k$, and by definition, it has the smallest priority of *any* node in the treap, so it must have the smallest priority in $X(i,k)$.

On the other hand, if x_k is the root, then $k \uparrow i$, so $i \not\uparrow k$. Moreover, x_i does not have the smallest priority in $X(i,k)$ — x_k does.

On the gripping hand[6], suppose some other node x_j is the root. If x_i and x_k are in different subtrees, then either $i < j < k$ or $i > j > k$, so $x_j \in X(i,k)$. In this case, we have both $i \not\uparrow k$ and $k \not\uparrow i$, and x_i does not have the smallest priority in $X(i,k)$ — x_j does.

Finally, if x_i and x_k are in the same subtree, the lemma follows from the inductive hypothesis (or, if you prefer, the Recursion Fairy), because the subtree is a smaller treap. The empty treap is the trivial base case. □

Since each node in $X(i,k)$ is equally likely to have smallest priority, we immediately have the probability we wanted:

$$\Pr[i \uparrow k] = \frac{[i \neq k]}{|k-i|+1} = \begin{cases} \dfrac{1}{k-i+1} & \text{if } i < k \\ 0 & \text{if } i = k \\ \dfrac{1}{i-k+1} & \text{if } i > k \end{cases}$$

To compute the expected depth of a node x_k, we just plug this probability into our formula and grind through the algebra.

$$\begin{aligned} \mathrm{E}[\mathit{depth}(x_k)] = \sum_{i=1}^{n} \Pr[i \uparrow k] &= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\ &= \sum_{j=2}^{k} \frac{1}{j} + \sum_{i=2}^{n-k+1} \frac{1}{j} \\ &= H_k - 1 + H_{n-k+1} - 1 \\ &< \ln k + \ln(n-k+1) - 2 \\ &< 2\ln n - 2. \end{aligned}$$

In conclusion, every search, insertion, deletion, split, and join operation in an n-node randomized binary search tree takes $O(\log n)$ expected time.

Since a treap is exactly the binary tree that results when you insert the keys in order of increasing priority, a randomized treap is the result of inserting the keys in *random* order. So our analysis also automatically gives us the expected depth of any node in a binary tree built by random insertions (without using priorities).

10.1.4 Randomized Quicksort (Again!)

We've already seen two completely different ways of describing randomized quicksort. The first is the familiar recursive one: choose a random pivot, partition, and recurse. The second is a less familiar iterative version: repeatedly choose a new random pivot, partition whatever subset contains it, and continue. But there's a third way to describe randomized quicksort, this time in terms of binary search trees.

[6]See Larry Niven and Jerry Pournelle, *The Gripping Hand*, Pocket Books, 1994.

<u>RandomizedQuicksort:</u>
  $T \leftarrow$ an empty binary search tree
  insert the keys into T *in random order*
  output the inorder sequence of keys in T

Our treap analysis tells us is that this algorithm will run in $O(n \log n)$ expected time, since each key is inserted in $O(\log n)$ expected time.

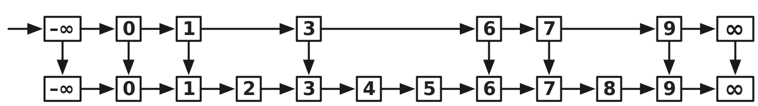
Why is this quicksort? Just like last time, all we've done is rearrange the order of the comparisons. Intuitively, the binary tree is just the recursion tree created by the normal version of quicksort. In the recursive formulation, we compare the initial pivot against everything else and then recurse. In the binary tree formulation, the first "pivot" becomes the root of the tree without any comparisons, but then later as each other key is inserted into the tree, it is compared against the root. Either way, the first pivot chosen is compared with everything else. The partition splits the remaining items into a left subarray and a right subarray; in the binary tree version, these are exactly the items that go into the left subtree and the right subtree. Since both algorithms define the same two subproblems, by induction, both algorithms perform the same comparisons.

We even saw the probability $1/(|k-i|+1)$ before, when we were talking about sorting nuts and bolts with a variant of randomized quicksort. In the more familiar setting of sorting an array of numbers, the probability that randomized quicksort compares the ith largest and kth largest elements is exactly $2/(|k-i|+1)$. The binary tree version of quicksort compares x_i and x_k if and only if $i \uparrow k$ or $k \uparrow i$, so the probabilities are exactly the same.

10.2 Skip Lists

Skip lists, which were first discovered by Bill Pugh in the late 1980's,[7] have many of the usual desirable properties of balanced binary search trees, but their structure is very different.

At a high level, a skip list is just a sorted linked list with some random shortcuts. To do a search in a normal singly-linked list of length n, we obviously need to look at n items in the worst case. To speed up this process, we can make a second-level list that contains roughly half the items from the original list. Specifically, for each item in the original list, we duplicate it with probability $1/2$. We then string together all the duplicates into a second sorted linked list, and add a pointer from each duplicate back to its original. Just to be safe, we also add sentinel nodes at the beginning and end of both lists.

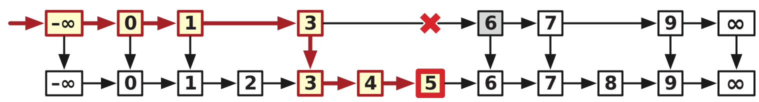


A linked list with some randomly-chosen shortcuts.

Now we can find a value x in this augmented structure using a two-stage algorithm. First, we scan for x in the shortcut list, starting at the $-\infty$ sentinel node. If we find x, we're done. Otherwise, we reach some value bigger than x and we know that x is not in the shortcut list. Let w be the largest item less than x in the shortcut list. In the second phase, we scan for x in the original list, starting from w. Again, if we reach a value bigger than x, we know that x is not in the data structure.

Since each node appears in the shortcut list with probability $1/2$, the expected number of nodes examined in the first phase is at most $n/2$. Only one of the nodes examined in the second

[7] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* 33(6):668–676, 1990.

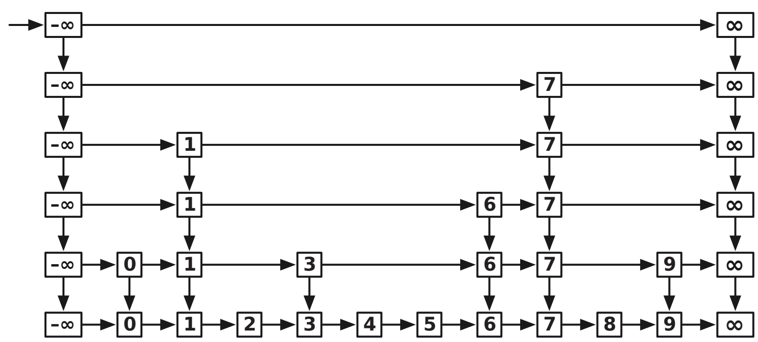


Searching for 5 in a list with shortcuts.

phase has a duplicate. The probability that any node is followed by k nodes without duplicates is 2^{-k}, so the expected number of nodes examined in the second phase is at most $1+\sum_{k\geq 0} 2^{-k} = 2$. Thus, by adding these random shortcuts, we've reduced the cost of a search from n to $n/2+2$, roughly a factor of two in savings.

10.2.1 Recursive Random Shortcuts

Now there's an obvious improvement—add shortcuts to the shortcuts, and repeat recursively. That's exactly how skip lists are constructed. For each node in the original list, we flip a coin over and over until we get tails. Each time we get heads, we make a duplicate of the node. The duplicates are stacked up in levels, and the nodes on each level are strung together into sorted linked lists. Each node v stores a search key ($key(v)$), a pointer to its next lower copy ($down(v)$), and a pointer to the next node in its level ($right(v)$).

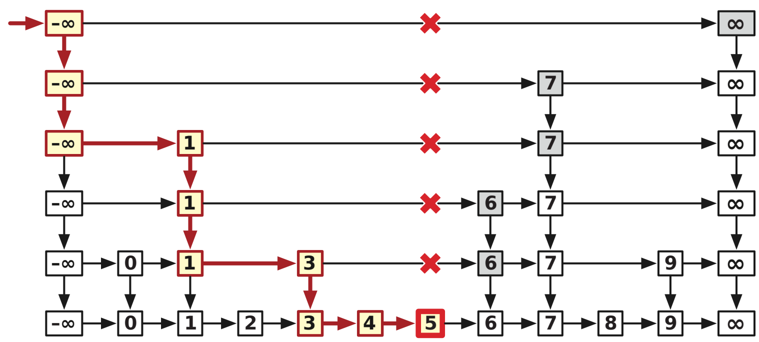


A skip list is a linked list with recursive random shortcuts.

The search algorithm for skip lists is very simple. Starting at the leftmost node L in the highest level, we scan through each level as far as we can without passing the target value x, and then proceed down to the next level. The search ends when we either reach a node with search key x or fail to find x on the lowest level.

```
SkipListFind(x, L):
  v ← L
  while (v ≠ Null and key(v) ≠ x)
      if key(right(v)) > x
          v ← down(v)
      else
          v ← right(v)
  return v
```

Intuitively, since each level of the skip lists has about half the number of nodes as the previous level, the total number of levels should be about $O(\log n)$. Similarly, each time we add another level of random shortcuts to the skip list, we cut the search time roughly in half, except for a constant overhead, so after $O(\log n)$ levels, we should have a search time of $O(\log n)$. Let's formalize each of these two intuitive observations.



Searching for 5 in a skip list.

10.2.2 Number of Levels

The actual values of the search keys don't affect the skip list analysis, so let's assume the keys are the integers 1 through n. Let $L(x)$ be the number of levels of the skip list that contain some search key x, not counting the bottom level. Each new copy of x is created with probability $1/2$ from the previous level, essentially by flipping a coin. We can compute the expected value of $L(x)$ recursively—with probability $1/2$, we flip tails and $L(x) = 0$; and with probability $1/2$, we flip heads, increase $L(x)$ by one, and recurse:

$$E[L(x)] = \frac{1}{2} \cdot 0 + \frac{1}{2}\big(1 + E[L(x)]\big)$$

Solving this equation gives us $\boldsymbol{E[L(x)] = 1}$.

In order to analyze the expected worst-case cost of a search, however, we need a bound on the *number of levels* $L = \max_x L(x)$. Unfortunately, we can't compute the average of a maximum the way we would compute the average of a sum. Instead, we derive a stronger result: ***The depth of a skip list storing n keys is $O(\log n)$ with high probability.*** "High probability" is a technical term that means the probability is at least $1 - 1/n^c$ for some constant $c \geq 1$; the hidden constant in the $O(\log n)$ bound could depend on c.

In order for a search key x to appear on level ℓ, it must have flipped ℓ heads in a row when it was inserted, so $\mathbf{Pr}[\boldsymbol{L(x) \geq \ell}] = \boldsymbol{2^{-\ell}}$. The skip list has at least ℓ levels if and only if $L(x) \geq \ell$ for at least one of the n search keys.

$$\Pr[L \geq \ell] = \Pr\Big[(L(1) \geq \ell) \vee (L(2) \geq \ell) \vee \cdots \vee (L(n) \geq \ell)\Big]$$

Using the *union bound* — $\Pr[A \vee B] \leq Pr[A] + \Pr[B]$ for any random events A and B — we can simplify this as follows:

$$\Pr[L \geq \ell] \;\leq\; \sum_{x=1}^{n} \Pr[L(x) \geq \ell] \;=\; n \cdot \Pr[L(x) \geq \ell] \;=\; \frac{n}{2^\ell}.$$

When $\ell \leq \lg n$, this bound is trivial. However, for any constant $c > 1$, we have a strong upper bound

$$\Pr[L \geq c \lg n] \leq \frac{1}{n^{c-1}}.$$

We conclude that ***with high probability, a skip list has $O(\log n)$ levels.***

This high-probability bound indirectly implies a bound on the *expected* number of levels. Some simple algebra gives us the following alternate definition for expectation:

$$\mathrm{E}[L] = \sum_{\ell \geq 0} \ell \cdot \Pr[L = \ell] = \sum_{\ell \geq 1} \Pr[L \geq \ell]$$

Clearly, if $\ell < \ell'$, then $\Pr[L(x) \geq \ell] > \Pr[L(x) \geq \ell']$. So we can derive an upper bound on the expected number of levels as follows:

$$\begin{aligned}
\mathrm{E}[L(x)] = \sum_{\ell \geq 1} \Pr[L \geq \ell] &= \sum_{\ell=1}^{\lg n} \Pr[L \geq \ell] \;+ \sum_{\ell \geq \lg n + 1} \Pr[L \geq \ell] \\
&\leq \sum_{\ell=1}^{\lg n} 1 \;+ \sum_{\ell \geq \lg n+1} \frac{n}{2^\ell} \\
&= \lg n + \sum_{i \geq 1} \frac{1}{2^i} && [i = \ell - \lg n] \\
&= \mathbf{\lg n + 2}
\end{aligned}$$

So in expectation, a skip list has *at most two* more levels than an ideal version where each level contains exactly half the nodes of the next level below.

10.2.3 Logarithmic Search Time

It's a little easier to analyze the cost of a search if we imagine running the algorithm backwards. SkipListFind takes the output from SkipListFind as input and traces back through the data structure to the upper left corner. Skip lists don't really have up and left pointers, but we'll pretend that they do so we don't have to write '$\mathit{down}(v) \to v$' or '$\mathit{right}(v) \to v$'.[8]

```
SkipListFind(v):
  while (v ≠ L)
      if up(v) exists
          v ← up(v)
      else
          v ← left(v)
```

Now for *every* node v in the skip list, $up(v)$ exists with probability $1/2$. So for purposes of analysis, SkipListFind is equivalent to the following algorithm:

```
FlipWalk(v):
  while (v ≠ L)
      if CoinFlip = Heads
      v ← up(v)
  else
      v ← left(v)
```

Obviously, the expected number of heads is exactly the same as the expected number of Tails. Thus, the expected running time of this algorithm is twice the expected number of upward jumps. Since we already know that the number of upward jumps is $O(\log n)$ with high probability, we can conclude that the worst-case search time is $O(\log n)$ with high probability (and therefore in expectation).

[8] Leonardo da Vinci wrote all his notes using mirror-writing, but not because he wanted to keep his discoveries secret. He just had really bad arthritis in his right hand!

Exercises

1. Prove that a treap is exactly the binary search tree that results from inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm.

2. Consider a treap T with n vertices. As in the notes, Identify nodes in T by the ranks of their search keys; thus, 'node 5' means the node with the 5th smallest search key. Let i, j, and k be integers such that $1 \le i \le j \le k \le n$.

   (a) The *left spine* of a binary tree is a path starting at the root and following only left-child pointers down to a leaf. What is the expected number of nodes in the left spine of T?

   (b) What is the expected number of leaves in T? *[Hint: What is the probability that node k is a leaf?]*

   (c) What is the expected number of nodes in T with two children?

   (d) What is the expected number of nodes in T with exactly one child?

   *(e) What is the expected number of nodes in T with exactly one *grandchild*?

   (f) Prove that the expected number of proper descendants of any node in a treap is exactly equal to the expected depth of that node.

   (g) What is the *exact* probability that node j is a common ancestor of node i and node k?

   (h) What is the *exact* expected length of the unique path from node i to node k in T?

3. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A ***heater*** is a priority search tree in which the *priorities* are given by the user, and the *search keys* are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is a sort of anti-treap.[9]

   The following problems consider an n-node heater T whose priorities are the integers from 1 to n. We identify nodes in T by their *priorities*; thus, 'node 5' means the node in T with *priority* 5. For example, the min-heap property implies that node 1 is the root of T. Finally, let i and j be integers with $1 \le i < j \le n$.

   (a) Prove that in a random permutation of the $(i+1)$-element set $\{1, 2, \ldots, i, j\}$, elements i and j are adjacent with probability $2/(i+1)$.

   (b) Prove that node i is an ancestor of node j with probability $2/(i+1)$. *[Hint: Use part (a)!]*

   (c) What is the probability that node i is a *descendant* of node j? *[Hint: Don't use part (a)!]*

   (d) What is the *exact* expected depth of node j?

   (e) Describe and analyze an algorithm to insert a new item into a heater. Express the expected running time of the algorithm in terms of the rank of the newly inserted item.

[9] There are those who think that life has nothing left to chance, a host of holy horrors to direct our aimless dance.

(f) Describe an algorithm to delete the minimum-priority item (the root) from an n-node heater. What is the expected running time of your algorithm?

*4. In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0,1]$. In practice, however, computers have access only to random *bits*. This problem asks you to analyze an implementation of treaps that takes this limitation into account.

Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1\,..\,\infty]$ of random bits, which is interpreted as the rational number

$$priority(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or "reveal") new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LargerPriority(v, w):
  for i ← 1 to ∞
      if i > ℓ_v
          ℓ_v ← i;  π_v[i] ← RandomBit
      if i > ℓ_w
          ℓ_w ← i;  π_w[i] ← RandomBit
      if π_v[i] > π_w[i]
          return v
      else if π_v[i] < π_w[i]
          return w
```

Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LargerPriority during these insertions.

(a) Prove that $E[L] = \Theta(n)$.

(b) Prove that $E[\ell_v] = \Theta(1)$ for any node v. *[Hint: This is equivalent to part (a). Why?]*

(c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. *[Hint: Why doesn't this contradict part (b)?]*

5. Prove the following basic facts about skip lists, where n is the number of keys.

(a) The expected number of nodes is $O(n)$.

(b) A new key can be inserted in $O(\log n)$ time with high probability.

(c) A key can be deleted in $O(\log n)$ time with high probability.

6. Suppose we are given two skip lists, one storing a set A of m keys the other storing a set B of n keys. Describe and analyze an algorithm to merge these into a single skip list storing the set $A \cup B$ in $O(n)$ expected time. Here we do *not* assume that every key in A is smaller than every key in B; the two sets maybe arbitrarily intermixed. *[Hint: Do the obvious thing.]*

*7. Any skip list $\mathcal{L}$ can be transformed into a binary search tree $T(\mathcal{L})$ as follows. The root of $T(\mathcal{L})$ is the leftmost node on the highest non-empty level of $\mathcal{L}$; the left and right subtrees are constructed recursively from the nodes to the left and to the right of the root. Let's call the resulting tree $T(\mathcal{L})$ a *skip list tree*.

   (a) Show that any search in $T(\mathcal{L})$ is no more expensive than the corresponding search in $\mathcal{L}$. (Searching in $T(\mathcal{L})$ could be *considerably* cheaper—why?)

   (b) Describe an algorithm to insert a new search key into a skip list tree in $O(\log n)$ expected time. Inserting key x into $T(\mathcal{L})$ should produce *exactly* the same tree as inserting x into $\mathcal{L}$ and then transforming $\mathcal{L}$ into a tree. *[Hint: You will need to maintain some additional information in the tree nodes.]*

   (c) Describe an algorithm to delete a search key from a skip list tree in $O(\log n)$ expected time. Again, deleting key x from $T(\mathcal{L})$ should produce *exactly* the same tree as deleting x from $\mathcal{L}$ and then transforming $\mathcal{L}$ into a tree.

8. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

   - MakeQueue: Return a new priority queue containing the empty set.
   - FindMin(Q): Return the smallest element of Q (if any).
   - DeleteMin(Q): Remove the smallest element in Q (if any).
   - Insert(Q, x): Insert element x into Q, if it is not already there.
   - DecreaseKey(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
   - Delete(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
   - Meld(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

   A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. Meld can be implemented using the following randomized algorithm:

```
MELD(Q1, Q2):
  if Q1 is empty return Q2
  if Q2 is empty return Q1
  if key(Q1) > key(Q2)
      swap Q1 ↔ Q2
  with probability 1/2
      left(Q1) ← MELD(left(Q1), Q2)
  else
      right(Q1) ← MELD(right(Q1), Q2)
  return Q1
```

   (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of Meld(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if each left/right choice is made with equal probability?]*

(b) Prove that MELD(Q_1, Q_2) runs in $O(\log n)$ time with high probability.

(c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

But, on the other hand, Uncle Abner said that the person that had took a bull by the tail once had learnt sixty or seventy times as much as a person that hadn't, and said a person that started in to carry a cat home by the tail was gitting knowledge that was always going to be useful to him, and warn't ever going to grow dim or doubtful.

— Mark Twain, *Tom Sawyer Abroad* (1894)

★11 Tail Inequalities

The simple recursive structure of skip lists made it relatively easy to derive an upper bound on the expected *worst-case* search time, by way of a stronger high-probability upper bound on the worst-case search time. We can prove similar results for treaps, but because of the more complex recursive structure, we need slightly more sophisticated probabilistic tools. These tools are usually called *tail inequalities*; intuitively, they bound the probability that a random variable with a bell-shaped distribution takes a value in the *tails* of the distribution, far away from the mean.

11.1 Markov's Inequality

Perhaps the simplest tail inequality was named after the Russian mathematician Andrey Markov; however, in strict accordance with Stigler's Law of Eponymy, it first appeared in the works of Markov's probability teacher, Pafnuty Chebyshev.[1]

Markov's Inequality. *Let X be a non-negative integer random variable. For any $t > 0$, we have* $\Pr[X \geq t] \leq \operatorname{E}[X]/t$.

Proof: The inequality follows from the definition of expectation by simple algebraic manipulation.

$$
\begin{aligned}
\operatorname{E}[X] &= \sum_{k=0}^{\infty} k \cdot \Pr[X = k] && \text{[definition of E}[X]\text{]} \\
&= \sum_{k=0}^{\infty} \Pr[X \geq k] && \text{[algebra]} \\
&\geq \sum_{k=0}^{t-1} \Pr[X \geq k] && \text{[since } t < \infty\text{]} \\
&\geq \sum_{k=0}^{t-1} \Pr[X \geq t] && \text{[since } k < t\text{]} \\
&= t \cdot \Pr[X \geq t] && \text{[algebra]}
\end{aligned}
$$

□

Unfortunately, the bounds that Markov's inequality implies (at least directly) are often very weak, even useless. (For example, Markov's inequality implies that with high probability, every node in an n-node treap has depth $O(n^2 \log n)$. Well, *duh!*) To get stronger bounds, we need to exploit some additional structure in our random variables.

[1]The closely related tail bound traditionally called Chebyshev's inequality was actually discovered by the French statistician Irénée-Jules Bienaymé, a friend and colleague of Chebyshev's.

11.2 Independence

A set of random variables $X_1, X_2, \dots, X_n$ are said to be *mutually independent* if and only if

$$\Pr\left[\bigwedge_{i=1}^{n}(X_i = x_i)\right] = \prod_{i=1}^{n}\Pr[X_i = x_i]$$

for all possible values $x_1, x_2, \dots, x_n$. For examples, different flips of the same fair coin are mutually independent, but the number of heads and the number of tails in a sequence of n coin flips are not independent (since they must add to n). Mutual independence of the X_i's implies that the expectation of the product of the X_i's is equal to the product of the expectations:

$$\mathrm{E}\left[\prod_{i=1}^{n} X_i\right] = \prod_{i=1}^{n}\mathrm{E}[X_i].$$

Moreover, if $X_1, X_2, \dots, X_n$ are independent, then for any function f, the random variables $f(X_1)$, $f(X_2), \dots, f(X_n)$ are also mutually independent.

— Discuss limited independence? —
— Add Chebychev and other moment inequalities? —

11.3 Chernoff Bounds

— Replace with Mihai's exponential-moment derivation! —

Suppose $X = \sum_{i=1}^{n} X_i$ is the sum of n mutually independent random *indicator* variables X_i. For each i, let $p_i = \Pr[X_i = 1]$, and let $\mu = \mathrm{E}[X] = \sum_i \mathrm{E}[X_i] = \sum_i p_i$.

Chernoff Bound (Upper Tail). $\boxed{\Pr[X > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}}$ *for any* $\delta > 0$.

Proof: The proof is fairly long, but it replies on just a few basic components: a clever substitution, Markov's inequality, the independence of the X_i's, The World's Most Useful Inequality $e^x > 1 + x$, a tiny bit of calculus, and lots of high-school algebra.

We start by introducing a variable t, whose role will become clear shortly.

$$Pr[X > (1+\delta)\mu] = \Pr[e^{tX} > e^{t(1+\delta)\mu}]$$

To cut down on the superscripts, I'll usually write $\exp(x)$ instead of e^x in the rest of the proof. Now apply Markov's inequality to the right side of this equation:

$$Pr[X > (1+\delta)\mu] < \frac{\mathrm{E}[\exp(tX)]}{\exp(t(1+\delta)\mu)}.$$

We can simplify the expectation on the right using the fact that the terms X_i are independent.

$$\mathrm{E}[\exp(tX)] = \mathrm{E}\left[\exp\left(t\sum_i X_i\right)\right] = \mathrm{E}\left[\prod_i \exp(tX_i)\right] = \prod_i \mathrm{E}[\exp(tX_i)]$$

We can bound the individual expectations $\mathrm{E}[\exp(tX_i)]$ using The World's Most Useful Inequality:

$$\mathrm{E}[\exp(tX_i)] = p_i e^t + (1 - p_i) = 1 + (e^t - 1)p_i < \exp\big((e^t - 1)p_i\big)$$

This inequality gives us a simple upper bound for $\mathrm{E}[e^{tX}]$:

$$\mathrm{E}[\exp(tX)] < \prod_i \exp((e^t - 1)p_i) < \exp\Big(\sum_i (e^t - 1)p_i\Big) = \exp((e^t - 1)\mu)$$

Substituting this back into our original fraction from Markov's inequality, we obtain

$$Pr[X > (1+\delta)\mu] < \frac{\mathrm{E}[\exp(tX)]}{\exp(t(1+\delta)\mu)} < \frac{\exp((e^t - 1)\mu)}{\exp(t(1+\delta)\mu)} = \big(\exp(e^t - 1 - t(1+\delta))\big)^{\mu}$$

Notice that this last inequality holds for *all* possible values of t. To obtain the final tail bound, we will choose t to make this bound as small as possible. To minimize $e^t - 1 - t - t\delta$, we take its derivative with respect to t and set it to zero:

$$\frac{d}{dt}(e^t - 1 - t(1+\delta)) = e^t - 1 - \delta = 0.$$

(And you thought calculus would never be useful!) This equation has just one solution $t = \ln(1+\delta)$. Plugging this back into our bound gives us

$$Pr[X > (1+\delta)\mu] < \big(\exp(\delta - (1+\delta)\ln(1+\delta))\big)^{\mu} = \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$$

And we're done! □

This form of the Chernoff bound can be a bit clumsy to use. A more complicated argument gives us the bound

$$\boxed{\Pr[X > (1+\delta)\mu] < e^{-\mu\delta^2/3} \text{ for any } 0 < \delta < 1.}$$

A similar argument gives us an inequality bounding the probability that X is significantly *smaller* than its expected value:

Chernoff Bound (Lower Tail). $\boxed{\Pr[X < (1-\delta)\mu] < \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} < e^{-\mu\delta^2/2}}$ *for any* $\delta > 0$.

11.4 Back to Treaps

In our analysis of randomized treaps, we wrote $i \uparrow k$ to indicate that the node with the ith smallest key ('node i') was a proper ancestor of the node with the kth smallest key ('node k'). We argued that

$$\Pr[i \uparrow k] = \frac{[i \neq k]}{|k - i| + 1},$$

and from this we concluded that the expected depth of node k is

$$\mathrm{E}[\mathrm{depth}(k)] = \sum_{i=1}^{n} \Pr[i \uparrow k] = H_k + H_{n-k} - 2 < 2\ln n.$$

To prove a worst-case expected bound on the depth of the tree, we need to argue that the *maximum* depth of any node is small. Chernoff bounds make this argument easy, once we establish that the relevant indicator variables are mutually independent.

Lemma 1. *For any index k, the $k-1$ random variables $[i \uparrow k]$ with $i < k$ are mutually independent. Similarly, for any index k, the $n-k$ random variables $[i \uparrow k]$ with $i > k$ are mutually independent.*

Proof: We explicitly consider only the first half of the lemma when $k = 1$, although the argument generalizes easily to other values of k. To simplify notation, let X_i denote the indicator variable $[i \uparrow 1]$. Fix $n-1$ arbitrary indicator values $x_2, x_3, \ldots, x_n$. We prove the lemma by induction on n, with the vacuous base case $n = 1$. The definition of conditional probability gives us

$$\begin{aligned}\Pr\left[\bigwedge_{i=2}^{n}(X_i = x_i)\right] &= \Pr\left[\bigwedge_{i=2}^{n-1}(X_i = x_i) \,\wedge\, X_n = x_n\right]\\ &= \Pr\left[\bigwedge_{i=2}^{n-1}(X_i = x_i) \,\middle|\, X_n = x_n\right] \cdot \Pr\big[X_n = x_n\big]\end{aligned}$$

Now recall that $X_n = 1$ (which means $1 \uparrow n$) if and only if node n has the smallest priority of all nodes. The other $n-2$ indicator variables X_i depend only on the order of the priorities of nodes 1 through $n-1$. There are exactly $(n-1)!$ permutations of the n priorities in which the nth priority is smallest, and each of these permutations is equally likely. Thus,

$$\Pr\left[\bigwedge_{i=2}^{n-1}(X_i = x_i) \,\middle|\, X_n = x_n\right] = \Pr\left[\bigwedge_{i=2}^{n-1}(X_i = x_i)\right]$$

The inductive hypothesis implies that the variables $X_2, \ldots, X_{n-1}$ are mutually independent, so

$$\Pr\left[\bigwedge_{i=2}^{n-1}(X_i = x_i)\right] = \prod_{i=2}^{n-1}\Pr[X_i = x_i].$$

We conclude that

$$\Pr\left[\bigwedge_{i=2}^{n}(X_i = x_i)\right] = \Pr\big[X_n = x_n\big] \cdot \prod_{i=2}^{n-1}\Pr[X_i = x_i] = \prod_{i=1}^{n-1}\Pr[X_i = x_i],$$

or in other words, that the indicator variables are mutually independent. □

Theorem 2. *The depth of a randomized treap with n nodes is $O(\log n)$ with high probability.*

Proof: First let's bound the probability that the depth of node k is at most $8\ln n$. There's nothing special about the constant 8 here; I'm being generous to make the analysis easier.

The depth is a sum of n indicator variables A_k^i, as i ranges from 1 to n. Our Observation allows us to partition these variables into two mutually independent subsets. Let $d_<(k) = \sum_{i<k}[i \uparrow k]$ and $d_>(k) = \sum_{i<k}[i \uparrow k]$, so that $depth(k) = d_<(k) + d_>(k)$. If $depth(k) > 8\ln n$, then either $d_<(k) > 4\ln n$ or $d_>(k) > 4\ln n$.

Chernoff's inequality, with $\mu = \mathrm{E}[d_<(k)] = H_k - 1 < \ln n$ and $\delta = 3$, bounds the probability that $d_<(k) > 4\ln n$ as follows.

$$\Pr[d_<(k) > 4\ln n] < \Pr[d_<(k) > 4\mu] < \left(\frac{e^3}{4^4}\right)^{\mu} < \left(\frac{e^3}{4^4}\right)^{\ln n} = n^{\ln(e^3/4^4)} = n^{3-4\ln 4} < \frac{1}{n^2}.$$

(The last step uses the fact that $4\ln 4 \approx 5.54518 > 5$.) The same analysis implies that $\Pr[d_>(k) > 4\ln n] < 1/n^2$. These inequalities imply the crude bound $\Pr[depth(k) > 4\ln n] < 2/n^2$.

Now consider the probability that the treap has depth greater than $10 \ln n$. Even though the distributions of different nodes' depths are *not* independent, we can conservatively bound the probability of failure as follows:

$$\Pr\left[\max_k depth(k) > 8\ln n\right] = \Pr\left[\bigwedge_{k=1}^{n} (depth(k) > 8\ln n)\right] \le \sum_{k=1}^{n} \Pr[depth(k) > 8\ln n] < \frac{2}{n}.$$

This argument implies more generally that for any constant c, the depth of the treap is greater than $c \ln n$ with probability at most $2/n^{c\ln c - c}$. We can make the failure probability an arbitrarily small polynomial by choosing c appropriately. $\square$

This lemma implies that any search, insertion, deletion, or merge operation on an n-node treap requires $O(\log n)$ time with high probability. In particular, the expected *worst-case* time for each of these operations is $O(\log n)$.

Exercises

1. Prove that for any integer k such that $1 < k < n$, the $n-1$ indicator variables $[i \uparrow k]$ with $i \ne k$ are *not* mutually independent. *[Hint: Consider the case $n = 3$.]*

2. Recall from Exercise 1 in the previous note that the expected number of descendants of any node in a treap is $O(\log n)$. Why doesn't the Chernoff-bound argument for depth imply that, with high probability, *every* node in a treap has $O(\log n)$ descendants? The conclusion is clearly bogus—Every treap has a node with n descendants!—but what's the hole in the argument?

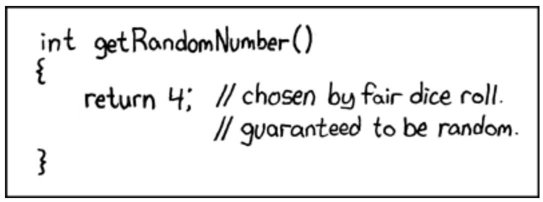
3. Recall from the previous lecture note that a *heater* is a sort of anti-treap, in which the priorities of the nodes are given, but their search keys are generated independently and uniformly from the unit interval $[0, 1]$.

   Prove that an n-node heater has depth $O(\log n)$ with high probability.

Insanity is repeating the same mistakes and expecting different results.
— *Narcotics Anonymous* (1981)

Calvin: *There! I finished our secret code!*
Hobbes: *Let's see.*
Calvin: *I assigned each letter a totally random number, so the code will be hard to crack. For letter "A", you write 3,004,572,688. "B" is 28,731,569½.*
Hobbes: *That's a good code all right.*
Calvin: *Now we just commit this to memory.*
Calvin: *Did you finish your map of our neighborhood?*
Hoobes: *Not yet. How many bricks does the front walk have?*

— Bill Watterson, "Calvin and Hobbes" (August 23, 1990)



[RFC 1149.5 specifies 4 as the standard IEEE-vetted random number.]

— Randall Munroe, *xkcd* (http://xkcd.com/221/)

12 Hash Tables

12.1 Introduction

A *hash table* is a data structure for storing a set of items, so that we can quickly determine whether an item is or is not in the set. The basic idea is to pick a *hash function* h that maps every possible item x to a small integer $h(x)$. Then we store x in slot $h(x)$ in an array. The array is the hash table.

Let's be a little more specific. We want to store a set of n items. Each item is an element of a fixed set $\mathcal{U}$ called the *universe*; we use u to denote the size of the universe, which is just the number of items in $\mathcal{U}$. A hash table is an array $T[1..m]$, where m is another positive integer, which we call the *table size*. Typically, m is much smaller than u. A *hash function* is any function of the form

$$h: \mathcal{U} \to \{0, 1, \ldots, m-1\},$$

mapping each possible item in $\mathcal{U}$ to a slot in the hash table. We say that an item x *hashes* to the slot $T[h(x)]$.

Of course, if $u = m$, then we can always just use the trivial hash function $h(x) = x$; n other words, we can use the item itself as the index into the table. This is called a *direct access table*, or more commonly, an *array*. In most applications, though, this approach requires much more space than we can reasonably allocate; on the other hand, we rarely need need to store more than a tiny fraction of $\mathcal{U}$. Ideally, the table size m should be roughly equal to the number n of items we actually want to store.

The downside of using a smaller table is that we must deal with *collisions*. We say that two items x and y *collide* if their hash values are equal: $h(x) = h(y)$. We are now left with two different (but interacting) design decisions. First, how to we choose a hash function h that can be evaluated quickly and that keeps the number of collisions as small as possible? Second, when collisions do occur, how do we deal with them?

12.2 The Importance of Being Random

If we already knew the precise data set that would be stored in our hash table, it is possible (but not particularly easy) to find a *perfect* hash function that avoids collisions entirely. Unfortunately, for most applications of hashing, we don't know what the user will put into the table. Thus, it is impossible *even in principle* to devise a perfect hash function in advance; no matter what hash function we choose, some pair of items from $\mathcal{U}$ *will* collide. Worse, for any fixed hash function, there is a subset of at least $|U|/m$ items that all hash to the same location. If our input data happens to come from such a subset, either by chance or malicious intent, our code will come to a grinding halt. This is a real security issue with core Internet routers, for example; every router on the Internet backbone survives millions of attacks per day, including timing attacks, from malicious agents.

The *only* way to provably avoid this worst-case behavior is to choose our hash functions *randomly*. Specifically, we will fix a set $\mathcal{H}$ of functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$, and then at run time, we choose our hash function randomly from the set $\mathcal{H}$ according to some fixed distribution. Different sets $\mathcal{H}$ and different distributions over that set imply different theoretical guarantees. Screw this into your brain:

Input data is *not* random!
So good hash functions *must be* random!

In particular, the simple deterministic hash function $h(x) = x \bmod m$, which is often taught and recommended under the name "the division method", is ***utterly stupid***. Many textbooks correctly observe that this hash function is bad when m is a power of 2, because then $h(x)$ is just the low-order bits of m, but then they bizarrely recommend making m prime to avoid such obvious collisions. But even when m is prime, any pair of items whose difference is an integer multiple of m collide with absolute certainty; for all integers a and x, we have $h(x + am) = h(x)$. Why would anyone use a hash function where they *know* certain pairs of keys *always* collide? Sheesh!

12.3 ...But Not Too Random

Most theoretical analysis of hashing assumes ***ideal random*** hash functions. Ideal randomness means that the hash function is chosen ***uniformly*** at random from the set of ***all*** functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$. Intuitively, for each new item x, we roll a new m-sided die to determine the hash value $h(x)$. Ideal randomness is a clean theoretical model, which provides the strongest possible theoretical guarantees.

Unfortunately, ideal random hash functions are a theoretical fantasy; evaluating such a function would require recording values in a separate data structure which we could access using the items in our set, which is exactly what hash tables are for! So instead, we look for families of hash functions with *just enough* randomness to guarantee good performance. Fortunately, most hashing analysis does not actually *require* ideal random hash functions, but only some weaker consequences of ideal randomness.

One property of ideal random hash functions that seems intuitively useful is ***uniformity***. A family $\mathcal{H}$ of hash functions is uniform if choosing a hash function uniformly at random from $\mathcal{H}$ makes every hash value equally likely for every item in the universe:

$$\boxed{\textbf{Uniform:}\quad \Pr_{h\in\mathcal{H}}\big[h(x)=i\big]=\frac{1}{m}\quad \text{for all } x \text{ and all } i}$$

We emphasize that this condition must hold for *every* item $x \in \mathcal{U}$ and *every* index i. Only the hash function h is random.

In fact, despite its intuitive appeal, uniformity is not terribly important or useful by itself. Consider the family $\mathcal{K}$ of *constant* hash functions defined as follows. For each integer a between 0 and $m-1$, let $const_a$ denote the constant function $const_a(x) = a$ for all x, and let $\mathcal{K} = \{const_a \mid 0 \le a \le m-1\}$ be the set of all such functions. It is easy to see that the set $\mathcal{K}$ is both perfectly uniform and utterly useless!

A much more important goal is to minimize the number of collisions. A family of hash functions is ***universal*** if, for any two items in the universe, the probability of collision is as small as possible:

$$\boxed{\textbf{Universal:}\quad \Pr_{h\in\mathcal{H}}\big[h(x)=h(y)\big]=\frac{1}{m}\quad \text{for all } x\neq y}$$

(Trivially, if $x = y$, then $\Pr[h(x) = h(y)] = 1$!) Again, we emphasize that this equation must hold for *every* pair of distinct items; only the function h is random. The family of constant functions is uniform but not universal; on the other hand, universal hash families are not necessarily uniform.[1]

Most elementary hashing analysis requires a weaker versions of universality. A family of hash functions is ***near-universal*** if the probability of collision is *close* to ideal:

$$\boxed{\textbf{Near-universal:}\quad \Pr_{h\in\mathcal{H}}\big[h(x)=h(y)\big]\le\frac{2}{m}\quad \text{for all } x\neq y}$$

There's nothing special about the number 2 in this definition; any other explicit constant will do.

On the other hand, some hashing analysis requires reasoning about larger sets of collisions. For any integer k, we say that a family of hash functions is ***strongly k-universal*** or ***k-uniform*** if for any sequence of k disjoint keys and any sequence of k hash values, the probability that each key maps to the corresponding hash value is $1/m^k$:

$$\boxed{\boldsymbol{k}\textbf{-uniform:}\quad \Pr\left[\bigwedge_{j=1}^{k} h(x_j)=i_j\right]=\frac{1}{m^k}\quad \text{for all distinct } x_1,\ldots,x_k \text{ and all } i_1,\ldots,i_k}$$

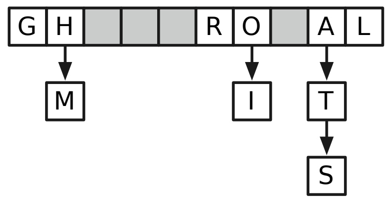
Ideal random hash functions are k-uniform for every positive integer k.

12.4 Chaining

One of the most common methods for resolving collisions in hash tables is called *chaining*. In a *chained* hash table, each entry $T[i]$ is not just a single item, but rather (a pointer to) a linked list of all the items that hash to $T[i]$. Let $\ell(x)$ denote the length of the list $T[h(x)]$. To see if

[1]Confusingly, universality is often called the ***uniform hashing assumption***, even though it is not an assumption that the hash function is uniform.

an item x is in the hash table, we scan the entire list $T[h(x)]$. The worst-case time required to search for x is $O(1)$ to compute $h(x)$ plus $O(1)$ for every element in $T[h(x)]$, or $O(1+\ell(x))$ overall. Inserting and deleting x also take $O(1+\ell(x))$ time.



A chained hash table with load factor 1.

Let's compute the expected value of $\ell(x)$ under this assumption; this will immediately imply a bound on the expected time to search for an item x. To be concrete, let's suppose that x is not already stored in the hash table. For all items x and y, we define the indicator variable

$$C_{x,y} = \big[h(x) = h(y)\big].$$

(In case you've forgotten the bracket notation, $C_{x,y} = 1$ if $h(x) = h(y)$ and $C_{x,y} = 0$ if $h(x) \neq h(y)$.) Since the length of $T[h(x)]$ is precisely equal to the number of items that collide with x, we have

$$\ell(x) = \sum_{y \in T} C_{x,y}.$$

Assuming h is chosen from a ***universal*** set of hash functions, we have

$$\mathrm{E}[C_{x,y}] = \Pr[C_{x,y} = 1] = \begin{cases} 1 & \text{if } x = y \\ 1/m & \text{otherwise} \end{cases}$$

Now we just have to grind through the definitions.

$$\mathrm{E}[\ell(x)] = \sum_{y \in T} \mathrm{E}[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m}$$

We call this fraction n/m the *load factor* of the hash table. Since the load factor shows up everywhere, we will give it its own symbol α.

$$\boxed{\alpha := \frac{n}{m}}$$

Similarly, if h is chosen from a ***near***-universal set of hash functions, then $\mathrm{E}[\ell(x)] \leq 2\alpha$. Thus, the expected time for an unsuccessful search in a chained hash table, using near-universal hashing, is $\Theta(1+\alpha)$. As long as the number of items n is only a constant factor bigger than the table size m, the search time is a constant. A similar analysis gives the same expected time bound (with a slightly smaller constant) for a successful search.

Obviously, linked lists are not the only data structure we could use to store the chains; any data structure that can store a set of items will work. For example, if the universe $\mathcal{U}$ has a total ordering, we can store each chain in a balanced binary search tree. This reduces the expected time for any search to $O(1+\log \ell(x))$, and under the simple uniform hashing assumption, the expected time for any search is $O(1+\log \alpha)$.

Another natural possibility is to work recursively! Specifically, for each $T[i]$, we maintain a hash table T_i containing all the items with hash value i. Collisions in those secondary tables are resolved recursively, by storing secondary overflow lists in tertiary hash tables, and so on. The resulting data structure is a tree of hash tables, whose leaves correspond to items that (at some level of the tree) are hashed without any collisions. If every hash table in this tree has size m, then the expected time for any search is $O(\log_m n)$. In particular, if we set $m = \sqrt{n}$, the expected time for any search is *constant*. On the other hand, there is no inherent reason to use the same hash table size everywhere; after all, hash tables deeper in the tree are storing fewer items.

Caveat Lector! The preceding analysis does *not* imply bounds on the expected *worst-case* search time is constant. The expected worst-case search time is $O(1 + L)$, where $L = \max_x \ell(x)$. Under the uniform hashing assumption, the maximum list size L is *very* likely to grow faster than any constant, unless the load factor α is *significantly* smaller than 1. For example, $\mathrm{E}[L] = \Theta(\log n / \log\log n)$ when $\alpha = 1$. We've stumbled on a powerful but counterintuitive fact about probability: When several individual items are distributed independently and uniformly at random, the resulting distribution is *not* uniform in the traditional sense! Later in this lecture, I'll describe how to achieve constant expected worst-case search time using secondary hash tables.

12.5 Multiplicative Hashing

Perhaps the simplest technique for near-universal hashing, first described by Carter and Wegman in the 1970s, is called ***multiplicative hashing***. I'll describe two variants of multiplicative hashing, one using modular arithmetic with prime numbers, the other using modular arithmetic with powers of two. In both variants, a hash function is specified by an integer parameter a, called a *salt*. The salt is chosen uniformly at random when the hash table is created and remains fixed for the entire lifetime of the table. All probabilities are defined with respect to the random choice of salt.

For any non-negative integer n, let $\boldsymbol{[n]}$ denote the n-element set $\{0, 1, \ldots, n-1\}$, and let $\boldsymbol{[n]^+}$ denote the $(n-1)$-element set $\{1, 2, \ldots, n-1\}$.

12.5.1 Prime multiplicative hashing

The first family of multiplicative hash function is defined in terms of a prime number $p > |\mathcal{U}|$. For any integer $a \in [p]^+$, define a function $\mathit{multp}_a \colon U \to [m]$ by setting

$$\boxed{\mathit{multp}_a(x) = (ax \bmod p) \bmod m}$$

and let

$$\mathcal{MP} := \left\{ \mathit{multp}_a \;\middle|\; a \in [p]^+ \right\}$$

denote the set of all such functions. Here, the integer a is the salt for the hash function multp_a. We claim that this family of hash functions is universal.

The use of prime modular arithmetic is motivated by the fact that *division* modulo prime numbers is well-defined.

Lemma 1. *For every integer $z \in [p]^+$, there is a unique integer $a \in [p]^+$ such that $az \bmod p = 1$.*

Proof: Let z be an arbitrary integer in $[p]^+$.

Suppose $az \bmod p = bz \bmod p$ for some integers $a, b \in [p]^+$. Then $(a-b)z \bmod p = 0$, which means $(a-b)z$ is divisible by p. Because p is prime, the inequality $1 \le z \le p-1$ implies that $a-b$ must be divisible by p. Similarly, the inequality $2-p < a-b < p-2$ implies that a and b must be equal. Thus, for each $z \in [p]^+$, there is *at most* one $a \in [p]^+$ such that $ax \bmod p = z$.

Similarly, suppose $az \bmod p = 0$ for some integer $a \in [p]^+$. Then because p is prime, either a or z is divisible by p, which is impossible.

We conclude that the set $\{az \bmod p \mid a \in [p]^+\}$ has $p-1$ distinct elements, all non-zero, and therefore is equal to $[p]^+$. In other words, multiplication by z defines a permutation of $[p]^+$. The lemma follows immediately. $\square$

For any integers $x, y \in \mathcal{U}$ and any salt $a \in [p]^+$, we have

$$\begin{aligned} multp_a(x) - multp_a(y) &= (ax \bmod p) \bmod m - (ay \bmod p) \bmod m \\ &= (ax \bmod p - ay \bmod p) \bmod m \\ &= ((ax - ay) \bmod p) \bmod m \\ &= (a(x-y) \bmod p) \bmod m \\ &= multp_a(x-y). \end{aligned}$$

Thus, we have a collision $multp_a(x) = multp_a(y)$ if and only if $multp_a(x-y) = 0$. Thus, to prove that $\mathcal{MP}$ is universal, it suffices to prove the following lemma.

Lemma 2. *For any $z \in [p]^+$, we have* $\Pr_a[\mathrm{multp}_a(z) = 0] \le 1/m$.

Proof: Fix an arbitrary integer $z \in [p]^+$. The previous lemma implies that for any integer $1 \le x \le p-1$, there is a unique integer a such that $(az \bmod p) = x$; specifically, $a = x \cdot z^{-1} \bmod p$. There are exactly $\lfloor (p-1)/m \rfloor$ integers k such that $1 \le km \le p-1$. Thus, there are exactly $\lfloor (p-1)/m \rfloor$ salts a such that $multp_a(z) = 0$. $\square$

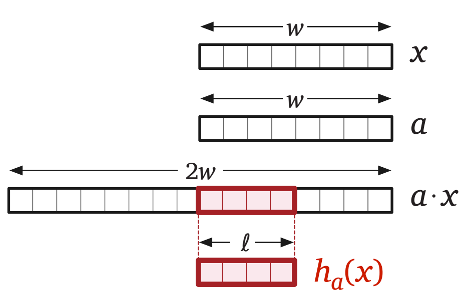
12.5.2 Binary multiplicative hashing

A slightly simpler variant of multiplicative hashing that avoids the need for large prime numbers was first analyzed by Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen in 1997. For this variant, we assume that $\mathcal{U} = [2^w]$ and that $m = 2^\ell$ for some integers w and ℓ. Thus, our goal is to hash w-bit integers ("words") to ℓ-bit integers ("labels").

For any *odd* integer $a \in [2^w]$, we define the hash function $multb_a \colon \mathcal{U} \to [m]$ as follows:

$$\boxed{multb_a(x) := \left\lfloor \frac{(a \cdot x) \bmod 2^w}{2^{w-\ell}} \right\rfloor}$$

Again, the odd integer a is the salt.



Binary multiplicative hashing.

If we think of any w-bit integer z as an array of bits $z[0..w-1]$, where $z[0]$ is the least significant bit, this function has an easy interpretation. The product $a \cdot x$ is $2w$ bits long; the hash value $multb_a(x)$ consists of the top ℓ bits of the bottom half:

$$multb_a(x) := (a \cdot x)[w-1..w-\ell]$$

Most programming languages automatically perform integer arithmetic modulo some power of two. If we are using an integer type with w bits, the function $multb_a(x)$ can be implemented by a single multiplication followed by a single right-shift. For example, in C:

```
#define hash(a,x) ((a)*(x) >> (WORDSIZE-HASHBITS))
```

Now we claim that the family $\mathcal{MB} := \{multb_a \mid a \text{ is odd}\}$ of all such functions is near-universal. To prove this claim, we again need to argue that division is well-defined, at least for a large subset of possible words. Let W denote the set of odd integers in $[2^w]$.

Lemma 3. *For any integers $x, z \in W$, there is exactly one integer $a \in W$ such that $ax \bmod 2^w = z$.*

Proof: Fix an integer $x \in W$. Suppose $ax \bmod 2^w = bx \bmod 2^w$ for some integers $a, b \in W$. Then $(b-a)x \bmod 2^w = 0$, which means $x(b-a)$ is divisible by 2^w. Because x is odd, $b-a$ must be divisible by 2^w. But $-2^w < b-a < 2^w$, so a and b must be equal. Thus, for each $z \in W$, there is *at most* one $a \in W$ such that $ax \bmod 2^w = z$. In other words, the function $f_x : W \to W$ defined by $f_x(a) := ax \bmod 2^w$ is injective. Every injective function from a finite set to itself is a bijection. □

Lemma 4. *$\mathcal{MB}$ is near-universal.*

Proof: Fix two distinct words $x, y \in \mathcal{U}$ such that $x < y$. If $multb_a(x) = multb_a(y)$, then the top ℓ bits of $a(y-x) \bmod 2^w$ are either all 0s (if $ax \bmod 2^w \le ay \bmod 2^w$) or all 1s (otherwise). Equivalently, if $multb_a(x) = multb_a(y)$, then either $multb_a(y-x) = 0$ or $multb_a(y-x) = m-1$. Thus,

$$\Pr[multb_a(x) = multb_a(y)] \;\le\; \Pr[multb_a(y-x) = 0] \;+\; \Pr[multb_a(y-x) = m-1].$$

We separately bound the terms on the right side of this inequality.

Because $x \ne y$, we can write $(y-x) \bmod 2^w = q2^r$ for some odd integer q and some integer $0 \le r \le w-1$. The previous lemma implies that $aq \bmod 2^w$ consists of $w-1$ random bits followed by a 1. Thus, $aq2^r \bmod 2^w$ consists of $w-r-1$ random bits, followed by a 1, followed by r 0s. There are three cases to consider:

- If $r < w-\ell$, then $multb_a(y-x)$ consists of ℓ random bits, so
$$\Pr[multb_a(y-x) = 0] \;=\; \Pr[multb_a(y-x) = m-1] \;=\; 1/2^\ell.$$
- If $r = w-\ell$, then $multb_a(y-x)$ consists of $\ell-1$ random bits followed by a 1, so
$$\Pr[multb_a(y-x) = 0] = 0 \quad \text{and} \quad \Pr[multb_a(y-x) = m-1] = 2/2^\ell.$$
- Finally, if $r < w-\ell$, then $multb_a(y-x)$ consists of zero or more random bits, followed by a 1, followed by one or more 0s, so
$$\Pr[multb_a(y-x) = 0] \;=\; \Pr[multb_a(y-x) = m-1] \;=\; 0.$$

In all cases, we have $\Pr[multb_a(x) = multb_a(y)] \le 2/2^\ell$, as required. □

*12.6 High Probability Bounds: Balls and Bins

Although any particular search in a chained hash tables requires only constant expected time, but what about the *worst* search time? Assuming that we are using ***ideal random*** hash functions, this question is equivalent to the following more abstract problem. Suppose we toss n balls independently and uniformly at random into one of n bins. Can we say anything about the number of balls in the fullest bin?

Lemma 5. *If n balls are thrown independently and uniformly into n bins, then with high probability, the fullest bin contains $O(\log n/\log\log n)$ balls.*

Proof: Let X_j denote the number of balls in bin j, and let $\hat{X} = \max_j X_j$ be the maximum number of balls in any bin. Clearly, $\mathrm{E}[X_j] = 1$ for all j.

Now consider the probability that bin j contains at least k balls. There are $\binom{n}{k}$ choices for those k balls; each chosen ball has probability $1/n$ of landing in bin j. Thus,

$$\Pr[X_j \geq k] \;=\; \binom{n}{k}\left(\frac{1}{n}\right)^k \;\leq\; \frac{n^k}{k!}\left(\frac{1}{n}\right)^k \;=\; \frac{1}{k!}$$

Setting $k = 2c\lg n/\lg\lg n$, we have

$$k! \geq k^{k/2} = \left(\frac{2c\lg n}{\lg\lg n}\right)^{2c\lg n/\lg\lg n} \geq \left(\sqrt{\lg n}\right)^{2c\lg n/\lg\lg n} = 2^{c\lg n} = n^c,$$

which implies that

$$\Pr\left[X_j \geq \frac{2c\lg n}{\lg\lg n}\right] < \frac{1}{n^c}.$$

This probability bound holds for every bin j. Thus, by the union bound, we conclude that

$$\Pr\left[\max_j X_j > \frac{2c\lg n}{\lg\lg n}\right] \;=\; \Pr\left[X_j > \frac{2c\lg n}{\lg\lg n} \text{ for all } j\right] \;\leq\; \sum_{j=1}^{n}\Pr\left[X_j > \frac{2c\lg n}{\lg\lg n}\right] \;<\; \frac{1}{n^{c-1}}. \qquad \square$$

A somewhat more complicated argument implies that if we throw n balls randomly into n bins, then with high probability, the most popular bin contains at least $\Omega(\log n/\log\log n)$ balls.

However, if we make the hash table large enough, we can expect every ball to land in its own bin. Suppose there are m bins. Let C_{ij} be the indicator variable that equals 1 if and only if $i \neq j$ and ball i and ball j land in the same bin, and let $C = \sum_{i<j} C_{ij}$ be the total number of pairwise collisions. Since the balls are thrown uniformly at random, the probability of a collision is exactly $1/m$, so $\mathrm{E}[C] = \binom{n}{2}/m$. In particular, if $m = n^2$, the expected number of collisions is less than $1/2$.

To get a high probability bound, let X_j denote the number of balls in bin j, as in the previous proof. We can easily bound the probability that bin j is empty, by taking the two most significant terms in a binomial expansion:

$$\Pr[X_j = 0] \;=\; \left(1 - \frac{1}{m}\right)^n \;=\; \sum_{i=1}^{n}\binom{n}{i}\left(\frac{-1}{m}\right)^i \;=\; 1 - \frac{n}{m} + \Theta\left(\frac{n^2}{m^2}\right) \;>\; 1 - \frac{n}{m}$$

We can similarly bound the probability that bin j contains exactly one ball:

$$\Pr[X_j = 1] \;=\; n\cdot\frac{1}{m}\left(1 - \frac{1}{m}\right)^{n-1} \;=\; \frac{n}{m}\left(1 - \frac{n-1}{m} + \Theta\left(\frac{n^2}{m^2}\right)\right) \;>\; \frac{n}{m} - \frac{n(n-1)}{m^2}$$

It follows immediately that $\Pr[X_j > 1] < n(n-1)/m^2$. The union bound now implies that $\Pr[\hat{X} > 1] < n(n-1)/m$. If we set $m = n^{2+\varepsilon}$ for any constant $\varepsilon > 0$, then the probability that no bin contains more than one ball is at least $1 - 1/n^{\varepsilon}$.

Lemma 6. *For any $\varepsilon > 0$, if n balls are thrown independently and uniformly into $n^{2+\varepsilon}$ bins, then with high probability, no bin contains more than one ball.*

We can give a slightly weaker version of this lemma that assumes only near-universal hashing. Suppose we hash n items into a table of size m. Linearity of expectation implies that the expected number of pairwise collisions is

$$\sum_{x<y} \Pr[h(x) = h(y)] \leq \binom{n}{2}\frac{2}{m} = \frac{n(n-1)}{m}.$$

In particular, if we set $m = cn^2$, the expected number of collisions is less than $1/c$, which implies that the probability of even a single collision is less than $1/c$.

12.7 Perfect Hashing

So far we are faced with two alternatives. If we use a small hash table to keep the space usage down, even if we use ideal random hash functions, the resulting worst-case expected search time is $\Theta(\log n/\log\log n)$ with high probability, which is not much better than a binary search tree. On the other hand, we can get constant worst-case search time, at least in expectation, by using a table of roughly quadratic size, but that seems unduly wasteful.

Fortunately, there is a fairly simple way to combine these two ideas to get a data structure of linear expected size, whose expected worst-case search time is constant. At the top level, we use a hash table of size $m = n$, but instead of linked lists, we use secondary hash tables to resolve collisions. Specifically, the jth secondary hash table has size $2n_j^2$, where n_j is the number of items whose primary hash value is j. Our earlier analysis implies that with probability at least $1/2$, the secondary hash table has no collisions at all, so the worst-case search time in any secondary hash table is $O(1)$. (If we discover a collision in some secondary hash table, we can simply rebuild that table with a new near-universal hash function.)

Although this data structure apparently needs significantly more memory for each secondary structure, the overall increase in space is insignificant, at least in expectation.

Lemma 7. *Assuming near-universal hashing, we have* $\mathrm{E}[\sum_i n_i^2] < 3n$.

Proof: let $h(x)$ denote the position of x in the primary hash table. We rewrite $\sum_i \mathrm{E}[n_i^2]$ in terms of the indicator variables $[h(x) = i]$ as follows. The first equation uses the definition of n_i; the rest is just routine algebra.

$$
\begin{aligned}
\sum_i n_i^2 &= \sum_i \left(\sum_x [h(x)=i]\right)^2 \\
&= \sum_i \left(\sum_x \sum_y [h(x)=i][h(y)=i]\right) \\
&= \sum_i \left(\sum_x [h(x)=i]^2 \;+\; 2\sum_{x<y} [h(x)=i][h(y)=i]\right) \\
&= \sum_x \sum_i [h(x)=i]^2 \;+\; 2\sum_{x<y}\sum_i [h(x)=i][h(y)=i] \\
&= \sum_x \sum_i [h(x)=i] \;+\; 2\sum_{x<y} [h(x)=h(y)]
\end{aligned}
$$

The first sum is equal to n, because each item x hashes to exactly one index i, and the second sum is just the number of pairwise collisions. Linearity of expectation immediately implies that

$$
\mathrm{E}\left[\sum_i n_i^2\right] \;=\; n + 2\sum_{x<y} \Pr[h(x)=h(y)] \;\le\; n + 2\cdot\frac{n(n-1)}{2}\cdot\frac{2}{n} \;=\; 3n-2. \qquad \square
$$

This lemma immediately implies that the expected size of our two-level hash table is $O(n)$. By our earlier analysis, the expected worst-case search time is $O(1)$.

12.8 Open Addressing

Another method used to resolve collisions in hash tables is called *open addressing*. Here, rather than building secondary data structures, we resolve collisions by looking elsewhere in the table. Specifically, we have a sequence of hash functions $\langle h_0, h_1, h_2, \dots, h_{m-1}\rangle$, such that for any item x, the *probe sequence* $\langle h_0(x), h_1(x), \dots, h_{m-1}(x)\rangle$ is a permutation of $\langle 0, 1, 2, \dots, m-1\rangle$. In other words, different hash functions in the sequence always map x to different locations in the hash table.

We search for x using the following algorithm, which returns the array index i if $T[i] = x$, 'absent' if x is not in the table but there is an empty slot, and 'full' if x is not in the table and there no no empty slots.

```
OpenAddressSearch(x):
  for i ← 0 to m − 1
      if T[h_i(x)] = x
          return h_i(x)
      else if T[h_i(x)] = ∅
          return 'absent'
  return 'full'
```

The algorithm for inserting a new item into the table is similar; only the second-to-last line is changed to $T[h_i(x)] \leftarrow x$. Notice that for an open-addressed hash table, the load factor is never bigger than 1.

Just as with chaining, we'd like to pretend that the sequence of hash values is truly random, for purposes of analysis. Specifically, most open-addressed hashing analysis uses the following assumption, which is impossible to enforce in practice, but leads to reasonably predictive results for most applications.

Strong uniform hashing assumption:

For any item x, the probe sequence $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x)\rangle$ is equally likely to be any permutation of the set $\{0, 1, 2, \ldots, m-1\}$.

Let's compute the expected time for an unsuccessful search in light of this assupmtion. Suppose there are currently n elements in the hash table. The strong uniform hashing assumption has two important consequences:

- **Uniformity:** Each hash value $h_i(x)$ is equally likely to be any integer in the set $\{0, 1, 2, \ldots, m-1\}$.
- **Indpendence:** If we ignore the first probe, the remaining probe sequence $\langle h_1(x), h_2(x), \ldots, h_{m-1}(x)\rangle$ is equally likely to be any permutation of the smaller set $\{0, 1, 2, \ldots, m-1\} \setminus \{h_0(x)\}$.

The first sentence implies that the probability that $T[h_0(x)]$ is occupied is exactly n/m. The second sentence implies that if $T[h_0(x)]$ is occupied, *our search algorithm recursively searches the rest of the hash table!* Since the algorithm will never again probe $T[h_0(x)]$, for purposes of analysis, we might as well pretend that slot in the table no longer exists. Thus, we get the following recurrence for the expected number of probes, as a function of m and n:

$$\mathrm{E}[T(m,n)] = 1 + \frac{n}{m}\,\mathrm{E}[T(m-1,n-1)].$$

The trivial base case is $T(m,0) = 1$; if there's nothing in the hash table, the first probe always hits an empty slot. We can now easily prove by induction that $\mathbf{E}[\boldsymbol{T(m,n)}] \le \boldsymbol{m/(m-n)}$:

$$\begin{aligned}
\mathrm{E}[T(m,n)] &= 1 + \frac{n}{m}\,\mathrm{E}[T(m-1,n-1)] \\
&\le 1 + \frac{n}{m}\cdot\frac{m-1}{m-n} && \text{[induction hypothesis]} \\
&< 1 + \frac{n}{m}\cdot\frac{m}{m-n} && [m-1 < m] \\
&= \frac{m}{m-n}\ \checkmark && \text{[algebra]}
\end{aligned}$$

Rewriting this in terms of the load factor $\alpha = n/m$, we get $\mathbf{E}[\boldsymbol{T(m,n)}] \le \mathbf{1}/(\mathbf{1}-\boldsymbol{\alpha})$. In other words, the expected time for an unsuccessful search is $O(1)$, unless the hash table is almost completely full.

12.9 Linear and Binary Probing

In practice, however, we can't generate ideal random probe sequences, so we must rely on a simpler probing scheme to resolve collisions. Perhaps the simplest scheme is ***linear probing***—use a single hash function $h(x)$ and define

$$\boxed{h_i(x) := (h(x) + i) \bmod m}$$

This strategy has several advantages, in addition to its obvious simplicity. First, because the probing strategy visits consecutive entries in the has table, linear probing exhibits better cache performance than other strategies. Second, as long as the load factor is strictly less than 1, the expected length of any probe sequence is provably constant; moreover, this performance is guaranteed even for hash functions with limited independence. On the other hand, the number or probes grows quickly as the load factor approaches 1, because the occupied cells in the hash table tend to cluster together. On the gripping hand, this clustering is arguably an *advantage* of linear probing, since any access to the hash table loads several nearby entries into the cache.

A simple variant of linear probing called ***binary probing*** is slightly easier to analyze. Assume that $m = 2^\ell$ for some integer ℓ (in a binary multiplicative hashing), and define

$$\boxed{h_i(x) := h(x) \oplus i}$$

where $\oplus$ denotes bitwise exclusive-or. This variant of linear probing has slightly better cache performance, because cache lines (and disk pages) usually cover address ranges of the form $[r2^k \mathrel{..} (r+1)2^k - 1]$; assuming the hash table is aligned in memory correctly, binary probing will scan one entire cache line before loading the next one.

Several more complex probing strategies have been proposed in the literature. Two of the most common are ***quadratic probing***, where we use a single hash function h and set $h_i(x) := (h(x) + i^2) \bmod m$, and ***double hashing***, where we use two hash functions h and h' and set $h_i(x) := (h(x) + i \cdot h'(x)) \bmod m$. These methods have some theoretical advantages over linear and binary probing, but they are not as efficient in practice, primarily due to cache effects.

*12.10 Analysis of Binary Probing

Lemma 8. *In a hash table of size $m = 2^\ell$ containing $n \le m/4$ keys, built using binary probing, the expected time for any search is $O(1)$,* **assuming ideal random hashing**.

Proof: The hash table is an array $H[0 \mathrel{..} m-1]$. For each integer k between 0 and ℓ, we partition H into $m/2^k$ ***level-k blocks*** of length 2^k; each level-k block has the form $H[c2^k \mathrel{..} (c+1)2^k - 1]$ for some integer c. Each level-k block contains exactly two level-$(k-1)$ blocks; thus, the blocks implicitly define a complete binary tree of depth ℓ.

Now suppose we want to search for a key x. For any integer k, let $B_k(x)$ denote the range of indices for the level-k block containing $H[h(x)]$:

$$B_k(x) = \left[2^k \lfloor h(x)/2^k \rfloor \mathrel{..} 2^k \lfloor h(x)/2^k \rfloor + 2^k - 1\right]$$

Similarly, let $B'_k(x)$ denote the sibling of $B_k(x)$ in the block tree; that is, $B'_k(x) = B_{k+1}(x) \setminus B_k(x)$. We refer to each $B_k(x)$ as an ***ancestor*** of x and each $B'_k(x)$ as an ***uncle*** of x. The proper ancestors of any uncle of x are also proper ancestors of x.

The binary probing algorithm can be recast conservatively as follows:

```
BinaryProbe(x):
  if H[h(x)] = x
      return True
  if H[h(x)] is empty
      return False
  for k = 0 to ℓ − 1
      for each index j in B'_k(x)
          if H[j] = x
              return True
          if H[j] is empty
              return False
```

For purposes of analysis, suppose the target item x is not in the table. (The time to search for an item that is in the table can only be faster.) Then the expected running time of BinaryProbe(x) can be expressed as follows:

$$\mathrm{E}[T(x)] \le \sum_{k=0}^{\ell-1} O(2^k) \cdot \Pr[B'_k(x) \text{ is full}].$$

Assuming ideal random hashing, all blocks at the same level have equal probability of being full. Let F_k denote the probability that a fixed level-k block is full. Then we have

$$\mathrm{E}[T(x)] \le \sum_{k=0}^{\ell-1} O(2^k) \cdot F_k.$$

Call a level-k block B ***popular*** if there are at least 2^k items y in the table such that $h(y) \in B$. Every popular block is full, but full blocks are not necessarily popular.

If block $B_k(x)$ is full but not popular, then $B_k(x)$ contains at least one item whose hash value is not in $B_k(x)$. Let y be the first such item inserted into the hash table. When y was inserted, some uncle block $B'_j(x) = B_j(y)$ with $j \ge k$ was already full. Let $B'_j(x)$ be the first uncle of $B_k(x)$ to become full. The only blocks that can overflow into $B_j(y)$ are its uncles, which are all either ancestors or uncles of $B_k(x)$. But when $B_j(y)$ became full, no other uncle of $B_k(x)$ was full. Moreover, $B_k(x)$ was not yet full (because there was still room for y), so no ancestor of $B_k(x)$ was full. It follows that $B'_j(x)$ is popular.

We conclude that if a block is full, then either that block or one of its uncles is popular. Thus, if we write P_k to denote the probability that a fixed level-k block is popular, we have

$$F_k \le 2P_k + \sum_{j>k} P_j.$$

We can crudely bound the probability P_k as follows. Each of the n items in the table hashes into a fixed level-k block with probability $2^k/m$; thus,

$$P_k = \binom{n}{2^k}\left(\frac{2^k}{m}\right)^{2^k} \le \frac{n^{2^k}}{(2^k)!}\frac{2^{k2^k}}{m^{2^k}} < \left(\frac{en}{m}\right)^{2^k}$$

(The last inequality uses a crude form of Stirling's approximation: $n! > n^n/e^n$.) Our assumption $n \le m/4$ implies the simpler inequality $P_k < (e/4)^{2^k}$. Because $e < 4$, it is easy to see that $\boldsymbol{P_k < 4^{-k}}$ for all sufficiently large k.

It follows that $F_k = O(4^{-k})$, which implies that the expected search time is at most $\sum_{k\ge0} O(2^k) \cdot O(4^{-k}) = \sum_{k\ge0} O(2^{-k}) = O(1)$. □

12.11 Cuckoo Hashing

★★★ Write this.

Exercises

1. Your boss wants you to find a *perfect* hash function for mapping a known set of n items into a table of size m. A hash function is *perfect* if there are *no* collisions; each of the n items

is mapped to a different slot in the hash table. Of course, a perfect hash function is only possible if $m \geq n$. (This is a different definition of "perfect" than the one considered in the lecture notes.) After cursing your algorithms instructor for not teaching you about (this kind of) perfect hashing, you decide to try something simple: repeatedly pick ideal random hash functions until you find one that happens to be perfect.

(a) Suppose you pick an ideal random hash function h. What is the *exact* expected number of collisions, as a function of n (the number of items) and m (the size of the table)? Don't worry about how to resolve collisions; just count them.

(b) What is the *exact* probability that a random hash function is perfect?

(c) What is the *exact* expected number of different random hash functions you have to test before you find a perfect hash function?

(d) What is the *exact* probability that none of the first N random hash functions you try is perfect?

(e) How many ideal random hash functions do you have to test to find a perfect hash function *with high probability*?

2. (a) Describe a set of hash functions that is uniform but not (near-)universal.

(b) Describe a set of hash functions that is universal but not (near-)uniform.

(c) Describe a set of hash functions that is universal but (near-)3-universal.

(d) A family of hash function is ***pairwise independent*** if knowing the hash value of any one item gives us absolutely no information about the hash value of any other item; more formally,

$$\Pr_{h \in \mathcal{H}}[h(x) = i \mid h(y) = j] \;=\; \Pr_{h \in \mathcal{H}}[h(x) = i]$$

or equivalently,

$$\Pr_{h \in \mathcal{H}}[(h(x) = i) \wedge (h(y) = j)] = \Pr_{h \in \mathcal{H}}[h(x) = i] \cdot \Pr_{h \in \mathcal{H}}[h(y) = j]$$

for all distinct items $x \neq y$ and all (possibly equal) hash values i and j.

Describe a set of hash functions that is uniform but not pairwise independent.

(e) Describe a set of hash functions that is pairwise independent but not (near-)uniform.

(f) Describe a set of hash functions that is universal but not pairwise independent.

(g) Describe a set of hash functions that is pairwise independent but not (near-)uniform.

(h) Describe a set of hash functions that is universal and pairwise independent but not uniform, or prove no such set exists.

3. (a) Prove that the set $\mathcal{MB}$ of binary multiplicative hash functions described in Section 12.5 is not uniform. *[Hint: What is* $\text{multb}_a(0)$*?]*

(b) Prove that $\mathcal{MB}$ is not pairwise independent. *[Hint: Compare* $\text{multb}_a(0)$ *and* $\text{multb}_a(2^{w-1})$*.]*

(c) Consider the following variant of multiplicative hashing, which uses slightly longer salt parameters. For any integers $a, b \in [2^{w+\ell}]$ where a is odd, let

$$h_{a,b}(x) := \big((a \cdot x + b) \bmod 2^{w+\ell}\big) \operatorname{div} 2^w = \left\lfloor \frac{(a \cdot x + b) \bmod 2^{w+\ell}}{2^w} \right\rfloor,$$

and let $\mathcal{MB}^+ = \{h_{a,b} \mid a, b \in [2^{w+\ell}] \text{ and } a \text{ odd}\}$. Prove that the family of hash functions $\mathcal{MB}^+$ is ***strongly near-universal***:

$$\Pr_{h \in \mathcal{MB}^+}\big[(h(x) = i) \wedge (h(y) = j)\big] \leq \frac{2}{m^2}$$

for all items $x \neq y$ and all (possibly equal) hash values i and j.

4. Suppose we are using an *open-addressed* hash table of size m to store n items, where $n \leq m/2$. Assume an ideal random hash function. For any i, let X_i denote the number of probes required for the ith insertion into the table, and let $X = \max_i X_i$ denote the length of the longest probe sequence.

   (a) Prove that $\Pr[X_i > k] \leq 1/2^k$ for all i and k.

   (b) Prove that $\Pr[X_i > 2 \lg n] \leq 1/n^2$ for all i.

   (c) Prove that $\Pr[X > 2 \lg n] \leq 1/n$.

   (d) Prove that $\mathrm{E}[X] = O(\log n)$.

Philosophers gathered from far and near
To sit at his feet and hear and hear,
Though he never was heard
To utter a word
But "Abracadabra, abracadab,
Abracada, abracad,
Abraca, abrac, abra, ab!"
'Twas all he had,
'Twas all they wanted to hear, and each
Made copious notes of the mystical speech,
Which they published next –
A trickle of text
In the meadow of commentary.
Mighty big books were these,
In a number, as leaves of trees;
In learning, remarkably – very!

— Jamrach Holobom, quoted by Ambrose Bierce,
The Devil's Dictionary (1911)

Why are our days numbered and not, say, lettered?

— Woody Allen, "Notes from the Overfed", *The New Yorker* (March 16, 1968)

13 String Matching

13.1 Brute Force

The basic object that we consider in this lecture note is a *string*, which is really just an array. The elements of the array come from a set Σ called the *alphabet*; the elements themselves are called *characters*. Common examples are ASCII text, where each character is an seven-bit integer, strands of DNA, where the alphabet is the set of nucleotides $\{A, C, G, T\}$, or proteins, where the alphabet is the set of 22 amino acids.

The problem we want to solve is the following. Given two strings, a ***text*** $T[1..n]$ and a ***pattern*** $P[1..m]$, find the first *substring* of the text that is the same as the pattern. (It would be easy to extend our algorithms to find *all* matching substrings, but we will resist.) A substring is just a contiguous subarray. For any *shift* s, let T_s denote the substring $T[s..s+m-1]$. So more formally, we want to find the smallest shift s such that $T_s = P$, or report that there is no match. For example, if the text is the string 'AMANAPLANACATACANALPANAMA'[1] and the pattern is 'CAN', then the output should be 15. If the pattern is 'SPAM', then the answer should be NONE. In most cases the pattern is much smaller than the text; to make this concrete, I'll assume that $m < n/2$.

[1]Dan Hoey (or rather, his computer program) found the following 540-word palindrome in 1984. We have better online dictionaries now, so I'm sure you could do better.

A man, a plan, a caret, a ban, a myriad, a sum, a lac, a liar, a hoop, a pint, a catalpa, a gas, an oil, a bird, a yell, a vat, a caw, a pax, a wag, a tax, a nay, a ram, a cap, a yam, a gay, a tsar, a wall, a car, a luger, a ward, a bin, a woman, a vassal, a wolf, a tuna, a nit, a pall, a fret, a watt, a bay, a daub, a tan, a cab, a datum, a gall, a hat, a fag, a zap, a say, a jaw, a lay, a wet, a gallop, a tug, a trot, a trap, a tram, a torr, a caper, a top, a tonk, a toll, a ball, a fair, a sax, a minim, a tenor, a bass, a passer, a capital, a rut, an amen, a ted, a cabal, a tang, a sun, an ass, a maw, a sag, a jam, a dam, a sub, a salt, an axon, a sail, an ad, a wadi, a radian, a room, a rood, a rip, a tad, a pariah, a revel, a reel, a reed, a pool, a plug, a pin, a peek, a parabola, a dog, a pat, a cud, a nu, a fan, a pal, a rum, a nod, an eta, a lag, an eel, a batik, a mug, a mot, a nap, a maxim, a mood, a leek, a grub, a gob, a gel, a drab, a citadel, a total, a cedar, a tap, a gag, a rat, a manor, a bar, a gal, a cola, a pap, a yaw, a tab, a raj, a gab, a nag, a pagan, a bag, a jar, a bat, a way, a papa, a local, a gar, a baron, a mat, a rag, a gap, a tar, a decal, a tot, a led, a tic, a bard, a leg, a bog, a burg, a keel, a doom, a mix, a map, an atom, a gum, a kit, a baleen, a gala, a ten, a don, a mural, a pan, a faun, a ducat, a pagoda, a lob, a rap, a keep, a nip, a gulp, a loop, a deer, a leer, a lever, a hair, a pad, a tapir, a door, a moor, an aid, a raid, a wad, an alias, an ox, an atlas, a bus, a madam, a jag, a saw, a mass, an anus, a gnat, a lab, a cadet, an em, a natural, a tip, a caress, a pass, a baronet, a minimax, a sari, a fall, a ballot, a knot, a pot, a rep, a carrot, a mart, a part, a tort, a gut, a poll, a gateway, a law, a jay, a sap, a zag, a fat, a hall, a gamut, a dab, a can, a tabu, a day, a batt, a waterfall, a patina, a nut, a flow, a lass, a van, a mow, a nib, a draw, a regular, a call, a war, a stay, a gam, a yap, a cam, a ray, an ax, a tag, a wax, a paw, a cat, a valley, a drib, a lion, a saga, a plat, a catnip, a pooh, a rail, a calamus, a dairyman, a bater, a canal—Panama!

Here's the 'obvious' brute force algorithm, but with one immediate improvement. The inner while loop compares the substring T_s with P. If the two strings are not equal, this loop stops at the first character mismatch.

```
ALMOSTBRUTEFORCE(T[1..n], P[1..m]):
  for s ← 1 to n − m + 1
      equal ← TRUE
      i ← 1
      while equal and i ≤ m
          if T[s + i − 1] ≠ P[i]
              equal ← FALSE
          else
              i ← i + 1
      if equal
          return s
  return NONE
```

In the worst case, the running time of this algorithm is $O((n-m)m) = O(nm)$, and we can actually achieve this running time by searching for the pattern `AAA...AAAB` with $m-1$ `A`'s, in a text consisting of n `A`'s.

In practice, though, breaking out of the inner loop at the first mismatch makes this algorithm quite practical. We can wave our hands at this by assuming that the text and pattern are both random. Then on average, we perform a constant number of comparisons at each position i, so the total expected number of comparisons is $O(n)$. Of course, neither English nor DNA is really random, so this is only a heuristic argument.

13.2 Strings as Numbers

For the moment, let's assume that the alphabet consists of the ten digits `0` through `9`, so we can interpret any array of characters as either a string or a decimal number. In particular, let p be the numerical value of the pattern P, and for any shift s, let t_s be the numerical value of T_s:

$$p = \sum_{i=1}^{m} 10^{m-i} \cdot P[i] \qquad t_s = \sum_{i=1}^{m} 10^{m-i} \cdot T[s+i-1]$$

For example, if $T =$ `3141592653589793`<u>`2384`</u>`626433832795028841971` and $m = 4$, then $t_{17} = 2384$.

Clearly we can rephrase our problem as follows: Find the smallest s, if any, such that $p = t_s$. We can compute p in $O(m)$ arithmetic operations, without having to explicitly compute powers of ten, using *Horner's rule*:

$$p = P[m] + 10\Big(P[m-1] + 10\big(P[m-2] + \cdots + 10\big(P[2] + 10 \cdot P[1]\big)\cdots\big)\Big)$$

We could also compute any t_s in $O(m)$ operations using Horner's rule, but this leads to essentially the same brute-force algorithm as before. But once we know t_s, we can actually compute t_{s+1} in constant time just by doing a little arithmetic — subtract off the most significant digit $T[s] \cdot 10^{m-1}$, shift everything up by one digit, and add the new least significant digit $T[r+m]$:

$$t_{s+1} = 10\big(t_s - 10^{m-1} \cdot T[s]\big) + T[s+m]$$

To make this fast, we need to precompute the constant 10^{m-1}. (And we know how to do that quickly, right?) So at least intuitively, it looks like we can solve the string matching problem in $O(n)$ worst-case time using the following algorithm:

```
NumberSearch(T[1..n], P[1..m]):
  σ ← 10^(m−1)
  p ← 0
  t_1 ← 0
  for i ← 1 to m
      p ← 10 · p + P[i]
      t_1 ← 10 · t_1 + T[i]
  for s ← 1 to n − m + 1
      if p = t_s
          return s
      t_{s+1} ← 10 · (t_s − σ · T[s]) + T[s + m]
  return None
```

Unfortunately, the most we can say is that the number of *arithmetic operations* is $O(n)$. These operations act on numbers with up to m digits. Since we want to handle arbitrarily long patterns, we can't assume that each operation takes only constant time! In fact, if we want to avoid expensive multiplications in the second-to-last line, we should represent each number as a string of decimal digits, which brings us back to our original brute-force algorithm!

13.3 Karp-Rabin Fingerprinting

To make this algorithm efficient, we will make one simple change, proposed by Richard Karp and Michael Rabin in 1981:

$$\boxed{\text{Perform all arithmetic modulo some prime number } q.}$$

We choose q so that the value $10q$ fits into a standard integer variable, so that we don't need any fancy long-integer data types. The values ($p \bmod q$) and ($t_s \bmod q$) are called the *fingerprints* of P and T_s, respectively. We can now compute ($p \bmod q$) and ($t_1 \bmod q$) in $O(m)$ *time* using Horner's rule:

$$p \bmod q = P[m] + \big(\cdots + \big(10 \cdot \big(P[2] + \big(10 \cdot P[1] \bmod q\big) \bmod q\big) \bmod q\big) \cdots\big)\big) \bmod q.$$

Similarly, given ($t_s \bmod q$), we can compute ($t_{s+1} \bmod q$) in constant time as follows:

$$t_{s+1} \bmod q = \big(10 \cdot \big(t_s - \big(\big(10^{m-1} \bmod q\big) \cdot T[s] \bmod q\big) \bmod q\big) \bmod q\big) + T[s+m] \bmod q.$$

Again, we have to precompute the value ($10^{m-1} \bmod q$) to make this fast.

If ($p \bmod q$) $\neq$ ($t_s \bmod q$), then certainly $P \neq T_s$. However, if ($p \bmod q$) $=$ ($t_s \bmod q$), we can't tell whether $P = T_s$ or not. All we know for sure is that p and t_s differ by some integer multiple of q. If $P \neq T_s$ in this case, we say there is a *false match* at shift s. To test for a false match, we simply do a brute-force string comparison. (In the algorithm below, $\tilde{p} = p \bmod q$ and $\tilde{t}_s = t_s \bmod q$.) The overall running time of the algorithm is $O(n + Fm)$, where F is the number of false matches.

Intuitively, we expect the fingerprints t_s to jump around between 0 and $q-1$ more or less at random, so the 'probability' of a false match 'ought' to be $1/q$. This intuition implies that $F = n/q$ "on average", which gives us an 'expected' running time of $O(n + nm/q)$. If we always choose $q \geq m$, this bound simplifies to $O(n)$.

But of course all this intuitive talk of probabilities is meaningless hand-waving, since we haven't actually done anything random yet! There are two simple methods to formalize this intuition.

Random Prime Numbers

The algorithm that Karp and Rabin actually proposed chooses the prime modulus q *randomly* from a sufficiently large range.

```
KarpRabin(T[1..n], P[1..m]):
  q ← a random prime number between 2 and ⌈m² lg m⌉
  σ ← 10^(m−1) mod q
  p̃ ← 0
  t̃_1 ← 0
  for i ← 1 to m
      p̃ ← (10 · p̃ mod q) + P[i] mod q
      t̃_1 ← (10 · t̃_1 mod q) + T[i] mod q
  for s ← 1 to n − m + 1
      if p̃ = t̃_s
          if P = T_s        ⟨⟨brute-force O(m)-time comparison⟩⟩
              return s
      t̃_{s+1} ← (10 · (t̃_s − (σ · T[s] mod q) mod q) mod q) + T[s + m] mod q
  return None
```

For any positive integer u, let $\pi(u)$ denote the number of prime numbers less than u. There are $\pi(m^2 \log m)$ possible values for q, each with the same probability of being chosen. Our analysis needs two results from number theory. I won't even try to prove the first one, but the second one is quite easy.

Lemma 1 (The Prime Number Theorem). $\pi(u) = \Theta(u/\log u)$.

Lemma 2. *Any integer x has at most $\lfloor \lg x \rfloor$ distinct prime divisors.*

Proof: If x has k distinct prime divisors, then $x \geq 2^k$, since every prime number is bigger than 1. □

Suppose there are no true matches, since a true match can only end the algorithm early, so $p \neq t_s$ for all s. There is a false match at shift s if and only if $\tilde{p} = \tilde{t}_s$, or equivalently, if q is one of the prime divisors of $|p - t_s|$. Because $p < 10^m$ and $t_s < 10^m$, we must have $|p - t_s| < 10^m$. Thus, Lemma 2 implies that $|p - t_s|$ has at most $O(m)$ prime divisors. We chose q randomly from a set of $\pi(m^2 \log m) = \Omega(m^2)$ prime numbers, so the probability of a false match at shift s is $O(1/m)$. Linearity of expectation now implies that the expected number of false matches is $O(n/m)$. We conclude that KarpRabin runs in $O(n + \mathrm{E}[F]m) = \boldsymbol{O(n)}$ ***expected time***.

Actually choosing a random prime number is not particularly easy; the best method known is to repeatedly generate a random integer and test whether it's prime. The Prime Number Theorem implies that we will find a prime number after $O(\log m)$ iterations. Testing whether a number x is prime by brute force requires roughly $O(\sqrt{x})$ divisions, each of which require $O(\log^2 x)$ time if we use standard long division. So the total time to choose q using this brute-force method is about $O(m \log^3 m)$. There are faster algorithms to test primality, but they are considerably more complex. In practice, it's enough to choose a random *probable* prime. Unfortunately, even describing what the phrase "probable prime" means is beyond the scope of this note.

Polynomial Hashing

A much simpler method relies on a classical string-hashing technique proposed by Lawrence Carter and Mark Wegman in the late 1970s. Instead of generating the prime modulus randomly, we generate *the radix of our number representation* randomly. Equivalently, we treat each string as the coefficient vector of a polynomial of degree $m-1$, and we evaluate that polynomial at some random number.

<u>CarterWegmanKarpRabin($T[1..n], P[1..m]$):</u>
  $q \leftarrow$ *prime number larger than* m^2
  $b \leftarrow$ Random$(q)-1$
  $\sigma \leftarrow b^{m-1} \bmod q$
  $\tilde{p} \leftarrow 0$
  $\tilde{t}_1 \leftarrow 0$
  for $i \leftarrow 1$ to m
    $\tilde{p} \leftarrow (b \cdot \tilde{p} \bmod q) + P[i] \bmod q$
    $\tilde{t}_1 \leftarrow (b \cdot \tilde{t}_1 \bmod q) + T[i] \bmod q$
  for $s \leftarrow 1$ to $n-m+1$
    if $\tilde{p} = \tilde{t}_s$
      if $P = T_s$ ⟨⟨*brute-force $O(m)$-time comparison*⟩⟩
        return s
    $\tilde{t}_{s+1} \leftarrow \big(b \cdot \big(\tilde{t}_s - \big(\sigma \cdot T[s] \bmod q\big) \bmod q\big) \bmod q\big) + T[s+m] \bmod q$
  return None

Fix an arbitrary prime number $q \geq m^2$, and choose b uniformly at random from the set $\{0, 1, \ldots, q-1\}$. We redefine the numerical values p and t_s using b in place of the alphabet size:

$$p(b) = \sum_{i=1}^{m} b^i \cdot P[m-i] \qquad t_s(b) = \sum_{i=1}^{m} b^i \cdot T[s-1+m-i],$$

Now define $\tilde{p}(b) = p(b) \bmod q$ and $\tilde{t}_s(b) = t_s(b) \bmod q$.

The function $f(b) = \tilde{p}(b) - \tilde{t}_s(b)$ is a polynomial of degree $m-1$ over the *variable* b. Because q is prime, the set $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$ with addition and multiplication modulo q defines a *field*. A standard theorem of abstract algebra states that any polynomial with degree $m-1$ over a field has at most $m-1$ roots in that field. Thus, there are at most $m-1$ elements $b \in \mathbb{Z}_q$ such that $f(b) = 0$.

It follows that if $P \neq T_s$, the probability of a false match at shift s is $\Pr_b[\tilde{p}(b) = \tilde{t}_s(b)] \leq (m-1)/q < 1/m$. Linearity of expectation now implies that the expected number of false positives is $O(n/m)$, so the modified Rabin-Karp algorithm also runs in ***$O(n)$ expected time***.

13.4 Redundant Comparisons

Let's go back to the character-by-character method for string matching. Suppose we are looking for the pattern 'ABRACADABRA' in some longer text using the (almost) brute force algorithm described in the previous lecture. Suppose also that when $s = 11$, the substring comparison fails at the fifth position; the corresponding character in the text (just after the vertical line below) is not a C. At this point, our algorithm would increment s and start the substring comparison from scratch.

```
HOCUSPOCUSABRA|BRACADABRA...
          ABRA|CADABRA
           ABR|ACADABRA
```

If we look carefully at the text and the pattern, however, we should notice right away that there's no point in looking at $s = 12$. We already know that the next character is a B — after all, it matched $P[2]$ during the previous comparison — so why bother even looking there? Likewise, we already know that the next two shifts $s = 13$ and $s = 14$ will also fail, so why bother looking there?

```
HOCUSPOCUSABRA|BRACADABRA...
          ABRA|CADABRA
           ABR|ACADABRA
            AB|RACADABRA
             A|BRACADABRA
```

Finally, when we get to $s = 15$, we can't immediately rule out a match based on earlier comparisons. However, for precisely the same reason, we shouldn't start the substring comparison over from scratch — we already know that $T[15] = P[4] =$ A. Instead, we should start the substring comparison at the *second* character of the pattern, since we don't yet know whether or not it matches the corresponding text character.

If you play with this idea long enough, you'll notice that the character comparisons should always advance through the text. **Once we've found a match for a text character, we never need to do another comparison with that text character again.** In other words, we should be able to optimize the brute-force algorithm so that it always *advances* through the text.

You'll also eventually notice a good rule for finding the next 'reasonable' shift s. A *prefix* of a string is a substring that includes the first character; a *suffix* is a substring that includes the last character. A prefix or suffix is *proper* if it is not the entire string. Suppose we have just discovered that $T[i] \neq P[j]$. **The next reasonable shift is the smallest value of s such that $T[s\,..\,i-1]$, which is a suffix of the previously-read text, is also a proper prefix of the pattern.**

in 1977, Donald Knuth, James Morris, and Vaughn Pratt published a string-matching algorithm that implements both of these ideas.

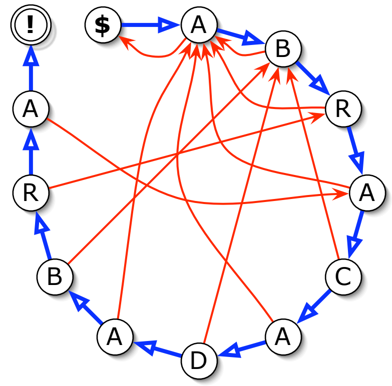
13.5 Finite State Machines

We can interpret any string matching algorithm that always advance through the text as feeding the text through a special type of ***finite-state machine***. A finite state machine is a directed graph. Each node (or ***state***) in the string-matching machine is labeled with a character from the pattern, except for two special nodes labeled \$ and !. Each node has two outgoing edges, a ***success*** edge and a ***failure*** edge. The success edges define a path through the characters of the pattern in order, starting at \$ and ending at !. Failure edges always point to earlier characters in the pattern.

We use the finite state machine to search for the pattern as follows. At all times, we have a current text character $T[i]$ and a current node in the graph, which is usually labeled by some pattern character $P[j]$. We iterate the following rules:

- If $T[i] = P[j]$, or if the current label is \$, follow the success edge to the next node and increment i. (So there is no failure edge from the start node \$.)
- If $T[i] \neq P[j]$, follow the failure edge back to an earlier node, but do not change i.

For the moment, let's simply assume that the failure edges are defined correctly—we'll see how to do that later. If we ever reach the node labeled !, then we've found an instance of the pattern in the text, and if we run out of text characters ($i > n$) before we reach !, then there is no match.



A finite state machine for the string 'ABRADACABRA'.
Thick arrows are the success edges; thin arrows are the failure edges.

The finite state machine is really just a (very!) convenient metaphor. In a real implementation, we would not construct the entire graph. Since the success edges always traverse the pattern characters in order, and each state has exactly one outgoing failure edge, we only have to remember the targets of the failure edges. We can encode this ***failure function*** in an array $fail[1..n]$, where for each index j, the failure edge from node j leads to node $fail[j]$. Following a failure edge back to an earlier state corresponds exactly, in our earlier formulation, to shifting the pattern forward. The failure function $fail[j]$ tells us how far to shift after a character mismatch $T[i] \neq P[j]$. Here's the actual algorithm:

```
KnuthMorrisPratt(T[1..n], P[1..m]):
  j ← 1
  for i ← 1 to n
      while j > 0 and T[i] ≠ P[j]
          j ← fail[j]
      if j = m        ⟨⟨Found it!⟩⟩
          return i − m + 1
      j ← j + 1
  return None
```

Before we discuss computing the failure function, let's analyze the running time of KnuthMorrisPratt under the assumption that a correct failure function is already known. At each character comparison, either we increase i and j by one, or we decrease j and leave i alone. We can increment i at most $n-1$ times before we run out of text, so there are at most $n-1$ successful comparisons. Similarly, there can be at most $n-1$ failed comparisons, since the number of times we decrease j cannot exceed the number of times we increment j. In other words, we can amortize character mismatches against earlier character matches. Thus, the total number of character comparisons performed by KnuthMorrisPratt in the worst case is $O(n)$.

13.6 Computing the Failure Function

We can now rephrase our second intuitive rule about how to choose a reasonable shift after a character mismatch $T[i] \neq P[j]$:

$P[1..fail[j]-1]$ is the longest proper prefix of $P[1..j-1]$ that is also a suffix of $T[1..i-1]$.

Notice, however, that if we are comparing $T[i]$ against $P[j]$, then we must have already matched the first $j-1$ characters of the pattern. In other words, we already know that $P[1..j-1]$ is a suffix of $T[1..i-1]$. Thus, we can rephrase the prefix-suffix rule as follows:

$\boxed{P[1..\mathit{fail}[j]-1] \text{ is the longest proper prefix of } P[1..j-1] \text{ that is also a suffix of } P[1..j-1].}$

This is the definition of the Knuth-Morris-Pratt failure function $\mathit{fail}[j]$ for all $j > 1$. By convention we set $\mathit{fail}[1] = 0$; this tells the KMP algorithm that if the first pattern character doesn't match, it should just give up and try the next text character.

$P[i]$	A	B	R	A	C	A	D	A	B	R	A
$\mathit{fail}[i]$	0	1	1	1	2	1	2	1	2	3	4

Failure function for the string 'ABRACADABRA'
(Compare with the finite state machine on the previous page.)

We could easily compute the failure function in $O(m^3)$ time by checking, for each j, whether every prefix of $P[1..j-1]$ is also a suffix of $P[1..j-1]$, but this is not the fastest method. The following algorithm essentially uses the KMP search algorithm to look for the pattern inside itself!

```
ComputeFailure(P[1..m]):
  j ← 0
  for i ← 1 to m
      fail[i] ← j        (∗)
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

Here's an example of this algorithm in action. In each line, the current values of i and j are indicated by superscripts; \$ represents the beginning of the string. (You should imagine pointing at $P[j]$ with your left hand and pointing at $P[i]$ with your right hand, and moving your fingers according to the algorithm's directions.)

Just as we did for KnuthMorrisPratt, we can analyze ComputeFailure by amortizing character mismatches against earlier character matches. Since there are at most m character matches, ComputeFailure runs in $O(m)$ time.

Let's prove (by induction, of course) that ComputeFailure correctly computes the failure function. The base case $\mathit{fail}[1] = 0$ is obvious. Assuming inductively that we correctly computed $\mathit{fail}[1]$ through $\mathit{fail}[i-1]$ in line $(*)$, we need to show that $\mathit{fail}[i]$ is also correct. Just after the ith iteration of line $(*)$, we have $j = \mathit{fail}[i]$, so $P[1..j-1]$ is the longest proper prefix of $P[1..i-1]$ that is also a suffix.

Let's define the iterated failure functions $\mathit{fail}^c[j]$ inductively as follows: $\mathit{fail}^0[j] = j$, and

$$\mathit{fail}^c[j] = \mathit{fail}[\mathit{fail}^{c-1}[j]] = \overbrace{\mathit{fail}[\mathit{fail}[\cdots[\mathit{fail}}^{c}[j]]\cdots]].$$

In particular, if $\mathit{fail}^{c-1}[j] = 0$, then $\mathit{fail}^c[j]$ is undefined. We can easily show by induction that every string of the form $P[1..\mathit{fail}^c[j]-1]$ is both a proper prefix and a proper suffix of $P[1..i-1]$, and in fact, these are the only examples. Thus, the longest proper prefix/suffix of $P[1..i]$ must be the longest string of the form $P[1..\mathit{fail}^c[j]]$—the one with smallest c—such that $P[\mathit{fail}^c[j]] = P[i]$. This is exactly what the while loop in ComputeFailure computes;

$j \leftarrow 0, i \leftarrow 1$	$\mathbf{\$}^j$	$\mathbf{A}^i$	B	R	A	C	A	D	A	B	R	X	...
$\mathit{fail}[i] \leftarrow j$		**0**											...
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	$\mathbf{B}^i$	R	A	C	A	D	A	B	R	X	...
$\mathit{fail}[i] \leftarrow j$		0	**1**										...
$j \leftarrow \mathit{fail}[j]$	$\mathbf{\$}^j$	A	$\mathbf{B}^i$	R	A	C	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	B	$\mathbf{R}^i$	A	C	A	D	A	B	R	X	...
$\mathit{fail}[i] \leftarrow j$		0	1	**1**									...
$j \leftarrow \mathit{fail}[j]$	$\mathbf{\$}^j$	A	B	$\mathbf{R}^i$	A	C	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	B	R	$\mathbf{A}^i$	C	A	D	A	B	R	X	...
$\mathit{fail}[i] \leftarrow j$		0	1	1	**1**								...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	$\mathbf{B}^j$	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	...
$\mathit{fail}[i] \leftarrow j$		0	1	1	1	**2**							...
$j \leftarrow \mathit{fail}[j]$	\$	$\mathbf{A}^j$	B	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	...
$j \leftarrow \mathit{fail}[j]$	$\mathbf{\$}^j$	A	B	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	B	R	A	C	$\mathbf{A}^i$	D	A	B	R	X	...
$\mathit{fail}[i] \leftarrow j$		0	1	1	1	2	**1**						...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	$\mathbf{B}^j$	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	...
$\mathit{fail}[i] \leftarrow j$		0	1	1	1	2	1	**2**					...
$j \leftarrow \mathit{fail}[j]$	\$	$\mathbf{A}^j$	B	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	...
$j \leftarrow \mathit{fail}[j]$	$\mathbf{\$}^j$	A	B	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	...
$j \leftarrow j+1, i \leftarrow i+1$	\$	$\mathbf{A}^j$	B	R	A	C	A	D	$\mathbf{A}^i$	B	R	X	...
$\mathit{fail}[i] \leftarrow j$		0	1	1	1	2	1	2	**1**				...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	$\mathbf{B}^j$	R	A	C	A	D	A	$\mathbf{B}^i$	R	X	...
$\mathit{fail}[i] \leftarrow j$		0	1	1	1	2	1	2	1	**2**			...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	B	$\mathbf{R}^j$	A	C	A	D	A	B	$\mathbf{R}^i$	X	...
$\mathit{fail}[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	**3**		...
$j \leftarrow j+1, i \leftarrow i+1$	\$	A	B	R	$\mathbf{A}^j$	C	A	D	A	B	R	$\mathbf{X}^i$	...
$\mathit{fail}[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	3	**4**	...
$j \leftarrow \mathit{fail}[j]$	\$	$\mathbf{A}^j$	B	R	A	C	A	D	A	B	R	$\mathbf{X}^i$	...
$j \leftarrow \mathit{fail}[j]$	$\mathbf{\$}^j$	A	B	R	A	C	A	D	A	B	R	$\mathbf{X}^i$	...

ComputeFailure in action. Do this yourself by hand!

the $(c+1)$th iteration compares $P[fail^c[j]] = P[fail^{c+1}[i]]$ against $P[i]$. ComputeFailure is actually a *dynamic programming* implementation of the following recursive definition of $fail[i]$:

$$fail[i] = \begin{cases} 0 & \text{if } i = 0, \\ \max\limits_{c \geq 1} \left\{ fail^c[i-1] + 1 \;\middle|\; P[i-1] = P[fail^c[i-1]] \right\} & \text{otherwise.} \end{cases}$$

13.7 Optimizing the Failure Function

We can speed up KnuthMorrisPratt slightly by making one small change to the failure function. Recall that after comparing $T[i]$ against $P[j]$ and finding a mismatch, the algorithm compares $T[i]$ against $P[fail[j]]$. With the current definition, however, it is possible that $P[j]$ and $P[fail[j]]$ are actually the same character, in which case the next character comparison will automatically fail. So why do the comparison at all?

We can optimize the failure function by 'short-circuiting' these redundant comparisons with some simple post-processing:

```
OptimizeFailure(P[1..m], fail[1..m]):
  for i ← 2 to m
      if P[i] = P[fail[i]]
          fail[i] ← fail[fail[i]]
```

We can also compute the optimized failure function directly by adding three new lines (in bold) to the ComputeFailure function.

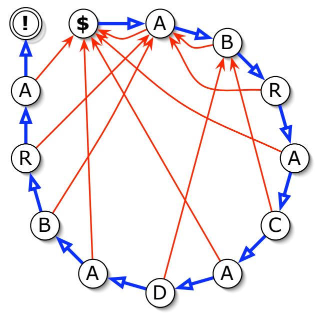
```
ComputeOptFailure(P[1..m]):
  j ← 0
  for i ← 1 to m
      if P[i] = P[j]
          fail[i] ← fail[j]
      else
          fail[i] ← j
      while j > 0 and P[i] ≠ P[j]
          j ← fail[j]
      j ← j + 1
```

This optimization slows down the preprocessing slightly, but it may significantly decrease the number of comparisons at each text character. The worst-case running time is still $O(n)$; however, the constant is about half as big as for the unoptimized version, so this could be a significant improvement in practice. Several examples of this optimization are given on the next page.

Exercises

1. Describe and analyze a two-dimensional variant of KarpRabin that searches for a given two-dimensional pattern $P[1..p][1..q]$ within a given two-dimensional "text" $T[1..m][1..,n]$. Your algorithm should report *all* index pairs (i, j) such that the subarray $T[i..i+p-1][j..j+q-1]$ is identical to the given pattern, in $O(pq+mn)$ expected time.

2. A *palindrome* is any string that is the same as its reversal, such as `X`, `ABBA`, or `REDIVIDER`. Describe and analyze an algorithm that computes the longest palindrome that is a (not



$P[i]$	A	B	R	A	C	A	D	A	B	R	A
unoptimized $fail[i]$	0	1	1	1	2	1	2	1	2	3	4
optimized $fail[i]$	0	1	1	**0**	2	**0**	2	**0**	**1**	**1**	**1**

Optimized finite state machine and failure function for the string 'ABRADACABRA'

$P[i]$	A	N	A	N	A	B	A	N	A	N	A	N	A
unoptimized $fail[i]$	0	1	1	2	3	4	1	2	3	4	5	6	5
optimized $fail[i]$	0	1	0	1	0	4	0	1	0	1	0	6	0

$P[i]$	A	B	A	B	C	A	B	A	B	C	A	B	C
unoptimized $fail[i]$	0	1	1	2	3	1	2	3	4	5	6	7	8
optimized $fail[i]$	0	1	0	1	3	0	1	0	1	3	0	1	8

$P[i]$	A	B	B	A	B	B	A	B	A	B	B	A	B
unoptimized $fail[i]$	0	1	1	1	2	3	4	5	6	2	3	4	5
optimized $fail[i]$	0	1	1	0	1	1	0	1	6	1	1	0	1

Failure functions for four more example strings.

necessarily proper) prefix of a given string $T[1..n]$. Your algorithm should run in $O(n)$ time (either expected or worst-case).

*3. How important is the requirement that the fingerprint modulus q is prime in the original Karp-Rabin algorithm? Specifically, suppose q is chosen uniformly at random in the range $1..N$. If $t_s \neq p$, what is the probability that $\tilde{t}_s = \tilde{p}$? What does this imply about the expected number of false matches? How large should N be to guarantee expected running time $O(m+n)$? *[Hint: This will require some additional number theory.]*

4. Describe a modification of KnuthMorrisPratt in which the pattern can contain any number of *wildcard* symbols **∗**, each of which matches an arbitrary string. For example, the pattern ABR∗CAD∗BRA appears in the text SCHABR**AIN**CADBRANCH; in this case, the second **∗** matches the empty string. Your algorithm should run in $O(m+n)$ time, where m is the length of the pattern and n is the length of the text.

5. Describe a modification of KnuthMorrisPratt in which the pattern can contain any number of *wildcard* symbols **?**, each of which matches an arbitrary single character. For example, the pattern ABR**?**CAD**?**BRA appears in the text SCHABR**U**CAD**I**BRANCH. Your algorithm should run in $O(m+qn)$ time, where m is the length of the pattern, n is the length of the text., and q is the number of **?**s in the pattern.

*6. Describe another algorithm for the previous problem that runs in time $O(m + kn)$, where k is the number of runs of consecutive non-wildcard characters in the pattern. For example, the pattern **?**`FISH`**???**`B`**??**`IS`**????**`CUIT`**?** has $k = 4$ runs.

7. Describe a modification of KnuthMorrisPratt in which the pattern can contain any number of *wildcard* symbols **=**, each of which matches *the same* arbitrary single character. For example, the pattern `=HOC=SPOC=S` appears in the texts `WHUHOCUSPOCUSOT` and `ABRAHOCASPOCASCADABRA`, but *not* in the text `FRISHOCUSPOCESTIX`. Your algorithm should run in $O(m + n)$ time, where m is the length of the pattern and n is the length of the text.

8. This problem considers the maximum length of a *failure chain* $j \to \mathit{fail}[j] \to \mathit{fail}[\mathit{fail}[j]] \to \mathit{fail}[\mathit{fail}[\mathit{fail}[j]]] \to \cdots \to 0$, or equivalently, the maximum number of iterations of the inner loop of KnuthMorrisPratt. This clearly depends on which failure function we use: unoptimized or optimized. Let m be an arbitrary positive integer.

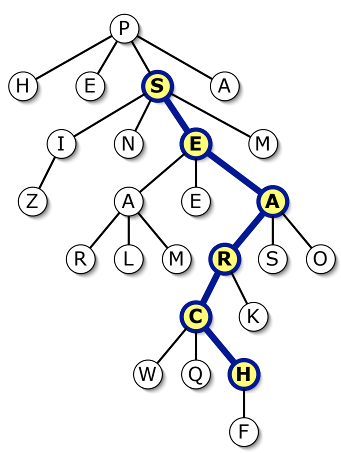
   (a) Describe a pattern $A[1..m]$ whose longest *unoptimized* failure chain has length m.

   (b) Describe a pattern $B[1..m]$ whose longest *optimized* failure chain has length $\Theta(\log m)$.

   *(c) Describe a pattern $C[1..m]$ *containing only two different characters*, whose longest optimized failure chain has length $\Theta(\log m)$.

   *(d) Prove that for any pattern of length m, the longest optimized failure chain has length at most $O(\log m)$.

9. Suppose we want to search for a string inside a labeled rooted tree. Our input consists of a *pattern string* $P[1..m]$ and a rooted *text tree* T with n nodes, each labeled with a single character. Nodes in T can have any number of children. Our goal is to either return a downward path in T whose labels match the string P, or report that there is no such path.

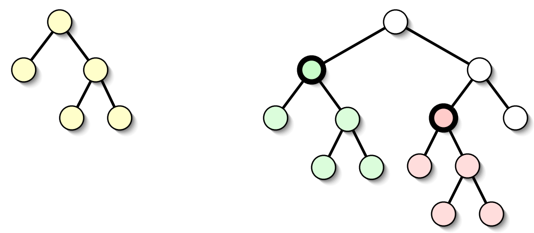


The string SEARCH appears on a downward path in the tree.

   (a) Describe and analyze a variant of KarpRabin that solves this problem in $O(m + n)$ expected time.

   (b) Describe and analyze a variant of KnuthMorrisPratt that solves this problem in $O(m + n)$ expected time.

10. Suppose we want to search a rooted binary tree for subtrees of a certain shape. The input consists of a *pattern tree* P with m nodes and a *text tree* T with n nodes. Every node in both trees has a left subtree and a right subtree, either or both of which may be empty. We want to report *all* nodes v in T such that the subtree rooted at v is structurally identical to P, ignoring all search keys, labels, or other data in the nodes—only the left/right pointer structure matters.



The pattern tree (left) appears exactly twice in the text tree (right).

(a) Describe and analyze a variant of KarpRabin that solves this problem in $O(m+n)$ expected time.

(b) Describe and analyze a variant of KnuthMorrisPratt that solves this problem in $O(m+n)$ expected time.

Jaques: *But, for the seventh cause; how did you find the quarrel on the seventh cause?*

Touchstone: *Upon a lie seven times removed:–bear your body more seeming, Audrey:–as thus, sir. I did dislike the cut of a certain courtier's beard: he sent me word, if I said his beard was not cut well, he was in the mind it was: this is called the Retort Courteous. If I sent him word again 'it was not well cut,' he would send me word, he cut it to please himself: this is called the Quip Modest. If again 'it was not well cut,' he disabled my judgment: this is called the Reply Churlish. If again 'it was not well cut,' he would answer, I spake not true: this is called the Reproof Valiant. If again 'it was not well cut,' he would say I lied: this is called the Counter-cheque Quarrelsome: and so to the Lie Circumstantial and the Lie Direct.*

Jaques: *And how oft did you say his beard was not well cut?*

Touchstone: *I durst go no further than the Lie Circumstantial, nor he durst not give me the Lie Direct; and so we measured swords and parted.*

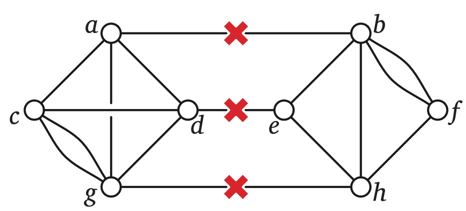
— William Shakespeare, *As You Like It*, Act V, Scene 4 (1600)

13 Randomized Minimum Cut

13.1 Setting Up the Problem

This lecture considers a problem that arises in robust network design. Suppose we have a connected multigraph[1] G representing a communications network like the UIUC telephone system, the Facebook social network, the internet, or Al-Qaeda. In order to disrupt the network, an enemy agent plans to remove some of the edges in this multigraph (by cutting wires, placing police at strategic drop-off points, or paying street urchins to 'lose' messages) to separate it into multiple components. Since his country is currently having an economic crisis, the agent wants to remove as few edges as possible to accomplish this task.

More formally, a *cut* partitions the nodes of G into two nonempty subsets. The *size* of the cut is the number of *crossing edges*, which have one endpoint in each subset. Finally, a *minimum* cut in G is a cut with the smallest number of crossing edges. The same graph may have several minimum cuts.



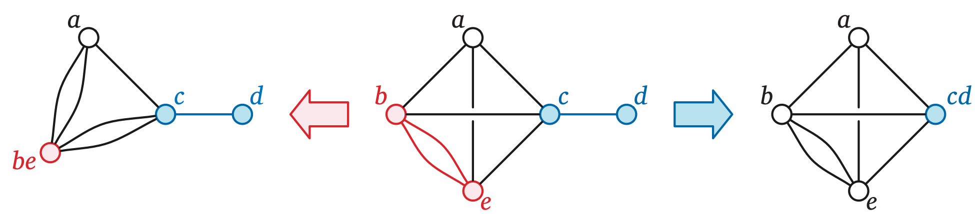
A multigraph whose minimum cut has three edges.

This problem has a long history. The classical deterministic algorithms for this problem rely on *network flow* techniques, which are discussed in another lecture. The fastest such algorithms (that we will discuss) run in $O(n^3)$ time and are fairly complex; we will see some of these later in the semester. Here I'll describe a relatively simple randomized algorithm discovered by David Karger when he was a Ph.D. student.[2]

[1]A multigraph allows multiple edges between the same pair of nodes. Everything in this lecture could be rephrased in terms of simple graphs where every edge has a non-negative weight, but this would make the algorithms and analysis slightly more complicated.

[2]David R. Karger*. Random sampling in cut, flow, and network design problems. Proc. 25th STOC, 648–657, 1994.

Karger's algorithm uses a primitive operation called *collapsing an edge*. Suppose u and v are vertices that are connected by an edge in some multigraph G. To collapse the edge $\{u,v\}$, we create a new node called uv, replace any edge of the form $\{u,w\}$ or $\{v,w\}$ with a new edge $\{uv,w\}$, and then delete the original vertices u and v. Equivalently, collapsing the edge shrinks the edge down to nothing, pulling the two endpoints together. The new collapsed graph is denoted $G/\{u,v\}$. We don't allow self-loops in our multigraphs; if there are multiple edges between u and v, collapsing any one of them deletes them all.



A graph G and two collapsed graphs $G/\{b,e\}$ and $G/\{c,d\}$.

Any edge in an n-vertex graph can be collapsed in $O(n)$ time, assuming the graph is represented as an adjacency list; I'll leave the precise implementation details as an easy exercise.

The correctness of our algorithms will eventually boil down the following simple observation: For any cut in $G/\{u,v\}$, there is cut in G with exactly the same number of crossing edges. In fact, in some sense, the 'same' edges form the cut in both graphs. The converse is not necessarily true, however. For example, in the picture above, the original graph G has a cut of size 1, but the collapsed graph $G/\{c,d\}$ does not.

This simple observation has two immediate but important consequences. First, collapsing an edge cannot decrease the minimum cut size. More importantly, collapsing an edge increases the minimum cut size if and only if that edge is part of *every* minimum cut.

13.2 Blindly Guessing

Let's start with an algorithm that tries to *guess* the minimum cut by randomly collapsing edges until the graph has only two vertices left.

```
GuessMinCut(G):
  for i ← n downto 2
      pick a random edge e in G
      G ← G/e
  return the only cut in G
```

Because each collapse requires $O(n)$ time, this algorithm runs in $O(n^2)$ time. Our earlier observations imply that as long as we never collapse an edge that lies in every minimum cut, our algorithm will actually guess correctly. But how likely is that?

Suppose G has only one minimum cut—if it actually has more than one, just pick your favorite—and this cut has size k. Every vertex of G must lie on at least k edges; otherwise, we could separate that vertex from the rest of the graph with an even smaller cut. Thus, the number of incident vertex-edge pairs is at least kn. Since every edge is incident to exactly two vertices, G must have at least $kn/2$ edges. That implies that if we pick an edge in G uniformly at random, the probability of picking an edge in the minimum cut is at most $2/n$. In other words, the probability that we don't screw up on the very first step is at least $1-2/n$.

Once we've collapsed the first random edge, the rest of the algorithm proceeds recursively (with independent random choices) on the remaining $(n-1)$-node graph. So the overall probability $P(n)$ that GUESSMINCUT returns the true minimum cut is given by the recurrence

$$P(n) \geq \frac{n-2}{n} \cdot P(n-1)$$

with base case $P(2) = 1$. We can expand this recurrence into a product, most of whose factors cancel out immediately.

$$P(n) \geq \prod_{i=3}^{n} \frac{i-2}{i} = \frac{\prod_{i=3}^{n}(i-2)}{\prod_{i=3}^{n} i} = \frac{\prod_{j=1}^{n-2} j}{\prod_{i=3}^{n} i} = \boxed{\frac{2}{n(n-1)}}$$

13.3 Blindly Guessing Over and Over

That's not very good. Fortunately, there's a simple method for increasing our chances of finding the minimum cut: run the guessing algorithm many times and return the smallest guess. Randomized algorithms folks like to call this idea *amplification*.

```
KARGERMINCUT(G):
  mink ← ∞
  for i ← 1 to N
      X ← GUESSMINCUT(G)
      if |X| < mink
          mink ← |X|
          minX ← X
  return minX
```

Both the running time and the probability of success will depend on the number of iterations N, which we haven't specified yet.

First let's figure out the probability that KARGERMINCUT returns the actual minimum cut. The only way for the algorithm to return the wrong answer is if GUESSMINCUT fails N times in a row. Since each guess is independent, our probability of success is at least

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^N \leq 1 - e^{-2N/n(n-1)},$$

by The World's Most Useful Inequality $1 + x \leq e^x$. By making N larger, we can make this probability arbitrarily close to 1, but never equal to 1. In particular, if we set $\boldsymbol{N = c\binom{n}{2}\ln n}$ for some constant c, then KARGERMINCUT is correct with probability at least

$$1 - e^{-c \ln n} = 1 - \frac{1}{n^c}.$$

When the failure probability is a polynomial fraction, we say that the algorithm is correct *with high probability*. Thus, KARGERMINCUT computes the minimum cut of any n-node graph in $\boldsymbol{O(n^4 \log n)}$ time.

If we make the number of iterations even larger, say $N = n^2(n-1)/2$, the success probability becomes $1 - e^{-n}$. When the failure probability is exponentially small like this, we say that the algorithm is correct with *very* high probability. In practice, very high probability is usually overkill; high probability is enough. (Remember, there is a small but non-zero probability that your computer will transform itself into a kitten before your program is finished.)

13.4 Not-So-Blindly Guessing

The $O(n^4 \log n)$ running time is actually comparable to some of the simpler flow-based algorithms, but it's nothing to get excited about. But we can improve our guessing algorithm, and thus decrease the number of iterations in the outer loop, by observing that *as the graph shrinks, the probability of collapsing an edge in the minimum cut increases*. At first the probability is quite small, only $2/n$, but near the end of execution, when the graph has only three vertices, we have a $2/3$ chance of screwing up!

A simple technique for working around this increasing probability of error was developed by David Karger and Cliff Stein.[3] Their idea is to group the first several random collapses a 'safe' phase, so that the cumulative probability of screwing up is small—less than 1/2, say—and a 'dangerous' phase, which is much more likely to screw up.

The safe phase shrinks the graph from n nodes to $n/\sqrt{2}+1$ nodes, using a sequence of $n-n/\sqrt{2}-1$ random collapses. Following our earlier analysis, the probability that *none* of these safe collapses touches the minimum cut is at least

$$\prod_{i=n/\sqrt{2}+2}^{n} \frac{i-2}{i} = \frac{(n/\sqrt{2})(n/\sqrt{2}+1)}{n(n-1)} = \frac{n+\sqrt{2}}{2(n-1)} > \frac{1}{2}.$$

Now, to get around the danger of the dangerous phase, we use amplification. However, instead of running through the dangerous phase once, we run it *twice* and keep the best of the two answers. Naturally, we treat the dangerous phase recursively, so we actually obtain a binary recursion tree, which expands as we get closer to the base case, instead of a single path. More formally, the algorithm looks like this:

```
CONTRACT(G, m):
  for i ← n downto m
      pick a random edge e in G
      G ← G/e
  return G
```

```
BETTERGUESS(G):
  if G has more than 8 vertices
      X_1 ← BETTERGUESS(CONTRACT(G, n/√2 + 1))
      X_2 ← BETTERGUESS(CONTRACT(G, n/√2 + 1))
      return min{X_1, X_2}
  else
      use brute force
```

This might look like we're just doing to same thing twice, but remember that CONTRACT (and thus BETTERGUESS) is randomized. Each call to CONTRACT contracts an independent random set of edges; X_1 and X_2 are almost always different cuts.

BETTERGUESS correctly returns the minimum cut unless *both* recursive calls return the wrong result. X_1 is the minimum cut of G if and only if (1) none of the edges of the minimum cut are CONTRACTed and (2) the recursive call to BETTERGUESS returns the minimum cut of the CONTRACTed graph. Thus, if $P(n)$ denotes the probability that BETTERGUESS returns a minimum cut of an n-node graph, then X_1 is the minimum cut with probability at least $1/2 \cdot P(n/\sqrt{2}+1)$. The same argument implies that X_2 is the minimum cut with probability at least $1/2 \cdot P(n/\sqrt{2}+1)$. Because these two events are independent, we have the following recurrence, with base case $P(n) = 1$ for all $n \le 6$.

$$P(n) \ge 1 - \left(1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

Using a series of transformations, Karger and Stein prove that $P(n) = \Omega(1/\log n)$. I've included the proof at the end of this note.

[3] David R. Karger* and Cliff Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. Proc. 25th STOC, 757–765, 1993.

For the running time, we get a simple recurrence that is easily solved using recursion trees or the Master theorem (after a domain transformation to remove the +1 from the recurrence).

$$T(n) = O(n^2) + 2T\left(\frac{n}{\sqrt{2}} + 1\right) = O(n^2 \log n)$$

So all this splitting and recursing has slowed down the guessing algorithm slightly, but the probability of failure is *exponentially* smaller!

Let's express the lower bound $P(n) = \Omega(1/\log n)$ explicitly as $P(n) \geq \alpha/\ln n$ for some constant α. (Karger and Stein's proof implies $\alpha > 2$). If we call BetterGuess $N = c\ln^2 n$ times, for some new constant c, the overall probability of success is at least

$$1 - \left(1 - \frac{\alpha}{\ln n}\right)^{c\ln^2 n} \geq 1 - e^{-(c/\alpha)\ln n} = 1 - \frac{1}{n^{c/\alpha}}.$$

By setting c sufficiently large, we can bound the probability of failure by an arbitrarily small polynomial function of n. In other words, we now have an algorithm that computes the minimum cut with high probability in only $\boldsymbol{O(n^2 \log^3 n)}$ time!

*13.5 Solving the Karger-Stein recurrence

Recall the following recurrence for the probability that BetterGuess successfully finds a minimum cut of an n-node graph:

$$P(n) \geq 1 - \left(1 - \frac{1}{2}P\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

Karger and Stein solve this rather ugly recurrence through a series of functional transformations. Let $p(k)$ denote the probability of success at the kth level of recursion, counting upward from the base case. This function satisfies the recurrence

$$p(k) \geq 1 - \left(1 - \frac{p(k-1)}{2}\right)^2 = p(k-1) - \frac{p(k-1)^2}{4}$$

with base case $p(0) = 1$. Let $\bar{p}(k)$ be the function that satisfies this recurrence with equality; clearly, $p(k) \geq \bar{p}(k)$. Substituting the function $z(k) = 4/\bar{p}(k) - 1$ into this recurrence implies (after a bit of algebra) gives a new recurrence

$$z(k) = z(k-1) + 2 + \frac{1}{z(k-1)}$$

with base case $z(0) = 3$. Clearly $z(k) > 1$ for all k, so we have a conservative upper bound $z(k) < z(k-1) + 3$, which implies (by induction) that $z(k) \leq 3k + 3$. Substituting $\bar{p}(k) = 4/(z(k) + 1)$ into this solution, we conclude that

$$p(k) \geq \bar{p}(k) > \frac{1}{3k+6} = \Omega(1/k).$$

To compute the number of levels of recursion that BetterGuess executes for an n-node graph, we solve the secondary recurrence

$$k(n) = 1 + k\left(\frac{n}{\sqrt{2}} + 1\right)$$

with base cases $k(n) = 0$ for all $n \leq 8$. After a domain transformation to remove the +1 from the right side, the recursion tree method (or the Master theorem) implies that $k(n) = \Theta(\log n)$.

We conclude that $\boldsymbol{P(n) = p(k(n)) = \Omega(1/\log n)}$, as promised. Whew!

Exercises

1. Suppose you had an algorithm to compute the minimum spanning tree of a graph in $O(m)$ time, where m is the number of edges in the input graph. Use this algorithm as a subroutine to improve the running time of GuessMinCut from $O(n^2)$ to $O(m)$.

   (In fact, there *is* a randomized algorithm—due to Philip Klein, David Karger, and Robert Tarjan—that computes the minimum spanning tree of any graph in $O(m)$ *expected* time. The fastest deterministic algorithm known in 2013 runs in $O(m\alpha(m))$ time.)

2. Suppose you are given a graph G with weighted edges, and your goal is to find a cut whose total weight (not just number of edges) is smallest.

   (a) Describe an algorithm to select a random edge of G, where the probability of choosing edge e is proportional to the weight of e.

   (b) Prove that if you use the algorithm from part (a), instead of choosing edges uniformly at random, the probability that GuessMinCut returns a minimum-weight cut is still $\Omega(1/n^2)$.

   (c) What is the running time of your modified GuessMinCut algorithm?

3. Prove that GuessMinCut returns the *second* smallest cut in its input graph with probability $\Omega(1/n^3)$. (The second smallest cut could be significantly larger than the minimum cut.)

4. Consider the following generalization of the BetterGuess algorithm, where we pass in a real parameter $\alpha > 1$ in addition to the graph G.

```
BetterGuess(G, α):
  n ← number of vertices in G
  if n > 8
      X_1 ← BetterGuess(Contract(G, n/α), α)
      X_2 ← BetterGuess(Contract(G, n/α), α)
      return min{X_1, X_2}
  else
      use brute force
```

   Assume for this question that the input graph G has a unique minimum cut.

   (a) What is the running time of the modified algorithm, as a function of n and α? *[Hint: Consider the cases $\alpha < \sqrt{2}$, $\alpha = \sqrt{2}$, and $\alpha > \sqrt{2}$ separately.]*

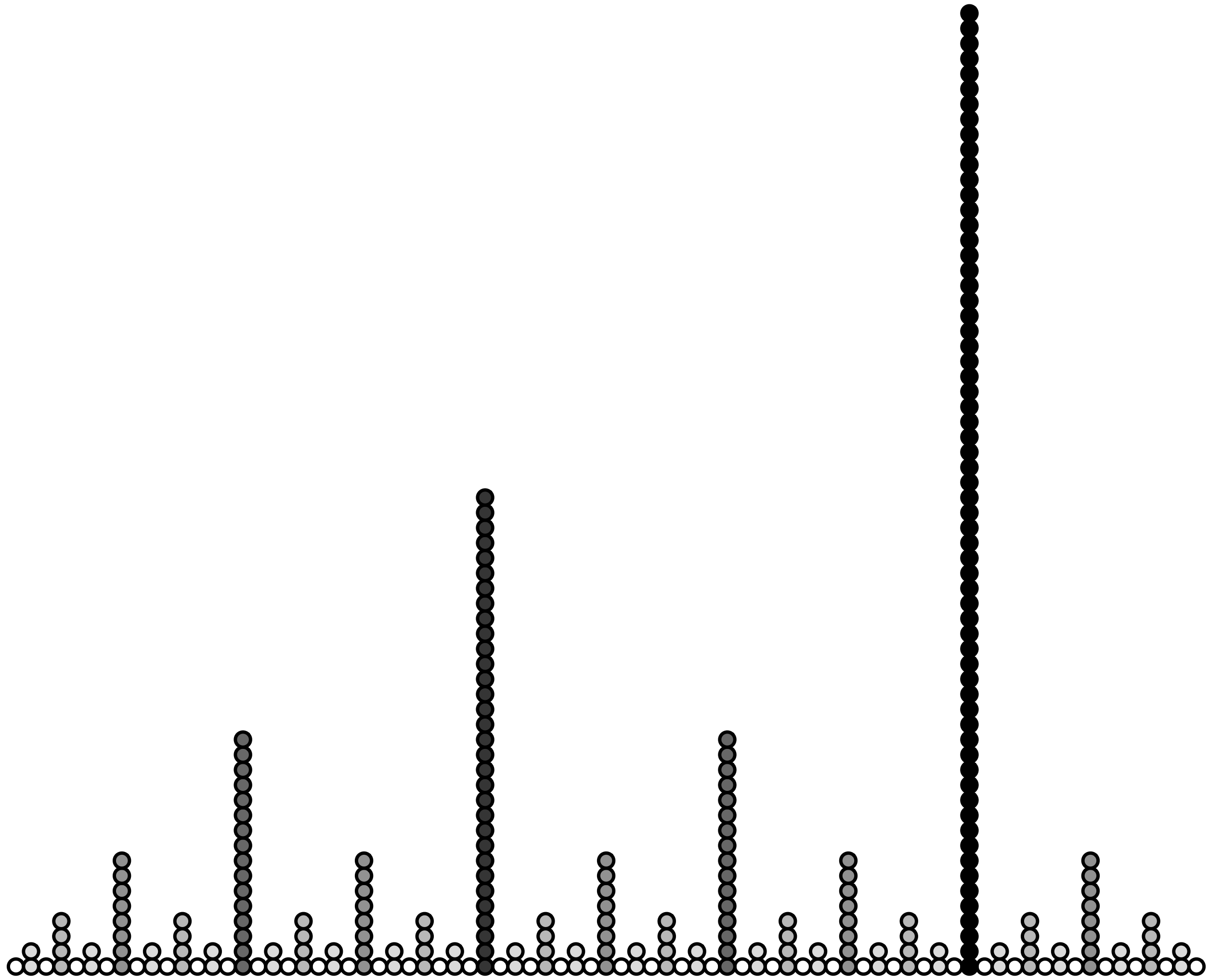
   (b) What is the probability that Contract$(G, n/\alpha)$ does not contract any edge in the minimum cut in G? Give both an exact expression involving both n and α, and a simple approximation in terms of just α. *[Hint: When $\alpha = \sqrt{2}$, the probability is approximately $1/2$.]*

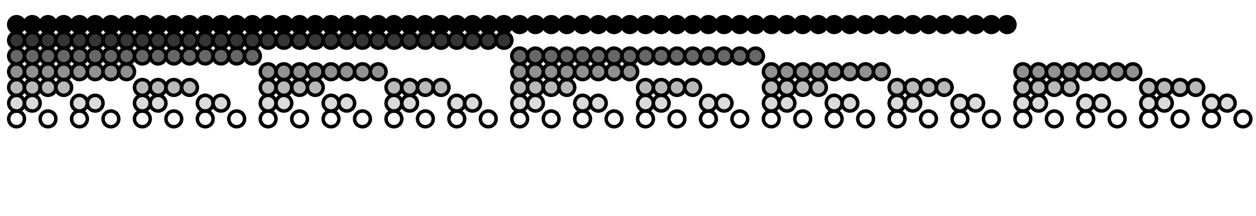
   (c) Estimate the probability that BetterGuess(G, α) returns the minimum cut in G, by adapting the solution to the Karger-Stein recurrence. *[Hint: Consider the cases $\alpha < \sqrt{2}$, $\alpha = \sqrt{2}$, and $\alpha > \sqrt{2}$ separately.]*

(d) Suppose we iterate BetterGuess(G, α) until we are guaranteed to see the minimum cut with high probability. What is the running time of the resulting algorithm? For which value of α is this running time minimized?

(e) Suppose we modify BetterGuess(G, α) further, to recurse four times instead of only twice. Now what is the best choice of α? What is the resulting running time?

Amortization





The goode workes that men don whil they ben in good lif al amortised by synne folwyng.

— Geoffrey Chaucer, "The Persones [Parson's] Tale" (c.1400)

I will gladly pay you Tuesday for a hamburger today.

— J. Wellington Wimpy, "Thimble Theatre" (1931)

I want my two dollars!

— Johnny Gasparini [Demian Slade], "Better Off Dead" (1985)

A dollar here, a dollar there. Over time, it adds up to two dollars.

— Jarod Kintz, *The Titanic Would Never Have Sunk if It Were Made out of a Sink* (2012)

15 Amortized Analysis

15.1 Incrementing a Binary Counter

It is a straightforward exercise in induction, which often appears on Homework 0, to prove that any non-negative integer n can be represented as the sum of distinct powers of 2. Although some students correctly use induction on the number of bits—pulling off either the least significant bit or the most significant bit in the binary representation and letting the Recursion Fairy convert the remainder—the most commonly submitted proof uses induction on the value of the integer, as follows:

Proof: The base case $n = 0$ is trivial. For any $n > 0$, the inductive hypothesis implies that there is set of distinct powers of 2 whose sum is $n-1$. If we add 2^0 to this set, we obtain a *multiset* of powers of two whose sum is n, which might contain two copies of 2^0. Then as long as there are two copies of any 2^i in the multiset, we remove them both and insert 2^{i+1} in their place. The sum of the elements of the multiset is unchanged by this replacement, because $2^{i+1} = 2^i + 2^i$. Each iteration decreases the size of the multiset by 1, so the replacement process must eventually terminate. When it does terminate, we have a *set* of distinct powers of 2 whose sum is n. □

This proof is describing an algorithm to increment a binary counter from $n-1$ to n. Here's a more formal (and shorter!) description of the algorithm to add 1 to a binary counter. The input B is an (infinite) array of bits, where $B[i] = 1$ if and only if 2^i appears in the sum.

```
INCREMENT(B[0..∞]):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
```

We've already argued that INCREMENT must terminate, but how quickly? Obviously, the running time depends on the array of bits passed as input. If the first k bits are all 1s, then INCREMENT takes $\Theta(k)$ time. The binary representation of any positive integer n is exactly $\lfloor \lg n \rfloor + 1$ bits long. Thus, if B represents an integer between 0 and n, INCREMENT takes $\Theta(\log n)$ time in the worst case.

15.2 Counting from 0 to n

Now suppose we call INCREMENT n times, starting with a zero counter. How long does it take to count from 0 to n? If we only use the worst-case running time for each INCREMENT, we get an upper bound of $O(n \log n)$ on the total running time. Although this bound is correct, we can do better; in fact, the total running time is only $\Theta(n)$. This section describes several general methods for deriving, or at least proving, this linear time bound. Many (perhaps even all) of these methods are logically equivalent, but different formulations are more natural for different problems.

15.2.1 Summation

Perhaps the simplest way to derive a tighter bound is to observe that INCREMENT doesn't flip $\Theta(\log n)$ bits *every* time it is called. The least significant bit $B[0]$ does flip in every iteration, but $B[1]$ only flips every other iteration, $B[2]$ flips every 4th iteration, and in general, $B[i]$ flips every 2^ith iteration. Because we start with an array full of 0's, a sequence of n INCREMENTS flips each bit $B[i]$ exactly $\lfloor n/2^i \rfloor$ times. Thus, the total number of bit-flips for the entire sequence is

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

(More precisely, the number of flips is exactly $2n - \#1(n)$, where $\#1(n)$ is the number of 1 bits in the binary representation of n.) Thus, *on average*, each call to INCREMENT flips just less than two bits, and therefore runs in constant time.

This sense of "on average" is quite different from the averaging we consider with randomized algorithms. There is no probability involved; we are averaging over a sequence of operations, not the possible running times of a single operation. This averaging idea is called ***amortization***—the ***amortized*** time for each INCREMENT is $O(1)$. Amortization is a ~~sleazy~~ clever trick used by accountants to average large one-time costs over long periods of time; the most common example is calculating uniform payments for a loan, even though the borrower is paying interest on less and less capital over time. For this reason, it is common to use "*cost*" as a synonym for running time in the context of amortized analysis. Thus, the worst-case *cost* of INCREMENT is $O(\log n)$, but the amortized *cost* is only $O(1)$.

Most textbooks call this particular technique "the aggregate method", or "aggregate analysis", but these are just fancy names for computing the total cost of all operations and then dividing by the number of operations.

> **The Summation Method.** *Let $T(n)$ be the worst-case running time for a sequence of n operations. The amortized time for each operation is $T(n)/n$.*

15.2.2 Taxation

A second method we can use to derive amortized bounds is called either the *accounting* method or the *taxation* method. Suppose it costs us a dollar to toggle a bit, so we can measure the running time of our algorithm in dollars. Time is money!

Instead of paying for each bit flip when it happens, the Increment Revenue Service charges a two-dollar *increment tax* whenever we want to set a bit from zero to one. One of those dollars is spent changing the bit from zero to one; the other is stored away as *credit* until we need to reset the same bit to zero. The key point here is that we always have enough credit saved up to pay for the next INCREMENT. The amortized cost of an INCREMENT is the total tax it incurs, which is exactly 2 dollars, since each INCREMENT changes just one bit from 0 to 1.

It is often useful to distribute the tax income to specific pieces of the data structure. For example, for each INCREMENT, we could store one of the two dollars on the single bit that is set for 0 to 1, so that *that* bit can pay to reset itself back to zero later on.

> **Taxation Method 1.** *Certain steps in the algorithm charge you taxes, so that the total cost incurred by the algorithm is never more than the total tax you pay. The amortized cost of an operation is the overall tax charged to you during that operation.*

A different way to schedule the taxes is for *every* bit to charge us a tax at *every* operation, regardless of whether the bit changes of not. Specifically, each bit $B[i]$ charges a tax of $1/2^i$ dollars for each INCREMENT. The total tax we are charged during each INCREMENT is $\sum_{i\geq 0} 2^{-i} = 2$ dollars. Every time a bit $B[i]$ actually needs to be flipped, it has collected exactly \$1, which is just enough for us to pay for the flip.

> **Taxation Method 2.** *Certain portions of the data structure charge you taxes at each operation, so that the total cost of maintaining the data structure is never more than the total taxes you pay. The amortized cost of an operation is the overall tax you pay during that operation.*

In both of the taxation methods, our task as algorithm analysts is to come up with an appropriate 'tax schedule'. Different 'schedules' can result in different amortized time bounds. The tightest bounds are obtained from tax schedules that *just barely* stay in the black.

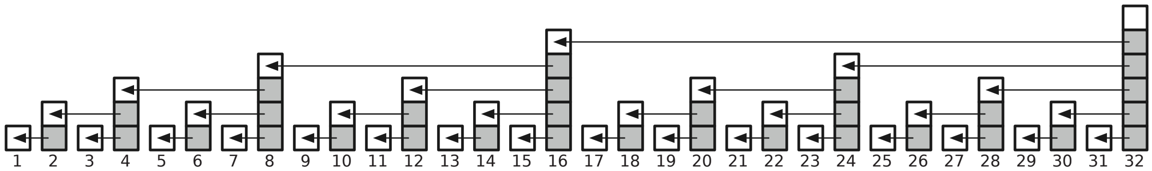
15.2.3 Charging

Another common method of amortized analysis involves *charging* the cost of some steps to some other, earlier steps. The method is similar to taxation, except that we focus on where each unit of tax is (or will be) *spent*, rather than where is it *collected*. By charging the cost of some operations to earlier operations, we are overestimating the total cost of any sequence of operations, since we pay for some charges from future operations that may never actually occur.

> **The Charging Method.** *Charge the cost of some steps of the algorithm to earlier steps, or to steps in some earlier operation. The amortized cost of the algorithm is its actual running time, minus its total charges* ***to*** *past operations, plus its total charge* ***from*** *future operations.*

For example, in our binary counter, suppose we charge the cost of clearing a bit (changing its value from 1 to 0) to the previous operation that sets that bit (changing its value from 0 to 1). If we flip k bits during an INCREMENT, we charge $k-1$ of those bit-flips to earlier bit-flips. Conversely, the single operation that sets a bit receives at most one unit of charge from the next time that bit is cleared. So instead of paying for k bit-flips, we pay for at most two: one for actually setting a bit, plus at most one charge from the future for clearing that same bit. Thus, the total amortized cost of the INCREMENT is at most two bit-flips.

We can visualize this charging scheme as follows. For each integer i, we represent the running time of the ith INCREMENT as a stack of blocks, one for each bit flip. The jth block in the ith stack is white if the ith INCREMENT changes $B[j]$ from 0 to 1, and shaded if the ith INCREMENT changes $B[j]$ from 1 to 0. If we moved each shaded block onto the white block directly to its left, there would at most two blocks in each stack.



Charging scheme for a binary counter.

15.2.4 Potential

The most powerful method (and the hardest to use) builds on a physics metaphor of 'potential energy'. Instead of associating costs or taxes with particular operations or pieces of the data structure, we represent prepaid work as *potential* that can be spent on later operations. The potential is a function of the entire data structure.

Let D_i denote our data structure after i operations have been performed, and let Φ_i denote its potential. Let c_i denote the actual cost of the ith operation (which changes D_{i-1} into D_i). Then the *amortized* cost of the ith operation, denoted a_i, is defined to be the actual cost plus the increase in potential:

$$\boxed{a_i = c_i + \Phi_i - \Phi_{i-1}}$$

So the *total* amortized cost of n operations is the actual total cost plus the total increase in potential:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

A potential function is *valid* if $\Phi_i - \Phi_0 \geq 0$ for all i. If the potential function is valid, then the total *actual* cost of any sequence of operations is always less than the total *amortized* cost:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i.$$

For our binary counter example, we can define the potential Φ_i after the ith INCREMENT to be the number of bits with value 1. Initially, all bits are equal to zero, so $\Phi_0 = 0$, and clearly $\Phi_i > 0$ for all $i > 0$, so this is a valid potential function. We can describe both the actual cost of an INCREMENT and the change in potential in terms of the number of bits set to 1 and reset to 0.

$$\begin{aligned} c_i &= \#\text{bits changed from 0 to 1} + \#\text{bits changed from 1 to 0} \\ \Phi_i - \Phi_{i-1} &= \#\text{bits changed from 0 to 1} - \#\text{bits changed from 1 to 0} \end{aligned}$$

Thus, the amortized cost of the ith INCREMENT is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times \#\text{bits changed from 0 to 1}$$

Since INCREMENT changes only *one* bit from 0 to 1, the amortized cost INCREMENT is 2.

> **The Potential Method.** *Define a potential function for the data structure that is initially equal to zero and is always non-negative. The amortized cost of an operation is its actual cost plus the change in potential.*

For this particular example, the potential is precisely the total unspent taxes paid using the taxation method, so it should be no surprise that we obtain precisely the same amortized cost. In general, however, there may be no natural way to interpret change in potential as "taxes" or "charges". Taxation and charging are useful when there is a convenient way to distribute costs to specific steps in the algorithm or components of the data structure. Potential arguments allow us to argue more globally when a local distribution is difficult or impossible.

Different potential functions can lead to different amortized time bounds. The trick to using the potential method is to come up with the best possible potential function. A good potential function goes up a little during any cheap/fast operation, and goes down a lot during any expensive/slow operation. Unfortunately, there is no general technique for finding good potential functions, except to play around with the data structure and try lots of possibilities (most of which won't work).

15.3 Incrementing and Decrementing

Now suppose we wanted a binary counter that we can both increment and decrement efficiently. A standard binary counter won't work, even in an amortized sense; if we alternate between 2^k and $2^k - 1$, *every* operation costs $\Theta(k)$ time.

A nice alternative is represent each integer as a *pair* (P, N) of bit strings, subject to the invariant $\boldsymbol{P \wedge N = 0}$ where $\wedge$ represents bit-wise AND. In other words,

For every index i, at most one of the bits $P[i]$ and $N[i]$ is equal to 1.

If we interpret P and N as binary numbers, the actual value of the counter is $P - N$; thus, intuitively, P represents the "positive" part of the pair, and N represents the "negative" part. Unlike the standard binary representation, this new representation is not unique, except for zero, which can only be represented by the pair $(0, 0)$. In fact, every positive or negative integer can be represented has an *infinite* number of distinct representations.

We can increment and decrement our double binary counter as follows. Intuitively, the INCREMENT algorithm increments P, and the DECREMENT algorithm increments N; however, in both cases, we must change the increment algorithm slightly to maintain the invariant $P \wedge N = 0$.

```
INCREMENT(P,N):
  i ← 0
  while P[i] = 1
      P[i] ← 0
      i ← i + 1
  if N[i] = 1
      N[i] ← 0
  else
      P[i] ← 1
```

```
DECREMENT(P,N):
  i ← 0
  while N[i] = 1
      N[i] ← 0
      i ← i + 1
  if P[i] = 1
      P[i] ← 0
  else
      N[i] ← 1
```

$$
\begin{array}{ccccccccccccc}
P = 10001 & & P = 10010 & & P = 10011 & & P = 10000 & & P = 10000 & & P = 10000 & & P = 10001 \\
N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 01000 & \xrightarrow{--} & N = 01001 & \xrightarrow{--} & N = 01010 & \xrightarrow{++} & N = 01010 \\
P - N = 5 & & P - N = 6 & & P - N = 7 & & P - N = 8 & & P - N = 7 & & P - N = 6 & & P - N = 7
\end{array}
$$

Incrementing and decrementing a double-binary counter.

Now suppose we start from $(0, 0)$ and apply a sequence of n INCREMENTs and DECREMENTs. In the worst case, each operation takes $\Theta(\log n)$ time, but what is the amortized cost? We can't use the aggregate method here, because we don't know what the sequence of operations looks like.

What about taxation? It's not hard to prove (by induction, of course) that after either $P[i]$ or $N[i]$ is set to 1, there must be at least 2^i operations, either INCREMENTS or DECREMENTS, before that bit is reset to 0. So if each bit $P[i]$ and $N[i]$ pays a tax of 2^{-i} at each operation, we will always have enough money to pay for the next operation. Thus, the amortized cost of each operation is at most $\sum_{i\geq 0} 2 \cdot 2^{-i} = 4$.

We can get even better amortized time bounds using the potential method. Define the potential Φ_i to be the number of 1-bits in both P and N after i operations. Just as before, we have

$$\begin{aligned} c_i &= \#\text{bits changed from 0 to 1} + \#\text{bits changed from 1 to 0} \\ \Phi_i - \Phi_{i-1} &= \#\text{bits changed from 0 to 1} - \#\text{bits changed from 1 to 0} \\ \Longrightarrow \qquad a_i &= 2 \times \#\text{bits changed from 0 to 1} \end{aligned}$$

Since each operation changes *at most* one bit to 1, the ith operation has amortized cost $a_i \leq 2$.

*15.4 Gray Codes

An attractive alternate solution to the increment/decrement problem was independently suggested by several students. *Gray codes* (named after Frank Gray, who discovered them in the 1950s) are methods for representing numbers as bit strings so that successive numbers differ by only one bit. For example, here is the four-bit *binary reflected* Gray code for the integers 0 through 15:

$$0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$

The general rule for incrementing a binary reflected Gray code is to invert the bit that would be set from 0 to 1 by a normal binary counter. In terms of bit-flips, this is the perfect solution; each increment of decrement *by definition* changes only one bit. Unfortunately, the naïve algorithm to *find* the single bit to flip still requires $\Theta(\log n)$ time in the worst case. Thus, so the total cost of maintaining a Gray code, using the obvious algorithm, is the same as that of maintaining a normal binary counter.

Fortunately, this is only true of the naïve algorithm. The following algorithm, discovered by Gideon Ehrlich[1] in 1973, maintains a Gray code counter in constant *worst-case* time per increment! The algorithm uses a separate 'focus' array $F[0..n]$ in addition to a Gray-code bit array $G[0..n-1]$.

```
EhrlichGrayInit(n):
  for i ← 0 to n − 1
      G[i] ← 0
  for i ← 0 to n
      F[i] ← i
```

```
EhrlichGrayIncrement(n):
  j ← F[0]
  F[0] ← 0
  if j = n
      G[n − 1] ← 1 − G[n − 1]
  else
      G[j] = 1 − G[j]
      F[j] ← F[j + 1]
      F[j + 1] ← j + 1
```

[1] Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. Assoc. Comput. Mach.* 20:500–513, 1973.

The EHRLICHGRAYINCREMENT algorithm obviously runs in $O(1)$ time, even in the worst case. Here's the algorithm in action with $n = 4$. The first line is the Gray bit-vector G, and the second line shows the focus vector F, both in reverse order:

$$G : 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$
$$F : 3210, 3211, 3220, 3212, 3310, 3311, 3230, 3213, 4210, 4211, 4220, 4212, 3410, 3411, 3240, 3214$$

Voodoo! I won't explain in detail how Ehrlich's algorithm works, except to point out the following invariant. Let $B[i]$ denote the ith bit in the *standard* binary representation of the current number. ***If $B[j] = 0$ and $B[j-1] = 1$, then $F[j]$ is the smallest integer $k > j$ such that $B[k] = 1$; otherwise, $F[j] = j$.*** Got that?

But wait — this algorithm only handles increments; what if we also want to decrement? Sorry, I don't have a clue. Extra credit, anyone?

15.5 Generalities and Warnings

Although computer scientists usually apply amortized analysis to understand the efficiency of maintaining and querying data structures, you should remember that amortization can be applied to *any* sequence of numbers. Banks have been using amortization to calculate fixed payments for interest-bearing loans for centuries. The IRS allows taxpayers to amortize business expenses or gambling losses across several years for purposes of computing income taxes. Some cell phone contracts let you to apply amortization to calling time, by rolling unused minutes from one month into the next month.

It's also important to remember that ***amortized time bounds are not unique***. For a data structure that supports multiple operations, different amortization schemes can assign different costs to *exactly the same* algorithms. For example, consider a generic data structure that can be modified by three algorithms: FOLD, SPINDLE, and MUTILATE. One amortization scheme might imply that FOLD and SPINDLE each run in $O(\log n)$ amortized time, while MUTILATE runs in $O(n)$ amortized time. Another scheme might imply that FOLD runs in $O(\sqrt{n})$ amortized time, while SPINDLE and MUTILATE each run in $O(1)$ amortized time. These two results are not necessarily inconsistent! Moreover, there is no general reason to prefer one of these sets of amortized time bounds over the other; our preference may depend on the context in which the data structure is used.

Exercises

1. Suppose we are maintaining a data structure under a series of n operations. Let $f(k)$ denote the actual running time of the kth operation. For each of the following functions f, determine the resulting amortized cost of a single operation. (For practice, try *all* of the methods described in this note.)

   (a) $f(k)$ is the largest integer i such that 2^i divides k.

   (b) $f(k)$ is the largest power of 2 that divides k.

   (c) $f(k) = n$ if k is a power of 2, and $f(k) = 1$ otherwise.

   (d) $f(k) = n^2$ if k is a power of 2, and $f(k) = 1$ otherwise.

   (e) $f(k)$ is the index of the largest Fibonacci number that divides k.

   (f) $f(k)$ is the largest Fibonacci number that divides k.

(g) $f(k) = k$ if k is a Fibonacci number, and $f(k) = 1$ otherwise.

(h) $f(k) = k^2$ if k is a Fibonacci number, and $f(k) = 1$ otherwise.

(i) $f(k)$ is the largest integer whose square divides k.

(j) $f(k)$ is the largest perfect square that divides k.

(k) $f(k) = k$ if k is a perfect square, and $f(k) = 1$ otherwise.

(l) $f(k) = k^2$ if k is a perfect square, and $f(k) = 1$ otherwise.

(m) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k)$ is the number of ancestors of node k.

(n) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k)$ is the number of descendants of node k.

(o) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k)$ is the *square* of the number of ancestors of node k.

(p) Let T be a *complete* binary search tree, storing the integer keys 1 through n. $f(k) = \mathit{size}(k) \lg \mathit{size}(k)$, where $\mathit{size}(k)$ is the number of descendants of node k.

(q) Let T be an *arbitrary* binary search tree, storing the integer keys 0 through n. $f(k)$ is the length of the path in T from node $k-1$ to node k.

(r) Let T be an *arbitrary* binary search tree, storing the integer keys 0 through n. $f(k)$ is the *square* of the length of the path in T from node $k-1$ to node k.

(s) Let T be a *complete* binary search tree, storing the integer keys 0 through n. $f(k)$ is the *square* of the length of the path in T from node $k-1$ to node k.

2. Consider the following modification of the standard algorithm for incrementing a binary counter.

```
INCREMENT(B[0..∞]):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
  SOMETHINGELSE(i)
```

The only difference from the standard algorithm is the function call at the end, to a black-box subroutine called SomethingElse.

Suppose we call Increment n times, starting with a counter with value 0. The amortized time of each Increment clearly depends on the running time of SomethingElse. Let $T(i)$ denote the worst-case running time of SomethingElse(i). For example, we proved in class that Increment algorithm runs in $O(1)$ amortized time when $T(i) = 0$.

(a) What is the amortized time per Increment if $T(i) = 42$?

(b) What is the amortized time per Increment if $T(i) = 2^i$?

(c) What is the amortized time per Increment if $T(i) = 4^i$?

(d) What is the amortized time per Increment if $T(i) = \sqrt{2}^i$?

(e) What is the amortized time per Increment if $T(i) = 2^i/(i+1)$?

3. An ***extendable array*** is a data structure that stores a sequence of items and supports the following operations.

   - AddToFront(x) adds x to the *beginning* of the sequence.
   - AddToEnd(x) adds x to the *end* of the sequence.
   - Lookup(k) returns the kth item in the sequence, or Null if the current length of the sequence is less than k.

   Describe a ***simple*** data structure that implements an extendable array. Your AddToFront and AddToBack algorithms should take $O(1)$ *amortized* time, and your Lookup algorithm should take $O(1)$ *worst-case* time. The data structure should use $O(n)$ space, where n is the ***current*** length of the sequence.

4. An ***ordered stack*** is a data structure that stores a sequence of items and supports the following operations.

   - OrderedPush(x) removes all items smaller than x from the beginning of the sequence and then adds x to the beginning of the sequence.
   - Pop deletes and returns the first item in the sequence (or Null if the sequence is empty).

   Suppose we implement an ordered stack with a simple linked list, using the obvious OrderedPush and Pop algorithms. Prove that if we start with an empty data structure, the amortized cost of each OrderedPush or Pop operation is $O(1)$.

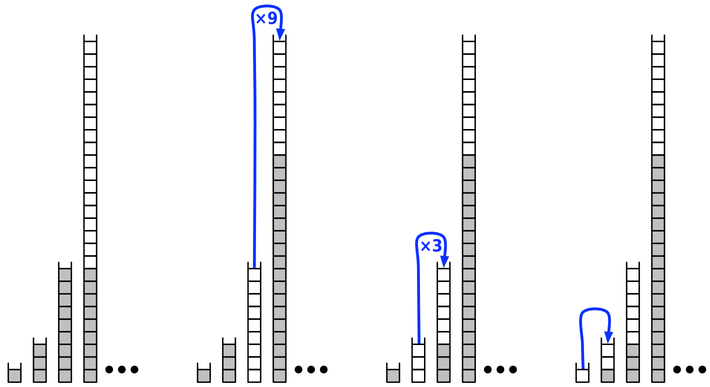
5. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. The user always pushes and pops elements from the smallest stack S_0. However, before any element can be pushed onto any full stack S_i, we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move all its members to S_{i+2}.) Similarly, before any element can be popped from any empty stack S_i, we first pop 3^i elements from S_{i+1} and push them onto S_i to make room. (Thus, if S_{i+1} is already empty, we first recursively fill it by popping elements from S_{i+2}.) Moving a single element from one stack to another takes $O(1)$ time.

   Here is pseudocode for the multistack operations MSPush and MSPop. The internal stacks are managed with the subroutines Push and Pop.

```
MPush(x):
  i ← 0
  while S_i is full
      i ← i + 1
  while i > 0
      i ← i − 1
      for j ← 1 to 3^i
          Push(S_{i+1}, Pop(S_i))
  Push(S_0, x)
```

```
MPop(x):
  i ← 0
  while S_i is empty
      i ← i + 1
  while i > 0
      i ← i − 1
      for j ← 1 to 3^i
          Push(S_i, Pop(S_{i+1}))
  return Pop(S_0)
```

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?



Making room in a multistack, just before pushing on a new element.

(b) Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack during its lifetime.

(c) Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where n is the maximum number of elements in the multistack during its lifetime.

6. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

   - Push(x): Add item x to the end of the sequence.
   - Pull(): Remove and return the item at the beginning of the sequence.

   It is easy to implement a queue using a doubly-linked list and a counter, so that the entire data structure uses $O(n)$ space (where n is the number of items in the queue) and the worst-case time per operation is $O(1)$.

   (a) Now suppose we want to support the following operation instead of Pull:

   - MultiPull(k): Remove the first k items from the front of the queue, and return the kth item removed.

   Suppose we use the obvious algorithm to implement MultiPull:

```
MultiPull(k):
  for i ← 1 to k
      x ← Pull()
  return x
```

   Prove that in any intermixed sequence of Push and MultiPull operations, the amortized cost of each operation is $O(1)$

   (b) Now suppose we *also* want to support the following operation instead of Push:

   - MultiPush(x, k): Insert k copies of x into the back of the queue.

   Suppose we use the obvious algorithm to implement MultiPuush:

```
MultiPush(k, x):
  for i ← 1 to k
      Push(x)
```

Prove that for any integers ℓ and n, there is a sequence of ℓ MultiPush and MultiPull operations that require $\Omega(n\ell)$ time, where n is the maximum number of items in the queue at any time. Such a sequence implies that the amortized cost of each operation is $\Omega(n)$.

(c) Describe a data structure that supports arbitrary intermixed sequences of MultiPush and MultiPull operations in $O(1)$ amortized cost per operation. Like a standard queue, your data structure should use only $O(1)$ space per item.

7. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

   - Push(x): Add item x to the end of the sequence.
   - Pull(): Remove and return the item at the beginning of the sequence.
   - Size(): Return the current number of items in the sequence.

   It is easy to implement a queue using a doubly-linked list, so that it uses $O(n)$ space (where n is the number of items in the queue) and the worst-case time for each of these operations is $O(1)$.

   Consider the following new operation, which removes every tenth element from the queue, starting at the beginning, in $\Theta(n)$ worst-case time.

```
Decimate():
  n ← Size()
  for i ← 0 to n − 1
      if i mod 10 = 0
          Pull()    ⟨⟨result discarded⟩⟩
      else
          Push(Pull())
```

   Prove that in any intermixed sequence of Push, Pull, and Decimate operations, the amortized cost of each operation is $O(1)$.

8. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array $A[1..n]$ that stores the height of n buildings on a city block, indexed from west to east. Building i has a good view of Lake Michigan if and only if every building to the east of i is shorter than i.

   Here is an algorithm that computes which buildings have a good view of Lake Michigan. What is the running time of this algorithm?

```
GoodView(A[1..n]):
  initialize a stack S
  for i ← 1 to n
      while (S not empty and A[i] > A[Top(S)])
          Pop(S)
      Push(S, i)
  return S
```

9. Suppose we can insert or delete an element into a hash table in $O(1)$ time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. *[Hint: Do not use potential functions.]*

10. Professor Pisano insists that the size of any hash table used in his class must always be a Fibonacci number. He insists on the following variant of the previous global rebuilding strategy. Suppose the current hash table has size F_k.

    - After an insertion, if the number of items in the table is F_{k-1}, we allocate a new hash table of size F_{k+1}, insert everything into the new table, and then free the old table.
    - After a deletion, if the number of items in the table is F_{k-3}, we allocate a new hash table of size F_{k-1}, insert everything into the new table, and then free the old table.

    Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. *[Hint: Do not use potential functions.]*

11. Remember the difference between stacks and queues? Good.

    (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard subroutines PUSH and POP.

    (b) A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

    - QUACKPUSH(x): add a new item x to the left end of the list;
    - QUACKPOP(): remove and return the item on the left end of the list;
    - QUACKPULL(): remove the item on the right end of the list.

    Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any QUACKPUSH, QUACKPOP, or QUACKPULL operation is $O(1)$. In particular, each element in the quack must be stored in *exactly one* of the three stacks. Again, you ***cannot*** access the component stacks except through the interface functions PUSH and POP.

12. Let's glom a whole bunch of earlier problems together. Yay! An ***random-access double-ended multi-queue*** or ***radmuque*** (pronounced "rad muck") stores a sequence of items and supports the following operations.

    - MULTIPUSH(x, k) adds k copies of item x to the *beginning* of the sequence.

- MULTIPOKE(x, k) adds k copies of item x to the *end* of the sequence.
- MULTIPOP(k) removes k items from the *beginning* of the sequence and retrns the last item removed. (If there are less than k items in the sequence, remove them all and return NULL.)
- MULTIPULL(k) removes k items from the *end* of the sequence and retrns the last item removed. (If there are less than k items in the sequence, remove them all and return NULL.)
- LOOKUP(k) returns the kth item in the sequence. (If there are less than k items in the sequence, return NULL.)

Describe and analyze a simple data structure that supports these operations using $O(n)$ space, where n is the current number of items in the sequence. LOOKUP should run in $O(1)$ *worst-case* time; all other operations should run in $O(1)$ *amortized* time.

13. Suppose you are faced with an infinite number of counters x_i, one for each integer i. Each counter stores an integer mod m, where m is a fixed global constant. All counters are initially zero. The following operation increments a single counter x_i; however, if x_i overflows (that is, wraps around from m to 0), the adjacent counters x_{i-1} and x_{i+1} are incremented recursively.

```
NUDGE_m(i):
  x_i ← x_i + 1
  while x_i ≥ m
      x_i ← x_i − m
      NUDGE_m(i − 1)
      NUDGE_m(i + 1)
```

(a) Prove that NUDGE$_3$ runs in $O(1)$ amortized time. *[Hint: Prove that NUDGE$_3$ always halts!]*

(b) What is the worst-case total time for n calls to NUDGE$_2$, if all counters are initially zero?

14. Now suppose you are faced with an infinite two-dimensional grid of modular counters, one counter $x_{i,j}$ for every pair of integers (i, j). Again, all counters are initially zero. The counters are modified by the following operation, where m is a fixed global constant:

```
2DNUDGE_m(i, j):
  x_{i,j} ← x_i + 1
  while x_{i,j} ≥ m
      x_{i,j} ← x_{i,j} − m
      2DNUDGE_m(i − 1, j)
      2DNUDGE_m(i, j + 1)
      2DNUDGE_m(i + 1, j)
      2DNUDGE_m(i, j − 1)
```

(a) Prove that 2DNUDGE$_5$ runs in $O(1)$ amortized time.

★(b) Prove or disprove: 2DNUDGE$_4$ also runs in $O(1)$ amortized time.

★(c) Prove or disprove: 2DNUDGE$_3$ always halts.

*15. Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of *fits*, where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fitstring 101110_F represents the number $F_6+F_4+F_3+F_2 = 8+3+2+1 = 14$. Describe algorithms to increment and decrement a single fitstring in constant amortized time. *[Hint: Most numbers can be represented by more than one fitstring!]*

*16. A *doubly lazy binary counter* represents any number as a weighted sum of powers of two, where each weight is one of *four* values: −1, 0, 1, or 2. (For succinctness, I'll write $\bar{1}$ instead of −1.) Every integer—positive, negative, or zero—has an infinite number of doubly lazy binary representations. For example, the number 13 can be represented as 1101 (the standard binary representation), or $2\bar{1}01$ (because $2 \cdot 2^3 - 2^2 + 2^0 = 13$) or $10\bar{1}1\bar{1}$ (because $2^4 - 2^2 + 2^1 - 2^0 = 13$) or $\bar{1}1200010\bar{1}1\bar{1}$ (because $-2^{10} + 2^9 + 2 \cdot 2^8 + 2^4 - 2^2 + 2^1 - 2^0 = 13$).

To increment a doubly lazy binary counter, we add 1 to the least significant bit, then carry the rightmost 2 (if any). To decrement, we subtract 1 from the lest significant bit, and then borrow the rightmost $\bar{1}$ (if any).

```
LazyIncrement(B[0..n]):
  B[0] ← B[0] + 1
  for i ← 1 to n − 1
      if B[i] = 2
          B[i] ← 0
          B[i+1] ← B[i+1] + 1
          return
```

```
LazyDecrement(B[0..n]):
  B[0] ← B[0] − 1
  for i ← 1 to n − 1
      if B[i] = −1
          B[i] ← 1
          B[i+1] ← B[i+1] − 1
          return
```

For example, here is a doubly lazy binary count from zero up to twenty and then back down to zero. The bits are written with the least significant bit $B[0]$ on the *right*, omitting all leading 0's.

$$0 \xrightarrow{++} 1 \xrightarrow{++} 10 \xrightarrow{++} 11 \xrightarrow{++} 20 \xrightarrow{++} 101 \xrightarrow{++} 110 \xrightarrow{++} 111 \xrightarrow{++} 120 \xrightarrow{++} 201 \xrightarrow{++} 210$$
$$\xrightarrow{++} 1011 \xrightarrow{++} 1020 \xrightarrow{++} 1101 \xrightarrow{++} 1110 \xrightarrow{++} 1111 \xrightarrow{++} 1120 \xrightarrow{++} 1201 \xrightarrow{++} 1210 \xrightarrow{++} 2011 \xrightarrow{++} 2020$$
$$\xrightarrow{--} 2011 \xrightarrow{--} 2010 \xrightarrow{--} 2001 \xrightarrow{--} 2000 \xrightarrow{--} 20\bar{1}1 \xrightarrow{--} 2\bar{1}10 \xrightarrow{--} 2\bar{1}01 \xrightarrow{--} 1100 \xrightarrow{--} 11\bar{1}1 \xrightarrow{--} 1010$$
$$\xrightarrow{--} 1001 \xrightarrow{--} 1000 \xrightarrow{--} 10\bar{1}1 \xrightarrow{--} 1\bar{1}10 \xrightarrow{--} 1\bar{1}01 \xrightarrow{--} 100 \xrightarrow{--} 1\bar{1}1 \xrightarrow{--} 10 \xrightarrow{--} 1 \xrightarrow{--} 0$$

Prove that for any intermixed sequence of increments and decrements of a doubly lazy binary number, starting with 0, the amortized time for each operation is $O(1)$. Do *not* assume, as in the example above, that all the increments come before all the decrements.

Everything was balanced before the computers went off line. Try and adjust something, and you unbalance something else. Try and adjust that, you unbalance two more and before you know what's happened, the ship is out of control.

— Blake, *Blake's 7*, "Breakdown" (March 6, 1978)

A good scapegoat is nearly as welcome as a solution to the problem.

— Anonymous

Let's play.

— El Mariachi [Antonio Banderas], *Desperado* (1992)

CAPTAIN: TAKE OFF EVERY 'ZIG'!!
CAPTAIN: YOU KNOW WHAT YOU DOING.
CAPTAIN: MOVE 'ZIG'.
CAPTAIN: FOR GREAT JUSTICE.

— *Zero Wing* (1992)

16 Scapegoat and Splay Trees

16.1 Definitions

★★★ Move intro paragraphs to earlier treap notes, or maybe to new appendix on basic data structures (arrays, stacks, queues, heaps, binary search trees).

I'll assume that everyone is already familiar with the standard terminology for binary search trees—node, search key, edge, root, internal node, leaf, right child, left child, parent, descendant, sibling, ancestor, subtree, preorder, postorder, inorder, etc.—as well as the standard algorithms for searching for a node, inserting a node, or deleting a node. Otherwise, consult your favorite data structures textbook.

For this lecture, we will consider only *full* binary trees—where every internal node has *exactly* two children—where only the *internal* nodes actually store search keys. In practice, we can represent the leaves with null pointers.

Recall that the *depth* of a node is its distance from the root, and its *height* is the distance to the farthest leaf in its subtree. The height (or depth) of the tree is just the height of the root. The *size* of a node is the number of nodes in its subtree. The size n of the whole tree is just the total number of nodes.

A tree with height h has at most 2^h leaves, so the minimum height of an n-leaf binary tree is $\lceil \lg n \rceil$. In the worst case, the time required for a search, insertion, or deletion to the height of the tree, so in general we would like keep the height as close to $\lg n$ as possible. The best we can possibly do is to have a *perfectly balanced* tree, in which each subtree has as close to half the leaves as possible, and both subtrees are perfectly balanced. The height of a perfectly balanced tree is $\lceil \lg n \rceil$, so the worst-case search time is $O(\log n)$. However, even if we started with a perfectly balanced tree, a malicious sequence of insertions and/or deletions could make the tree arbitrarily unbalanced, driving the search time up to $\Theta(n)$.

To avoid this problem, we need to periodically modify the tree to maintain 'balance'. There are several methods for doing this, and depending on the method we use, the search tree is given a different name. Examples include AVL trees, red-black trees, height-balanced trees, weight-balanced trees, bounded-balance trees, path-balanced trees, B-trees, treaps, randomized binary search trees, skip lists,[1] and jumplists. Some of these trees support searches, insertions, and deletions, in $O(\log n)$ *worst-case* time, others in $O(\log n)$ *amortized* time, still others in $O(\log n)$ *expected* time.

In this lecture, I'll discuss three binary search tree data structures with good *amortized* performance. The first two are variants of *lazy* balanced trees: *lazy weight-balanced trees*, developed by Mark Overmars* in the early 1980s, [14] and *scapegoat trees*, discovered by Arne Andersson* in 1989 [1, 2] and independently[2] by Igal Galperin* and Ron Rivest in 1993 [11]. The third structure is the *splay tree*, discovered by Danny Sleator and Bob Tarjan in 1981 [19, 16].

16.2 Lazy Deletions: Global Rebuilding

First let's consider the simple case where we start with a perfectly-balanced tree, and we only want to perform searches and deletions. To get good search and delete times, we can use a technique called *global rebuilding*. When we get a delete request, we locate and mark the node to be deleted, *but we don't actually delete it*. This requires a simple modification to our search algorithm—we still use marked nodes to guide searches, but if we search for a marked node, the search routine says it isn't there. This keeps the tree more or less balanced, but now the search time is no longer a function of the amount of data currently stored in the tree. To remedy this, we also keep track of how many nodes have been marked, and then apply the following rule:

> **Global Rebuilding Rule.** *As soon as half the nodes in the tree have been marked, rebuild a new perfectly balanced tree containing only the unmarked nodes.*[3]

With this rule in place, a search takes $O(\log n)$ time in the worst case, where n is the number of unmarked nodes. Specifically, since the tree has at most n marked nodes, or $2n$ nodes altogether, we need to examine at most $\lg n + 1$ keys. There are several methods for rebuilding the tree in $O(n)$ time, where n is the size of the new tree. (Homework!) So a single deletion can cost $\Theta(n)$ time in the worst case, but only if we have to rebuild; most deletions take only $O(\log n)$ time.

We spend $O(n)$ time rebuilding, but only after $\Omega(n)$ deletions, so the *amortized* cost of rebuilding the tree is $O(1)$ per deletion. (Here I'm using a simple version of the 'taxation method'. For each deletion, we charge a \$1 tax; after n deletions, we've collected \$$n$, which is just enough to pay for rebalancing the tree containing the remaining n nodes.) Since we also have to find and mark the node being 'deleted', the total amortized time for a deletion is $\boldsymbol{O(\log n)}$.

16.3 Insertions: Partial Rebuilding

Now suppose we only want to support searches and insertions. We can't 'not really insert' new nodes into the tree, since that would make them unavailable to the search algorithm.[4] So instead, we'll use another method called *partial rebuilding*. We will insert new nodes normally, but whenever a *subtree* becomes unbalanced enough, we rebuild it. The definition of 'unbalanced enough' depends on an arbitrary constant $\alpha > 1$.

Each node v will now also store *height*(v) and *size*(v). We now modify our insertion algorithm with the following rule:

---

[1] Yeah, yeah. Skip lists aren't really binary search trees. Whatever you say, Mr. Picky.

[2] The claim of independence is Andersson's [2]. The two papers actually describe very slightly different rebalancing algorithms. The algorithm I'm using here is closer to Andersson's, but my analysis is closer to Galperin and Rivest's.

[3] Alternately: When the number of unmarked nodes is one less than an exact power of two, rebuild the tree. This rule ensures that the tree is always *exactly* balanced.

[4] Well, we could use the Bentley-Saxe* logarithmic method [3], but that would raise the query time to $O(\log^2 n)$.

> **Partial Rebuilding Rule.** *After we insert a node, walk back up the tree updating the heights and sizes of the nodes on the search path. If we encounter a node v where $height(v) > \alpha \cdot \lg(size(v))$, rebuild its subtree into a perfectly balanced tree (in $O(size(v))$ time).*

If we always follow this rule, then after an insertion, the height of the tree is at most $\alpha \cdot \lg n$. Thus, since α is a constant, the worst-case search time is $O(\log n)$. In the worst case, insertions require $\Theta(n)$ time—we might have to rebuild the entire tree. However, the *amortized* time for each insertion is again only $O(\log n)$. Not surprisingly, the proof is a little bit more complicated than for deletions.

Define the *imbalance* $I(v)$ of a node v to be the absolute difference between the sizes of its two subtrees:

$$Imbal(v) := |size(left(v)) - size(right(v))|$$

A simple induction proof implies that $Imbal(v) \leq 1$ for every node v in a perfectly balanced tree. In particular, immediately after we rebuild the subtree of v, we have $Imbal(v) \leq 1$. On the other hand, each insertion into the subtree of v increments either $size(left(v))$ or $size(right(v))$, so $Imbal(v)$ changes by at most 1.

The whole analysis boils down to the following lemma.

Lemma 1. *Just before we rebuild v's subtree, $Imbal(v) = \Omega(size(v))$.*

Before we prove this lemma, let's first look at what it implies. If $Imbal(v) = \Omega(size(v))$, then $\Omega(size(v))$ keys have been inserted in the v's subtree since the last time it was rebuilt from scratch. On the other hand, rebuilding the subtree requires $O(size(v))$ time. Thus, if we amortize the rebuilding cost across all the insertions since the previous rebuild, v is charged *constant* time for each insertion into its subtree. Since each new key is inserted into at most $\alpha \cdot \lg n = O(\log n)$ subtrees, the total amortized cost of an insertion is $\boldsymbol{O(\log n)}$.

Proof: Since we're about to rebuild the subtree at v, we must have $height(v) > \alpha \cdot \lg size(v)$. Without loss of generality, suppose that the node we just inserted went into v's left subtree. Either we just rebuilt this subtree or we didn't have to, so we also have $height(left(v)) \leq \alpha \cdot \lg size(left(v))$. Combining these two inequalities with the recursive definition of height, we get

$$\alpha \cdot \lg size(v) \;<\; height(v) \;\leq\; height(left(v)) + 1 \;\leq\; \alpha \cdot \lg size(left(v)) + 1.$$

After some algebra, this simplifies to $size(left(v)) > size(v)/2^{1/\alpha}$. Combining this with the identity $size(v) = size(left(v)) + size(right(v)) + 1$ and doing some more algebra gives us the inequality

$$size(right(v)) < \left(1 - 1/2^{1/\alpha}\right) size(v) - 1.$$

Finally, we combine these two inequalities using the recursive definition of imbalance.

$$Imbal(v) \;\geq\; size(left(v)) - size(right(v)) - 1 \;>\; \left(2/2^{1/\alpha} - 1\right) size(v)$$

Since α is a constant bigger than 1, the factor in parentheses is a positive constant. □

16.4 Scapegoat (Lazy Height-Balanced) Trees

Finally, to handle both insertions and deletions efficiently, *scapegoat trees* use both of the previous techniques. We use partial rebuilding to re-balance the tree after insertions, and global rebuilding to re-balance the tree after deletions. Each search takes $O(\log n)$ time in the worst case, and the amortized time for any insertion or deletion is also $O(\log n)$. There are a few small technical details left (which I won't describe), but no new ideas are required.

Once we've done the analysis, we can actually simplify the data structure. It's not hard to prove that at most one subtree (the *scapegoat*) is rebuilt during any insertion. Less obviously, we can even get the same amortized time bounds (except for a small constant factor) if we only maintain the three integers in addition to the actual tree: the size of the entire tree, the height of the entire tree, and the number of marked nodes. Whenever an insertion causes the tree to become unbalanced, we can compute the sizes of all the subtrees on the search path, starting at the new leaf and stopping at the scapegoat, in time proportional to the size of the scapegoat subtree. Since we need that much time to re-balance the scapegoat subtree, this computation increases the running time by only a small constant factor! Thus, unlike almost every other kind of balanced trees, scapegoat trees require only $O(1)$ extra space.

16.5 Rotations, Double Rotations, and Splaying

Another method for maintaining balance in binary search trees is by adjusting the shape of the tree locally, using an operation called a *rotation*. A rotation at a node x decreases its depth by one and increases its parent's depth by one. Rotations can be performed in constant time, since they only involve simple pointer manipulation.

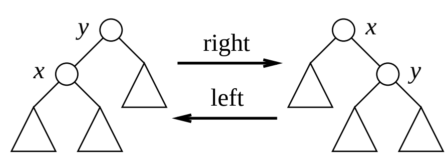


Figure 1. A right rotation at x and a left rotation at y are inverses.

For technical reasons, we will need to use rotations two at a time. There are two types of double rotations, which might be called *zig-zag* and *roller-coaster*. A zig-zag at x consists of two rotations at x, in opposite directions. A roller-coaster at a node x consists of a rotation at x's parent followed by a rotation at x, both in the same direction. Each double rotation decreases the depth of x by two, leaves the depth of its parent unchanged, and increases the depth of its grandparent by either one or two, depending on the type of double rotation. Either type of double rotation can be performed in constant time.

Finally, a *splay* operation moves an arbitrary node in the tree up to the root through a series of double rotations, possibly with one single rotation at the end. Splaying a node v requires time proportional to $depth(v)$. (Obviously, this means the depth *before* splaying, since after splaying v is the root and thus has depth zero!)

16.6 Splay Trees

A *splay tree* is a binary search tree that is kept more or less balanced by splaying. Intuitively, after we access any node, we move it to the root with a splay operation. In more detail:

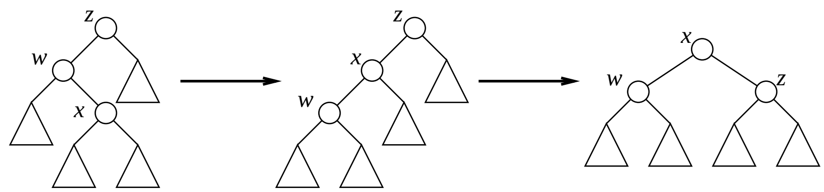


Figure 2. A zig-zag at x. The symmetric case is not shown.

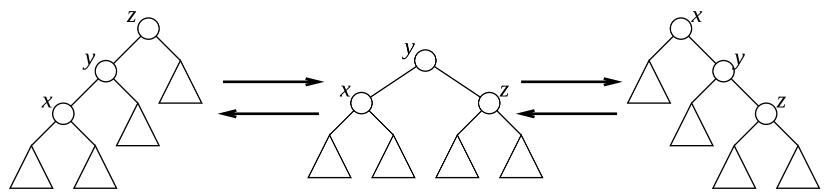


Figure 3. A right roller-coaster at x and a left roller-coaster at z.

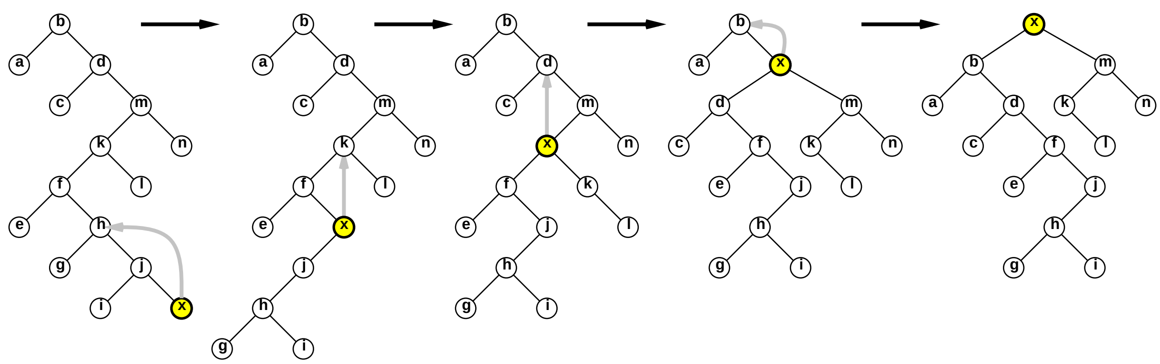


Figure 4. Splaying a node. Irrelevant subtrees are omitted for clarity.

- **Search:** Find the node containing the key using the usual algorithm, or its predecessor or successor if the key is not present. Splay whichever node was found.
- **Insert:** Insert a new node using the usual algorithm, then splay that node.
- **Delete:** Find the node x to be deleted, splay it, and then delete it. This splits the tree into two subtrees, one with keys less than x, the other with keys bigger than x. Find the node w in the left subtree with the largest key (the inorder predecessor of x in the original tree), splay it, and finally join it to the right subtree.

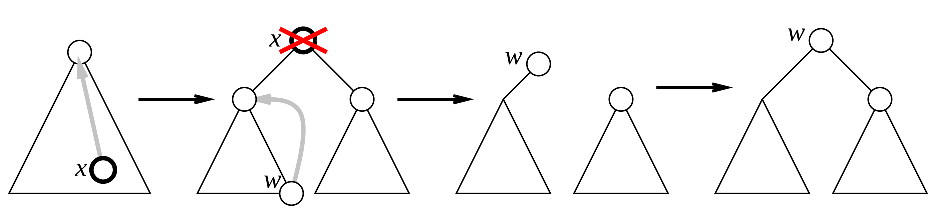


Figure 5. Deleting a node in a splay tree.

Each search, insertion, or deletion consists of a constant number of operations of the form *walk down to a node, and then splay it up to the root*. Since the walk down is clearly cheaper than the splay up, all we need to get good amortized bounds for splay trees is to derive good amortized bounds for a single splay.

Believe it or not, the easiest way to do this uses the potential method. We define the *rank* of a node v to be $\lfloor \lg \mathit{size}(v) \rfloor$, and the *potential* of a splay tree to be the sum of the ranks of its nodes:

$$\Phi := \sum_v \mathit{rank}(v) = \sum_v \lfloor \lg \mathit{size}(v) \rfloor$$

It's not hard to observe that a perfectly balanced binary tree has potential $\Theta(n)$, and a linear chain of nodes (a perfectly *unbalanced* tree) has potential $\Theta(n \log n)$.

The amortized analysis of splay trees boils down to the following lemma. Here, $\mathit{rank}(v)$ denotes the rank of v before a (single or double) rotation, and $\mathit{rank}'(v)$ denotes its rank afterwards. Recall that the amortized cost is defined to be the number of rotations plus the drop in potential.

The Access Lemma. *The amortized cost of a single rotation at any node v is at most $1 + 3\,\mathit{rank}'(v) - 3\,\mathit{rank}(v)$, and the amortized cost of a double rotation at any node v is at most $3\,\mathit{rank}'(v) - 3\,\mathit{rank}(v)$.*

Proving this lemma is a straightforward but tedious case analysis of the different types of rotations. For the sake of completeness, I'll give a proof (of a generalized version) in the next section.

By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node v is at most $1 + 3\,\mathit{rank}'(v) - 3\,\mathit{rank}(v)$, where $\mathit{rank}'(v)$ is the rank of v after the entire splay. (The intermediate ranks cancel out in a nice telescoping sum.) But after the splay, v is the root! Thus, $\mathit{rank}'(v) = \lfloor \lg n \rfloor$, which implies that the amortized cost of a splay is at most $3 \lg n - 1 = O(\log n)$.

We conclude that every insertion, deletion, or search in a splay tree takes $O(\log n)$ amortized time.

*16.7 Other Optimality Properties

In fact, splay trees are optimal in several other senses. Some of these optimality properties follow easily from the following generalization of the Access Lemma.

Let's arbitrarily assign each node v a non-negative real *weight* $w(v)$. These weights are not actually stored in the splay tree, nor do they affect the splay algorithm in any way; they are only used to help with the analysis. We then redefine the *size* $s(v)$ of a node v to be the sum of the weights of the descendants of v, including v itself:

$$s(v) := w(v) + s(\mathit{right}(v)) + s(\mathit{left}(v)).$$

If $w(v) = 1$ for every node v, then the size of a node is just the number of nodes in its subtree, as in the previous section. As before, we define the *rank* of any node v to be $r(v) = \lg s(v)$, and the *potential* of a splay tree to be the sum of the ranks of all its nodes:

$$\Phi = \sum_v r(v) = \sum_v \lg s(v)$$

In the following lemma, $r(v)$ denotes the rank of v before a (single or double) rotation, and $r'(v)$ denotes its rank afterwards.

The Generalized Access Lemma. *For **any** assignment of non-negative weights to the nodes, the amortized cost of a single rotation at any node x is at most $1+3r'(x)-3r(x)$, and the amortized cost of a double rotation at any node v is at most $3r'(x)-3r(x)$.*

Proof: First consider a single rotation, as shown in Figure 1.

$$\begin{aligned} 1+\Phi'-\Phi &= 1+r'(x)+r'(y)-r(x)-r(y) && \text{[only } x \text{ and } y \text{ change rank]} \\ &\le 1+r'(x)-r(x) && [r'(y)\le r(y)] \\ &\le 1+3r'(x)-3r(x) && [r'(x)\ge r(x)] \end{aligned}$$

Now consider a zig-zag, as shown in Figure 2. Only w, x, and z change rank.

$$\begin{aligned} &2+\Phi'-\Phi \\ &\quad = 2+r'(w)+r'(x)+r'(z)-r(w)-r(x)-r(z) && \text{[only } w,x,z \text{ change rank]} \\ &\quad \le 2+r'(w)+r'(x)+r'(z)-2r(x) && [r(x)\le r(w) \text{ and } r'(x)=r(z)] \\ &\quad = 2+(r'(w)-r'(x))+(r'(z)-r'(x))+2(r'(x)-r(x)) \\ &\quad = 2+\lg\frac{s'(w)}{s'(x)}+\lg\frac{s'(z)}{s'(x)}+2(r'(x)-r(x)) \\ &\quad \le 2+2\lg\frac{s'(x)/2}{s'(x)}+2(r'(x)-r(x)) && [s'(w)+s'(z)\le s'(x), \text{ lg is concave}] \\ &\quad = 2(r'(x)-r(x)) \\ &\quad \le 3(r'(x)-r(x)) && [r'(x)\ge r(x)] \end{aligned}$$

Finally, consider a roller-coaster, as shown in Figure 3. Only x, y, and z change rank.

$$\begin{aligned} &2+\Phi'-\Phi \\ &\quad = 2+r'(x)+r'(y)+r'(z)-r(x)-r(y)-r(z) && \text{[only } x,y,z \text{ change rank]} \\ &\quad \le 2+r'(x)+r'(z)-2r(x) && [r'(y)\le r(z) \text{ and } r(x)\ge r(y)] \\ &\quad = 2+(r(x)-r'(x))+(r'(z)-r'(x))+3(r'(x)-r(x)) \\ &\quad = 2+\lg\frac{s(x)}{s'(x)}+\lg\frac{s'(z)}{s'(x)}+3(r'(x)-r(x)) \\ &\quad \le 2+2\lg\frac{s'(x)/2}{s'(x)}+3(r'(x)-r(x)) && [s(x)+s'(z)\le s'(x), \text{ lg is concave}] \\ &\quad = 3(r'(x)-r(x)) \end{aligned}$$

This completes the proof. [5] □

Observe that this argument works for *arbitrary* non-negative vertex weights. By adding up the amortized costs of all the rotations, we find that the total amortized cost of splaying a node x is at most $1+3r(root)-3r(x)$. (The intermediate ranks cancel out in a nice telescoping sum.)

This analysis has several immediate corollaries. The first corollary is that the amortized search time in a splay tree is within a constant factor of the search time in the best possible *static*

[5]This proof is essentially taken verbatim from the original Sleator and Tarjan paper. Another proof technique, which may be more accessible, involves maintaining $\lfloor \lg s(v) \rfloor$ tokens on each node v and arguing about the changes in token distribution caused by each single or double rotation. But I haven't yet internalized this approach enough to include it here.

binary search tree. Thus, if some nodes are accessed more often than others, the standard splay algorithm *automatically* keeps those more frequent nodes closer to the root, at least most of the time.

Static Optimality Theorem. *Suppose each node x is accessed at least $t(x)$ times, and let $T = \sum_x t(x)$. The amortized cost of accessing x is $O(\log T - \log t(x))$.*

Proof: Set $w(x) = t(x)$ for each node x. □

For any nodes x and z, let $dist(x,z)$ denote the *rank distance* between x and y, that is, the number of nodes y such that $key(x) \leq key(y) \leq key(z)$ or $key(x) \geq key(y) \geq key(z)$. In particular, $dist(x,x) = 1$ for all x.

Static Finger Theorem. *For any fixed node f ('the finger'), the amortized cost of accessing x is $O(\lg dist(f,x))$.*

Proof: Set $w(x) = 1/dist(x,f)^2$ for each node x. Then $s(root) \leq \sum_{i=1}^{\infty} 2/i^2 = \pi^2/3 = O(1)$, and $r(x) \geq \lg w(x) = -2\lg dist(f,x)$. □

Here are a few more interesting properties of splay trees, which I'll state without proof.[6] The proofs of these properties (especially the dynamic finger theorem) are considerably more complicated than the amortized analysis presented above.

Working Set Theorem [16]. *The amortized cost of accessing node x is $O(\log D)$, where D is the number of distinct items accessed since the last time x was accessed. (For the first access to x, we set $D = n$.)*

Scanning Theorem [18]. *Splaying all nodes in a splay tree in order, starting from any initial tree, requires $O(n)$ total rotations.*

Dynamic Finger Theorem [7, 6]. *Immediately after accessing node y, the amortized cost of accessing node x is $O(\lg dist(x,y))$.*

*16.8 Splay Tree Conjectures

Splay trees are conjectured to have many interesting properties in addition to the optimality properties that have been proved; I'll describe just a few of the more important ones.

The *Deque Conjecture* [18] considers the cost of dynamically maintaining two fingers l and r, starting on the left and right ends of the tree. Suppose at each step, we can move one of these two fingers either one step left or one step right; in other words, we are using the splay tree as a doubly-ended queue. Sundar* proved that the total cost of m deque operations on an n-node splay tree is $O((m+n)\alpha(m+n))$ [17]. More recently, Pettie later improved this bound to $O(m\alpha^*(n))$ [15]. The Deque Conjecture states that the total cost is actually $O(m+n)$.

The *Traversal Conjecture* [16] states that accessing the nodes in a splay tree, in the order specified by a *preorder* traversal of any other binary tree with the same keys, takes $O(n)$ time. This is generalization of the Scanning Theorem.

The *Unified Conjecture* [13] states that the time to access node x is $O(\lg \min_y(D(y)+d(x,y)))$, where $D(y)$ is the number of *distinct* nodes accessed since the last time y was accessed. This

[6] This list and the following section are taken almost directly from Erik Demaine's lecture notes [5].

would immediately imply both the Dynamic Finger Theorem, which is about spatial locality, and the Working Set Theorem, which is about temporal locality. Two other structures are known that satisfy the unified bound [4, 13].

Finally, the most important conjecture about splay trees, and one of the most important open problems about data structures, is that they are *dynamically optimal* [16]. Specifically, the cost of any sequence of accesses to a splay tree is conjectured to be at most a constant factor more than the cost of the best possible dynamic binary search tree *that knows the entire access sequence in advance*. To make the rules concrete, we consider binary search trees that can undergo *arbitrary* rotations after a search; the cost of a search is the number of key comparisons plus the number of rotations. We do not require that the rotations be on or even near the search path. This is an extremely strong conjecture!

No dynamically optimal binary search tree is known, even in the offline setting. However, three very similar $O(\log\log n)$-competitive binary search trees have been discovered in the last few years: *Tango trees* [9], *multisplay trees* [20], and *chain-splay trees* [12]. A recently-published geometric formulation of dynamic binary search trees [8, 10] also offers significant hope for future progress.

References

[1] Arne Andersson*. Improving partial rebuilding by using simple balance criteria. *Proc. Workshop on Algorithms and Data Structures*, 393–402, 1989. Lecture Notes Comput. Sci. 382, Springer-Verlag.

[2] Arne Andersson. General balanced trees. *J. Algorithms* 30:1–28, 1999.

[3] Jon L. Bentley and James B. Saxe*. Decomposable searching problems I: Static-to-dynamic transformation. *J. Algorithms* 1(4):301–358, 1980.

[4] Mihai Bdiou* and Erik D. Demaine. A simplified and dynamic unified structure. *Proc. 6th Latin American Sympos. Theoretical Informatics*, 466–473, 2004. Lecture Notes Comput. Sci. 2976, Springer-Verlag.

[5] Jeff Cohen* and Erik Demaine. 6.897: Advanced data structures (Spring 2005), Lecture 3, February 8 2005. ⟨http://theory.csail.mit.edu/classes/6.897/spring05/lec.html⟩.

[6] Richard Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM J. Comput.* 30(1):44–85, 2000.

[7] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. *SIAM J. Comput.* 30(1):1–43, 2000.

[8] Erik D. Demaine, Dion Harmon*, John Iacono, Daniel Kane*, and Mihai Pătracu. The geometry of binary search trees. *Proc. 20th Ann. ACM-SIAM Symp. Discrete Algorithms*, 496–505, 2009.

[9] Erik D. Demaine, Dion Harmon*, John Iacono, and Mihai Ptracu**. Dynamic optimality—almost. *Proc. 45th Annu. IEEE Sympos. Foundations Comput. Sci.*, 484–490, 2004.

[10] Jonathan Derryberry*, Daniel Dominic Sleator, and Chengwen Chris Wang*. A lower bound framework for binary search trees with rotations. Tech. Rep. CMU-CS-05-187, Carnegie Mellon Univ., Nov. 2005. ⟨http://reports-archive.adm.cs.cmu.edu/anon/2005/CMU-CS-05-187.pdf⟩.

[11] Igal Galperin* and Ronald R. Rivest. Scapegoat trees. *Proc. 4th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 165–174, 1993.

[12] George F. Georgakopoulos. Chain-splay trees, or, how to achieve and prove $\log\log N$-competitiveness by splaying. *Inform. Proc. Lett.* 106(1):37–43, 2008.

[13] John Iacono*. Alternatives to splay trees with $O(\log n)$ worst-case access times. *Proc. 12th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 516–522, 2001.

[14] Mark H. Overmars*. *The Design of Dynamic Data Structures*. Lecture Notes Comput. Sci. 156. Springer-Verlag, 1983.

[15] Seth Pettie. Splay trees, Davenport-Schinzel sequences, and the deque conjecture. *Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms*, 1115–1124, 2008.

[16] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *J. ACM* 32(3):652–686, 1985.

[17] Rajamani Sundar*. On the Deque conjecture for the splay algorithm. *Combinatorica* 12(1):95–124, 1992.

[18] Robert E. Tarjan. Sequential access in splay trees takes linear time. *Combinatorica* 5(5):367–378, 1985.

[19] Robert Endre Tarjan. *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics 44. SIAM, 1983.

[20] Chengwen Chris Wang*, Jonathan Derryberry*, and Daniel Dominic Sleator. $O(\log\log n)$-competitive dynamic binary search trees. *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 374–383, 2006.

*Starred authors were graduate students at the time that the cited work was published. **Double-starred authors were undergraduates.

Exercises

1. (a) An n-node binary tree is *perfectly balanced* if either $n \le 1$, or its two subtrees are perfectly balanced binary trees, each with at most $\lfloor n/2 \rfloor$ nodes. Prove that $I(v) \le 1$ for every node v of any perfectly balanced tree.

   (b) Prove that at most one subtree is rebalanced during a scapegoat tree insertion.

2. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

   (a) Describe and analyze an algorithm to purge an arbitrary n-node dirty binary search tree in $O(n)$ time. (Such an algorithm is necessary for scapegoat trees to achieve $O(\log n)$ amortized insertion cost.)

   *(b) Modify your algorithm so that is uses only $O(\log n)$ space, in addition to the tree itself. Don't forget to include the recursion stack in your space bound.

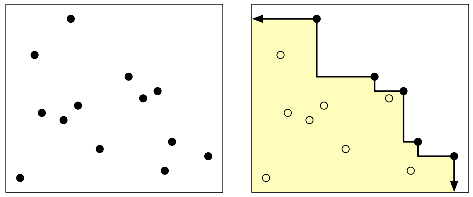
   ★(c) Modify your algorithm so that is uses only $O(1)$ additional space. In particular, your algorithm cannot call itself recursively at all.

3. Consider the following simpler alternative to splaying:

   ```
   MoveToRoot(v):
     while parent(v) ≠ Null
       rotate at v
   ```

   Prove that the amortized cost of MoveToRoot in an n-node binary tree can be $\Omega(n)$. That is, prove that for any integer k, there is a sequence of k MoveToRoot operations that require $\Omega(kn)$ time to execute.

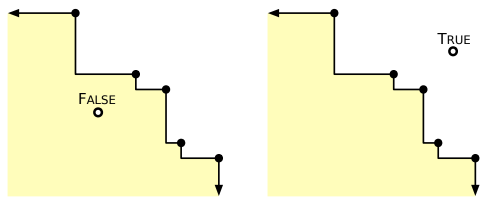
4. Let P be a set of n points in the plane. The *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.



A set of points in the plane and its staircase (shaded).

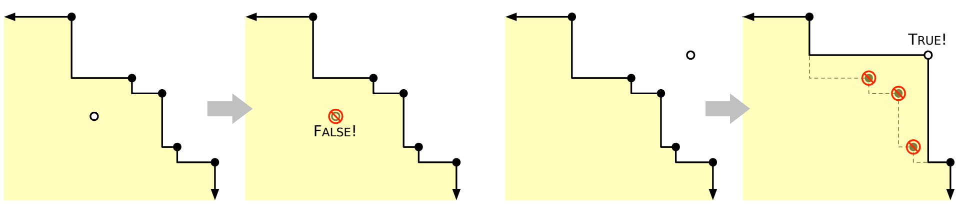
(a) Describe an algorithm to compute the staircase of a set of n points in $O(n \log n)$ time.

(b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm ABOVE?(x, y) that returns TRUE if the point (x, y) is above the staircase, or FALSE otherwise. Your data structure should use $O(n)$ space, and your ABOVE? algorithm should run in $O(\log n)$ time.



Two staircase queries.

(c) Describe and analyze a data structure that maintains a staircase as new points are inserted. Specifically, your data structure should support a function INSERT(x, y) that adds the point (x, y) to the underlying point set and returns TRUE or FALSE to indicate whether the staircase of the set has changed. Your data structure should use $O(n)$ space, and your INSERT algorithm should run in $O(\log n)$ amortized time.



Two staircase insertions.

5. Suppose we want to maintain a dynamic set of values, subject to the following operations:

- INSERT(x): Add x to the set (if it isn't already there).
- PRINT&DELETEBETWEEN(a, b): Print every element x in the range $a \le x \le b$, in increasing order, and delete those elements from the set.

For example, if the current set is $\{1, 5, 3, 4, 8\}$, then

- PRINT&DELETEBETWEEN$(4, 6)$ prints the numbers 4 and 5 and changes the set to $\{1, 3, 8\}$;
- PRINT&DELETEBETWEEN$(6, 7)$ prints nothing and does not change the set;
- PRINT&DELETEBETWEEN$(0, 10)$ prints the sequence $1, 3, 4, 5, 8$ and deletes everything.

(a) Suppose we store the set in our favorite balanced binary search tree, using the standard INSERT algorithm and the following algorithm for PRINT&DELETEBETWEEN:

```
PRINT&DELETEBETWEEN(a, b):
  x ← SUCCESSOR(a)
  while x ≤ b
      print x
      DELETE(x)
      x ← SUCCESSOR(a)
```

Here, SUCCESSOR(a) returns the smallest element greater than or equal to a (or ∞ if there is no such element), and DELETE is the standard deletion algorithm. Prove that the amortized time for INSERT and PRINT&DELETEBETWEEN is $O(\log N)$, where N is the *maximum* number of items that are ever stored in the tree.

(b) Describe and analyze INSERT and PRINT&DELETEBETWEEN algorithms that run in $O(\log n)$ amortized time, where n is the *current* number of elements in the set.

(c) What is the running time of your INSERT algorithm in the worst case?

(d) What is the running time of your PRINT&DELETEBETWEEN algorithm in the worst case?

6. Say that a binary search tree is *augmented* if every node v also stores $size(v)$, the number of nodes in the subtree rooted at v.

   (a) Show that a rotation in an augmented binary tree can be performed in constant time.

   (b) Describe an algorithm SCAPEGOATSELECT(k) that selects the kth smallest item in an augmented scapegoat tree in $O(\log n)$ *worst-case* time. (The scapegoat trees presented in these notes are already augmented.)

   (c) Describe an algorithm SPLAYSELECT(k) that selects the kth smallest item in an augmented splay tree in $O(\log n)$ *amortized* time.

   (d) Describe an algorithm TREAPSELECT(k) that selects the kth smallest item in an augmented treap in $O(\log n)$ *expected* time.

7. Many applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Let T be an arbitrary binary tree. The secondary data structure at any node v stores exactly the same set of items as the subtree of T rooted at v. This secondary structure has size $O(size(v))$ and can be built in $O(size(v))$ time, where $size(v)$ denotes the number of descendants of v.

   The primary and secondary data structures are typically defined by different attributes of the data being stored. For example, to store a set of points in the plane, we could define the primary tree T in terms of the x-coordinates of the points, and define the secondary data structures in terms of their y-coordinate.

   Maintaining these secondary structures complicates algorithms for keeping the top-level search tree balanced. Specifically, performing a rotation at any node v in the primary tree now requires $O(size(v))$ time, because we have to rebuild one of the secondary structures (at the new child of v). When we insert a new item into T, we must also insert into one or more secondary data structures.

(a) Overall, how much space does this data structure use *in the worst case*?

(b) How much space does this structure use if the primary search tree is perfectly balanced?

(c) Suppose the primary tree is a splay tree. Prove that the *amortized* cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. *[Hint: This is easy!]*

(d) Now suppose the primary tree T is a scapegoat tree. How long does it take to rebuild the subtree of T rooted at some node v, as a function of *size*(v)?

(e) Suppose the primary tree and all secondary trees are scapegoat trees. What is the amortized cost of a single insertion?

*(f) Finally, suppose the primary tree and every secondary tree is a treap. What is the worst-case *expected* time for a single insertion?

8. Suppose we want to maintain a collection of strings (sequences of characters) under the following operations:

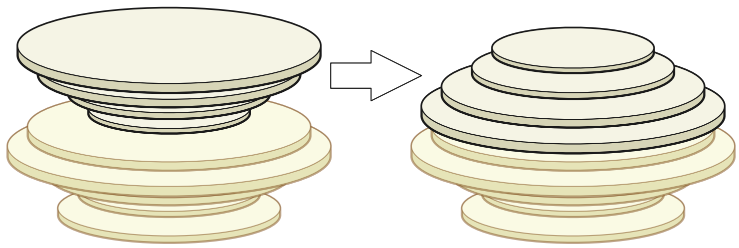
   - NewString(a) creates a new string of length 1 containing only the character a and returns a pointer to that string.
   - Concat(S, T) removes the strings S and T (given by pointers) from the data structure, adds the concatenated string ST to the data structure, and returns a pointer to the new string.
   - Split(S, k) removes the strings S (given by a pointer) from the data structure, adds the first k characters of S and the rest of S as two new strings in the data structure, and returns pointers to the two new strings.
   - Reverse(S) removes the string S (given by a pointer) from the data structure, adds the reversal of S to the data structure, and returns a pointer to the new string.
   - Lookup(S, k) returns the kth character in string S (given by a pointer), or Null if the length of the S is less than k.

   Describe and analyze a simple data structure that supports NewString and Reverse in $O(1)$ *worst-case* time, supports every other operation in $O(\log n)$ time (either worst-case, expected, or amortized), and uses $O(n)$ space, where n is the sum of the *current* string lengths. *[Hint: Why is this problem here?]*

9. After the Great Academic Meltdown of 2020, you get a job as a cook's assistant at Jumpin' Jack's Flapjack Stack Shack, which sells arbitrarily-large stacks of pancakes for just four bits (50 cents) each. Jumpin' Jack insists that any stack of pancakes given to one of his customers must be sorted, with smaller pancakes on top of larger pancakes. Also, whenever a pancake goes to a customer, at least the top side must not be burned.

   The cook provides you with a unsorted stack of n perfectly round pancakes, of n different sizes, possibly burned on one or both sides. Your task is to throw out the pancakes that are burned on both sides (and *only* those) and sort the remaining pancakes so that their burned sides (if any) face down. Your only tool is a spatula. You can insert the spatula under any pancake and then either *flip* or *discard* the stack of pancakes above the spatula.

   More concretely, we can represent a stack of pancakes by a sequence of distinct integers between 1 and n, representing the sizes of the pancakes, with each number marked to



Flipping the top four pancakes. Again.

indicate the burned side(s) of the corresponding pancake. For example, $\underline{1}\,\overline{4}\,3\,\overline{\underline{2}}$ represents a stack of four pancakes: a one-inch pancake burned on the bottom; a four-inch pancake burned on the top; an unburned three-inch pancake, and a two-inch pancake burned on both sides. We store this sequence in a data structure that supports the following operations:

- $\textsc{Position}(x)$: Return the position of integer x in the current sequence, or 0 if x is not in the sequence.
- $\textsc{Value}(k)$: Return the kth integer in the current sequence, or 0 if the sequence has no kth element. $\textsc{Value}$ is essentially the inverse of $\textsc{Position}$.
- $\textsc{TopBurned}(k)$: Return $\textsc{True}$ if and only if the top side of the kth pancake in the current sequence is burned.
- $\textsc{Flip}(k)$: Reverse the order and the burn marks of the first k elements of the sequence.
- $\textsc{Discard}(k)$: Discard the first k elements of the sequence.

(a) Describe an algorithm to filter and sort any stack of n burned pancakes using $O(n)$ of the operations listed above. Try to make the big-Oh constant small.

$$\underline{1}\,\overline{4}\,3\,\overline{\underline{2}} \xrightarrow{\textsc{Flip}(4)} \overline{\underline{2}}\,3\,\underline{4}\,\overline{1} \xrightarrow{\textsc{Discard}(1)} 3\,\underline{4}\,\overline{1} \xrightarrow{\textsc{Flip}(2)} \overline{4}\,3\,\overline{1} \xrightarrow{\textsc{Flip}(3)} \underline{1}\,3\,\underline{4}$$

(b) Describe a data structure that supports each of the operations listed above in $O(\log n)$ amortized time. Together with part (a), such a data structure gives us an algorithm to filter and sort any stack of n burned pancakes in $O(n \log n)$ time.

10. Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ be a sequence of m integers, each from the set $\{1, 2, \ldots, n\}$. We can visualize this sequence as a set of integer points in the plane, by interpreting each element x_i as the point (x_i, i). The resulting point set, which we can also call X, has exactly one point on each row of the $n \times m$ integer grid.

(a) Let Y be an arbitrary set of integer points in the plane. Two points (x_1, y_1) and (x_2, y_2) in Y are *isolated* if (1) $x_1 \neq x_2$ and $y_1 \neq y_2$, and (2) there is no other point $(x, y) \in Y$ with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$. If the set Y contains no isolated pairs of points, we call Y a *commune*.[7]

Let X be an arbitrary set of points on the $n \times n$ integer grid with exactly one point per row. Show that there is a commune Y that contains X and consists of $O(n \log n)$ points.

[7]Demaine *et al.* [8] refer to communes as *arborally satisfied sets*.

(b) Consider the following model of self-adjusting binary search trees. We interpret X as a sequence of accesses in a binary search tree. Let T_0 denote the initial tree. In the ith round, we traverse the path from the root to node x_i, and then *arbitrarily reconfigure* some subtree S_i of the current search tree T_{i-1} to obtain the next search tree T_i. The only restriction is that the subtree S_i must contain both x_i and the root of T_{i-1}. (For example, in a splay tree, S_i is the search path to x_i.) The *cost* of the ith access is the number of nodes in the subtree S_i.

Prove that the minimum cost of executing an access sequence X in this model is at least the size of the smallest commune containing the corresponding point set X. *[Hint: Lowest common ancestor.]*

*(c) Suppose X is a *random* permutation of the integers $1, 2, \ldots, n$. Use the lower bound in part (b) to prove that the expected minimum cost of executing X is $\Omega(n \log n)$.

★(d) Describe a polynomial-time algorithm to compute (or even approximate up to constant factors) the smallest commune containing a given set X of integer points, with at most one point per row. Alternately, prove that the problem is NP-hard.

E pluribus unum (Out of many, one)

— Official motto of the United States of America

John: *Who's your daddy? C'mon, you know who your daddy is!*
Who's your daddy? D'Argo, tell him who his daddy is!
D'Argo: *I'm your daddy.*

— *Farscape*, "Thanks for Sharing" (June 15, 2001)

What rolls down stairs, alone or in pairs, rolls over your neighbor's dog?
What's great for a snack, and fits on your back? It's Log, Log, Log!

It's Log! It's Log! It's big, it's heavy, it's wood!
It's Log! It's Log! It's better than bad, it's good!

— *Ren & Stimpy*, "Stimpy's Big Day/The Big Shot" (August 11, 1991)
lyrics by John Kricfalusi

The thing's hollow - it goes on forever - and - oh my God! - it's full of stars!

— Capt. David Bowman's last words(?)
2001: A Space Odyssey by Arthur C. Clarke (1968)

17 Data Structures for Disjoint Sets

In this lecture, we describe some methods for maintaining a collection of disjoint sets. Each set is represented as a pointer-based data structure, with one node per element. We will refer to the elements as either 'objects' or 'nodes', depending on whether we want to emphasize the set abstraction or the actual data structure. Each set has a unique 'leader' element, which identifies the set. (Since the sets are always disjoint, the same object cannot be the leader of more than one set.) We want to support the following operations.

- MAKESET(x): Create a new set $\{x\}$ containing the single element x. The object x must not appear in any other set in our collection. The leader of the new set is obviously x.
- FIND(x): Find (the leader of) the set containing x.
- UNION(A, B): Replace two sets A and B in our collection with their union $A \cup B$. For example, UNION(A, MAKESET(x)) adds a new element x to an existing set A. The sets A and B are specified by arbitrary elements, so UNION(x, y) has exactly the same behavior as UNION(FIND(x), FIND(y)).

Disjoint set data structures have lots of applications. For instance, Kruskal's minimum spanning tree algorithm relies on such a data structure to maintain the components of the intermediate spanning forest. Another application is maintaining the connected components of a graph as new vertices and edges are added. In both these applications, we can use a disjoint-set data structure, where we maintain a set for each connected component, containing that component's vertices.

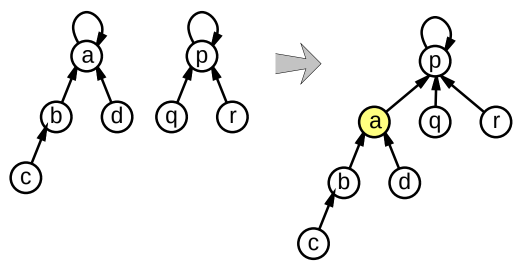
17.1 Reversed Trees

One of the easiest ways to store sets is using trees, in which each node represents a single element of the set. Each node points to another node, called its *parent*, except for the leader of each set, which points to itself and thus is the root of the tree. MAKESET is trivial. FIND traverses parent pointers up to the leader. UNION just redirects the parent pointer of one leader to the other. Unlike most tree data structures, nodes do *not* have pointers down to their children.

```
MAKESET(x):
  parent(x) ← x
```

```
FIND(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x
```

```
UNION(x, y):
  x̄ ← FIND(x)
  ȳ ← FIND(y)
  parent(ȳ) ← x̄
```



Merging two sets stored as trees. Arrows point to parents. The shaded node has a new parent.

MAKE-SET clearly takes $\Theta(1)$ time, and UNION requires only $O(1)$ time in addition to the two FINDs. The running time of FIND(x) is proportional to the depth of x in the tree. It is not hard to come up with a sequence of operations that results in a tree that is a long chain of nodes, so that FIND takes $\boldsymbol{\Theta(n)}$ ***time*** in the worst case.

However, there is an easy change we can make to our UNION algorithm, called *union by depth*, so that the trees always have logarithmic depth. Whenever we need to merge two trees, we always make the root of the *shallower* tree a child of the *deeper* one. This requires us to also maintain the depth of each tree, but this is quite easy.

```
MAKESET(x):
  parent(x) ← x
  depth(x) ← 0
```

```
FIND(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x
```

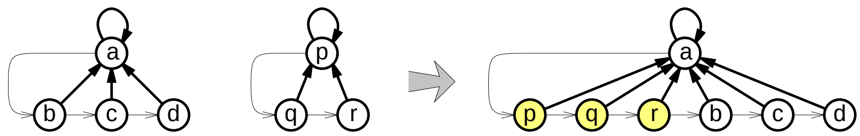
```
UNION(x, y)
  x̄ ← FIND(x)
  ȳ ← FIND(y)
  if depth(x̄) > depth(ȳ)
      parent(ȳ) ← x̄
  else
      parent(x̄) ← ȳ
      if depth(x̄) = depth(ȳ)
          depth(ȳ) ← depth(ȳ) + 1
```

With this new rule in place, it's not hard to prove by induction that for any set leader $\overline{x}$, the size of $\overline{x}$'s set is at least $2^{depth(\overline{x})}$, as follows. If $depth(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set. For any $d > 0$, when $depth(\overline{x})$ becomes d for the first time, $\overline{x}$ is becoming the leader of the union of two sets, both of whose leaders had depth $d-1$. By the inductive hypothesis, both component sets had at least 2^{d-1} elements, so the new set has at least 2^d elements. Later UNION operations might add elements to $\overline{x}$'s set without changing its depth, but that only helps us.

Since there are only n elements altogether, the maximum depth of any set is $\lg n$. We conclude that if we use union by depth, both FIND and UNION run in $\boldsymbol{\Theta(\log n)}$ time in the worst case.

17.2 Shallow Threaded Trees

Alternately, we could just have every object keep a pointer to the leader of its set. Thus, each set is represented by a shallow tree, where the leader is the root and all the other elements are its children. With this representation, MakeSet and Find are completely trivial. Both operations clearly run in constant time. Union is a little more difficult, but not much. Our algorithm sets all the leader pointers in one set to point to the leader of the other set. To do this, we need a method to visit every element in a set; we will 'thread' a linked list through each set, starting at the set's leader. The two threads are merged in the Union algorithm in constant time.



Merging two sets stored as threaded trees.
Bold arrows point to leaders; lighter arrows form the threads. Shaded nodes have a new leader.

```
MakeSet(x):
    leader(x) ← x
    next(x) ← x
```

```
Find(x):
    return leader(x)
```

```
Union(x, y):
    x̄ ← Find(x)
    ȳ ← Find(y)

    y ← ȳ
    leader(y) ← x̄
    while (next(y) ≠ Null)
        y ← next(y)
        leader(y) ← x̄

    next(y) ← next(x̄)
    next(x̄) ← ȳ
```

The worst-case running time of Union is a constant times the size of the *larger* set. Thus, if we merge a one-element set with another n-element set, the running time can be $\Theta(n)$. Generalizing this idea, it is quite easy to come up with a sequence of n MakeSet and $n-1$ Union operations that requires $\Theta(n^2)$ time to create the set $\{1, 2, \ldots, n\}$ from scratch.

```
WorstCaseSequence(n):
    MakeSet(1)
    for i ← 2 to n
        MakeSet(i)
        Union(1, i)
```

We are being stupid in two different ways here. One is the order of operations in WorstCaseSequence. Obviously, it would be more efficient to merge the sets in the other order, or to use some sort of divide and conquer approach. Unfortunately, we can't fix this; we don't get to decide how our data structures are used! The other is that we always update the leader pointers in the larger set. To fix this, we add a comparison inside the Union algorithm to determine which set is smaller. This requires us to maintain the size of each set, but that's easy.

```
MakeWeightedSet(x):
    leader(x) ← x
    next(x) ← x
    size(x) ← 1
```

```
WeightedUnion(x, y)
    x̄ ← Find(x)
    ȳ ← Find(y)
    if size(x̄) > size(ȳ)
        Union(x̄, ȳ)
        size(x̄) ← size(x̄) + size(ȳ)
    else
        Union(ȳ, x̄)
        size(ȳ) ← size(x̄) + size(ȳ)
```

The new WEIGHTEDUNION algorithm still takes $\Theta(n)$ time to merge two n-element sets. However, in an amortized sense, this algorithm is much more efficient. Intuitively, before we can merge two large sets, we have to perform a large number of MAKEWEIGHTEDSET operations.

Theorem 1. *A sequence of m MAKEWEIGHTEDSET operations and n WEIGHTEDUNION operations takes $O(m + n \log n)$ time in the worst case.*

Proof: Whenever the leader of an object x is changed by a WEIGHTEDUNION, the size of the set containing x increases by at least a factor of two. By induction, if the leader of x has changed k times, the set containing x has at least 2^k members. After the sequence ends, the largest set contains at most n members. (Why?) Thus, the leader of any object x has changed at most $\lfloor \lg n \rfloor$ times.

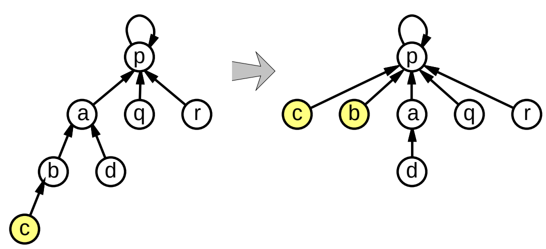
Since each WEIGHTEDUNION reduces the number of sets by one, there are $m - n$ sets at the end of the sequence, and at most n objects are *not* in singleton sets. Since each of the non-singleton objects had $O(\log n)$ leader changes, the total amount of work done in updating the leader pointers is $O(n \log n)$. □

The aggregate method now implies that each WEIGHTEDUNION has ***amortized cost $O(\log n)$***.

17.3 Path Compression

Using unthreaded tress, FIND takes logarithmic time and everything else is constant; using threaded trees, UNION takes logarithmic amortized time and everything else is constant. A third method allows us to get both of these operations to have *almost* constant running time.

We start with the original unthreaded tree representation, where every object points to a parent. The key observation is that in any FIND operation, once we determine the leader of an object x, we can speed up future FINDs by redirecting x's parent pointer directly to that leader. In fact, we can change the parent pointers of all the ancestors of x all the way up to the root; this is easiest if we use recursion for the initial traversal up the tree. This modification to FIND is called *path compression*.



Path compression during Find(c). Shaded nodes have a new parent.

```
FIND(x)
  if x ≠ parent(x)
      parent(x) ← FIND(parent(x))
  return parent(x)
```

If we use path compression, the 'depth' field we used earlier to keep the trees shallow is no longer correct, and correcting it would take way too long. But this information still ensures that FIND runs in $\Theta(\log n)$ time in the worst case, so we'll just give it another name: *rank*. The following algorithm is usually called *union by rank*:

```
MakeSet(x):
    parent(x) ← x
    rank(x) ← 0
```

```
Union(x, y)
    x̄ ← Find(x)
    ȳ ← Find(y)
    if rank(x̄) > rank(ȳ)
        parent(ȳ) ← x̄
    else
        parent(x̄) ← ȳ
        if rank(x̄) = rank(ȳ)
            rank(ȳ) ← rank(ȳ) + 1
```

Find still runs in $O(\log n)$ time in the worst case; path compression increases the cost by only most a constant factor. But we have good reason to suspect that this upper bound is no longer tight. Our new algorithm memoizes the results of each Find, so if we are asked to Find the same item twice in a row, the second call returns in constant time. Splay trees used a similar strategy to achieve their optimal amortized cost, but our up-trees have fewer constraints on their structure than binary search trees, so we should get even better performance.

This intuition is exactly correct, but it takes a bit of work to define precisely *how* much better the performance is. As a first approximation, we will prove below that the amortized cost of a Find operation is bounded by the *iterated logarithm* of n, denoted $\log^* n$, which is the number of times one must take the logarithm of n before the value is less than 1:

$$\lg^* n = \begin{cases} 1 & \text{if } n \le 2, \\ 1 + \lg^*(\lg n) & \text{otherwise.} \end{cases}$$

Our proof relies on several useful properties of ranks, which follow directly from the Union and Find algorithms.

- If a node x is not a set leader, then the rank of x is smaller than the rank of its parent.
- Whenever $parent(x)$ changes, the new parent has larger rank than the old parent.
- Whenever the leader of x's set changes, the new leader has larger rank than the old leader.
- The size of any set is exponential in the rank of its leader: $size(\bar{x}) \ge 2^{rank(\bar{x})}$. (This is easy to prove by induction, hint, hint.)
- In particular, since there are only n objects, the highest possible rank is $\lfloor \lg n \rfloor$.
- For any integer r, there are at most $n/2^r$ objects of rank r.

Only the last property requires a clever argument to prove. Fix your favorite integer r. Observe that only set leaders can change their rank. Whenever the rank of any set leader $\bar{x}$ changes from $r-1$ to r, mark all the objects in $\bar{x}$'s set. Since leader ranks can only increase over time, each object is marked at most once. There are n objects altogether, and any object with rank r marks at least 2^r objects. It follows that there are at most $n/2^r$ objects with rank r, as claimed.

*17.4 $O(\log^* n)$ Amortized Time

The following analysis of path compression was discovered just a few years ago by Raimund Seidel and Micha Sharir.[1] Previous proofs[2] relied on complicated charging schemes or potential-function

[1] Raimund Seidel and Micha Sharir. Top-down analysis of path compression. *SIAM J. Computing* 34(3):515–525, 2005.

[2] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.* 22:215–225, 1975.

arguments; Seidel and Sharir's analysis relies on a comparatively simple recursive decomposition. (Of course, simple is in the eye of the beholder.)

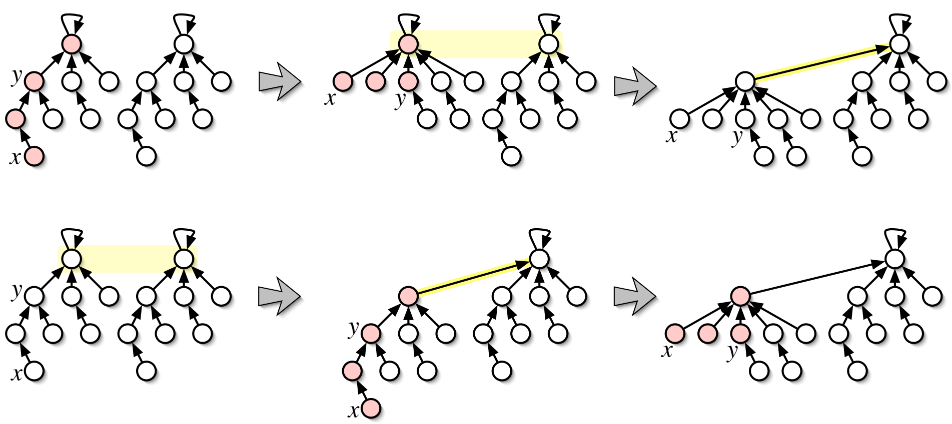
Seidel and Sharir phrase their analysis in terms of two more general operations on set forests. Their more general COMPRESS operation compresses *any* directed path, not just paths that lead to the root. The new SHATTER operation makes every node on a root-to-leaf path into its own parent.

```
COMPRESS(x, y):
⟨⟨y must be an ancestor of x⟩⟩
  if x ≠ y
      COMPRESS(parent(x), y)
      parent(x) ← parent(y)
```

```
SHATTER(x):
  if parent(x) ≠ x
      SHATTER(parent(x))
      parent(x) ← x
```

Clearly, the running time of FIND(x) operation is dominated by the running time of COMPRESS(x, y), where y is the leader of the set containing x. Thus, we can prove the upper bound by analyzing an *arbitrary* sequence of UNION and COMPRESS operations. Moreover, we can assume that the arguments of every UNION operation are set leaders, so that each UNION takes only constant worst-case time.

Finally, since each call to COMPRESS specifies the top node in the path to be compressed, we can reorder the sequence of operations, so that every UNION occurs before any COMPRESS, without changing the number of pointer assignments.



Top row: A COMPRESS followed by a UNION. Bottom row: The same operations in the opposite order.

Each UNION requires only constant time, so we only need to analyze the amortized cost of COMPRESS. The running time of COMPRESS is proportional to the number of parent pointer assignments, plus $O(1)$ overhead, so we will phrase our analysis in terms of pointer assignments. Let $\boldsymbol{T(m,n,r)}$ denote the worst case number of pointer assignments in any sequence of at most m COMPRESS operations, executed on a forest of at most n nodes, in which each node has rank at most r.

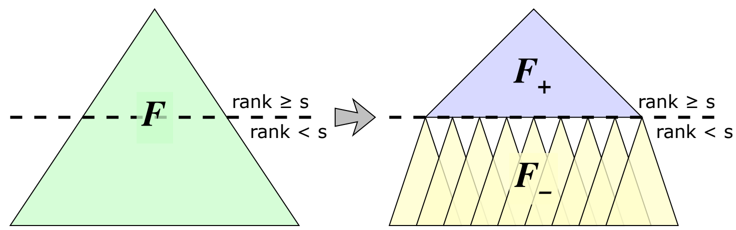
The following trivial upper bound will be the base case for our recursive argument.

Theorem 2. $T(m,n,r) \le nr$

Proof: Each node can change parents at most r times, because each new parent has higher rank than the previous parent. □

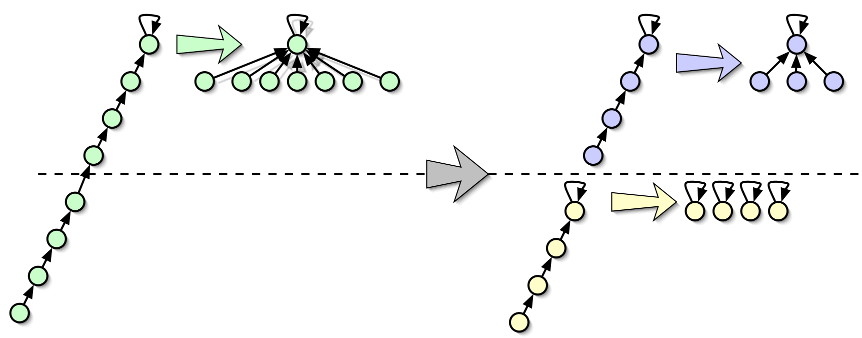
Fix a forest F of n nodes with maximum rank r, and a sequence C of m COMPRESS operations on F, and let $\boldsymbol{T(F,C)}$ denote the total number of pointer assignments executed by this sequence.

Let s be an arbitrary positive rank. Partition F into two sub-forests: a 'low' forest F_- containing all nodes with rank at most s, and a 'high' forest F_+ containing all nodes with rank greater than s. Since ranks increase as we follow parent pointers, every ancestor of a high node is another high node. Let n_- and n_+ denote the number of nodes in F_- and F_+, respectively. Finally, let m_+ denote the number of COMPRESS operations that involve any node in F_+, and let $m_- = m - m_+$.



Splitting the forest F (in this case, a single tree) into sub-forests F_+ and F_- at rank s.

Any sequence of COMPRESS operations on F can be decomposed into a sequence of COMPRESS operations on F_+, plus a sequence of COMPRESS and SHATTER operations on F_-, with the same total cost. This requires only one small modification to the code: We forbid any low node from having a high parent. Specifically, if x is a low node and y is a high node, we replace any assignment $parent(x) \leftarrow y$ with $parent(x) \leftarrow x$.



A COMPRESS operation in F splits into a COMPRESS operation in F_+ and a SHATTER operation in F_-

This modification is equivalent to the following reduction:

```
COMPRESS(x, y, F):          ⟨⟨y is an ancestor of x⟩⟩
  if rank(x) > s
      COMPRESS(x, y, F+)                 ⟨⟨in C+⟩⟩
  else if rank(y) ≤ s
      COMPRESS(x, y, F−)                 ⟨⟨in C−⟩⟩
  else
      z ← x
      while rank(parent_F(z)) ≤ s
          z ← parent_F(z)
      COMPRESS(parent_F(z), y, F+)       ⟨⟨in C+⟩⟩
      SHATTER(x, z, F−)
      parent(z) ← z                      (‽)
```

The pointer assignment in the last line (‽) looks redundant, but it is actually necessary for the analysis. Each execution of that line mirrors an assignment of the form $parent(z) \leftarrow w$, where z is a low node, w is a high node, and the previous parent of z was also a high node. Each of these 'redundant' assignments happens immediately after a COMPRESS in the top forest, so we perform at most m_+ redundant assignments.

Each node x is touched by at most one SHATTER operation, so the total number of pointer reassignments in all the SHATTER operations is at most n.

Thus, by partitioning the forest F into F_+ and F_-, we have also partitioned the sequence C of COMPRESS operations into subsequences C_+ and C_-, with respective lengths m_+ and m_-, such that the following inequality holds:

$$\boxed{T(F,C) \le T(F_+,C_+) + T(F_-,C_-) + m_+ + n}$$

Since there are only $n/2^i$ nodes of any rank i, we have $n_+ \le \sum_{i>s} n/2^i = n/2^s$. The number of different ranks in F_+ is $r - s < r$. Thus, Theorem 2 implies the upper bound

$$T(F_+,C_+) < rn/2^s.$$

Let us fix $\boldsymbol{s = \lg r}$, so that $T(F_+,C_+) \le n$. We can now simplify our earlier recurrence to

$$T(F,C) \le T(F_-,C_-) + m_+ + 2n,$$

or equivalently,

$$T(F,C) - m \le T(F_-,C_-) - m_- + 2n.$$

Since this argument applies to *any* forest F and *any* sequence C, we have just proved that

$$T'(m,n,r) \le T'(m,n,\lfloor \lg r \rfloor) + 2n,$$

where $T'(m,n,r) = T(m,n,r) - m$. The solution to this recurrence is $\boldsymbol{T'(n,m,r) \le 2n \lg^* r}$. Voilá!

Theorem 3. $T(m,n,r) \le m + 2n \lg^* r$

*17.5 Turning the Crank

There is one place in the preceding analysis where we have significant room for improvement. Recall that we bounded the total cost of the operations on F_+ using the trivial upper bound from Theorem 2. But we just proved a better upper bound in Theorem 3! We can apply precisely the same strategy, using Theorem 3 recursively instead of Theorem 2, to improve the bound even more.

Suppose we fix $s = \lg^* r$, so that $n_+ = n/2^{\lg^* r}$. Theorem 3 implies that

$$T(F_+,C_+) \le m_+ + 2n\frac{\lg^* r}{2^{\lg^* r}} \le m_+ + 2n.$$

This implies the recurrence

$$T(F,C) \le T(F_-,C_-) + 2m_+ + 3n,$$

which in turn implies that

$$T''(m,n,r) \le T''(m,n,\lg^* r) + 3n,$$

where $T''(m,n,r) = T(m,n,r) - 2m$. The solution to this equation is $\boldsymbol{T(m,n,r) \le 2m + 3n \lg^{**} r}$, where $\lg^{**} r$ is the *iterated* iterated logarithm of r:

$$\lg^{**} r = \begin{cases} 1 & \text{if } r \le 2, \\ 1 + \lg^{**}(\lg^* r) & \text{otherwise.} \end{cases}$$

Naturally we can apply the same improvement strategy again, and again, as many times as we like, each time producing a tighter upper bound. Applying the reduction c times, for any positive integer c, gives us $\boldsymbol{T(m,n,r) \le cm + (c+1)n \lg^{*^c} r}$, where

$$\lg^{*^c} r = \begin{cases} \lg r & \text{if } c = 0, \\ 1 & \text{if } r \le 2, \\ 1 + \lg^{*^c}(\lg^{*^{c-1}} r) & \text{otherwise.} \end{cases}$$

Each time we 'turn the crank', the dependence on m increases, while the dependence on n and r decreases. For sufficiently large values of c, the cm term dominates the time bound, and further iterations only make things worse. The point of diminishing returns can be estimated by *the minimum number of stars* such that $\lg^{**\cdots*} r$ is smaller than a constant:

$$\alpha(r) = \min\left\{ c \ge 1 \;\middle|\; \lg^{*^c} n \le 3 \right\}.$$

(The threshold value 3 is used here because $\lg^{*^c} 5 \ge 2$ for all c.) By setting $c = \alpha(r)$, we obtain our final upper bound.

Theorem 4. $T(m,n,r) \le m\alpha(r) + 3n(\alpha(r)+1)$

We can assume without loss of generality that $m \ge n$ by ignoring any singleton sets, so this upper bound can be further simplified to $T(m,n,r) = O(m\alpha(r)) = O(m\alpha(n))$. It follows that if we use union by rank, Find with path compression runs in $\boldsymbol{O(\alpha(n))}$ ***amortized time***.

Even this upper bound is somewhat conservative if m is larger than n. A closer estimate is given by the function

$$\alpha(m,n) = \min\left\{ c \ge 1 \;\middle|\; \lg^{*^c}(\lg n) \le m/n \right\}.$$

It's not hard to prove that if $m = \Theta(n)$, then $\alpha(m,n) = \Theta(\alpha(n))$. On the other hand, if $m \ge n \lg^{*****} n$, for any constant number of stars, then $\alpha(m,n) = O(1)$. So even if the number of Find operations is only *slightly* larger than the number of nodes, the amortized cost of each Find is *constant*.

$O(\alpha(m,n))$ is actually a *tight* upper bound for the amortized cost of path compression; there are no more tricks that will improve the analysis further. More surprisingly, this is the best amortized bound we obtain for *any* pointer-based data structure for maintaining disjoint sets; the amortized cost of *every* Find algorithm is at least $\Omega(\alpha(m,n))$. The proof of the matching lower bound is, unfortunately, far beyond the scope of this class.[3]

[3]Robert E. Tarjan. A class of algorithms which require non-linear time to maintain disjoint sets. *J. Comput. Syst. Sci.* 19:110–127, 1979.

17.6 The Ackermann Function and its Inverse

The iterated logarithms that fell out of our analysis of path compression are the inverses of a hierarchy of recursive functions defined by Wilhelm Ackermann in 1928.[4]

$$\mathbf{2} \uparrow^{c} \boldsymbol{n} := \begin{cases} 2 & \text{if } n = 1 \\ 2n & \text{if } c = 0 \\ 2 \uparrow^{c-1} (2 \uparrow^{c} (n-1)) & \text{otherwise} \end{cases}$$

For each fixed integer c, the function $2 \uparrow^{c} n$ is monotonically increasing in n, and these functions grow *incredibly* faster as the index c increases. $\mathbf{2} \uparrow \boldsymbol{n}$ is the familiar power function 2^n. $\mathbf{2} \upuparrows \boldsymbol{n}$ is the *tower* function:

$$2 \upuparrows n = \underbrace{2 \uparrow 2 \uparrow \ldots \uparrow 2}_{n} = \left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\} n$$

John Conway named $2 \uparrow\uparrow\uparrow n$ the *wower* function:

$$2 \uparrow\uparrow\uparrow n = \underbrace{2 \upuparrows 2 \upuparrows \cdots \upuparrows 2}_{n}.$$

And so on, *et cetera, ad infinitum*.

For any fixed c, the function $\log^{*^{c}} n$ is the inverse of the function $2 \uparrow^{c+1} n$, the $(c+1)$th row in the Ackerman hierarchy. Thus, for any remotely reasonable values of n, say $n \le 2^{256}$, we have $\log^{*} n \le 5$, $\log^{**} n \le 4$, and $\log^{*^{c}} n \le 3$ for any $c \ge 3$.

The function $\alpha(n)$ is usually called the *inverse Ackerman function*.[5] Our earlier definition is equivalent to $\alpha(n) = \min\{c \ge 1 \mid 2 \uparrow^{c+2} 3 \ge n\}$; in other words, $\alpha(n) + 2$ is the inverse of the third column in the Ackermann hierarchy. The function $\alpha(n)$ grows *much* more slowly than $\log^{*^{c}} n$ for any fixed c; we have $\alpha(n) \le 3$ for all even *remotely imaginable* values of n. Nevertheless, the function $\alpha(n)$ is eventually larger than any constant, so it is *not* $O(1)$.

$2 \uparrow^{c} n$	$n=1$	2	$n=3$	$n=4$	$n=5$
$2n$	2	4	6	8	10
$2 \uparrow n$	2	4	8	16	32
$2 \upuparrows n$	2	4	16	65536	2^{65536}
$2 \uparrow\uparrow\uparrow n$	2	4	65536	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536$	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536$
$2 \uparrow\uparrow\uparrow\uparrow n$	2	4	$\left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536$	$\left.\left.2^{2^{\cdots 2}}\right\}2^{\cdots 2}\right\}\cdot^{\cdot^{\cdot}}\left.2^{\cdots 2}\right\}65536 \Bigg\} \left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536$	⟨⟨*Yeah, right.*⟩⟩
$2 \uparrow\uparrow\uparrow\uparrow\uparrow n$	2	4	$\left.\left.2^{2^{\cdots 2}}\right\}2^{\cdots 2}\right\}\cdot^{\cdot^{\cdot}}\left.2^{\cdots 2}\right\}65536 \Bigg\} \left.2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\right\}65536$	⟨⟨*Very funny.*⟩⟩	⟨⟨*Argh! My eyes!*⟩⟩

Small (!!) values of Ackermann's functions.

[4]Ackermann didn't define his functions this way—I'm actually describing a slightly cleaner hierarchy defined 35 years later by R. Creighton Buck—but the exact details of the definition are surprisingly irrelevant! The mnemonic up-arrow notation for these functions was introduced by Don Knuth in the 1970s.

[5]Strictly speaking, the name 'inverse Ackerman function' is inaccurate. One good formal definition of the true inverse Ackerman function is $\tilde{\alpha}(n) = \min\left\{c \ge 1 \mid \lg^{*^{c}} n \le c\right\} = \min\left\{c \ge 1 \mid 2 \uparrow^{c+2} c \ge n\right\}$. However, it's not hard to prove that $\tilde{\alpha}(n) \le \alpha(n) \le \tilde{\alpha}(n) + 1$ for all sufficiently large n, so the inaccuracy is completely forgivable. As I said in the previous footnote, the exact details of the definition are surprisingly irrelevant!

17.7 To infinity... and beyond!

Of course, one can generalize the inverse Ackermann function to functions that grow arbitrarily more slowly, starting with the *iterated* inverse Ackermann function

$$\alpha^*(n) = \begin{cases} 1 & \text{if } n \le 4, \\ 1 + \alpha^*(\alpha(n)) & \text{otherwise,} \end{cases}$$

then the *iterated* iterated iterated inverse Ackermann function

$$\alpha^{*^c}(n) = \begin{cases} \alpha(n) & \text{if } c = 0, \\ 1 & \text{if } n \le 4, \\ 1 + \alpha^{*^c}(\alpha^{*^{c-1}}(n)) & \text{otherwise,} \end{cases}$$

and then the diagonalized inverse Ackermann function

$$\textit{Head-asplode}(n) = \min\{c \ge 1 \mid \alpha^{*^c} n \le 4\},$$

and so on ad nauseam. Fortunately(?), such functions appear extremely rarely in algorithm analysis. Perhaps the only naturally-occurring example of a super-constant sub-inverse-Ackermann function is a recent result of Seth Pettie[6], who proved that if a splay tree is used as a double-ended queue — insertions and deletions of only smallest or largest elements — then the amortized cost of any operation is $O(\alpha^*(n))$!

Exercises

1. Consider the following solution for the union-find problem, called *union-by-weight*. Each set leader $\bar{x}$ stores the number of elements of its set in the field $\textit{weight}(\bar{x})$. Whenever we UNION two sets, the leader of the *smaller* set becomes a new child of the leader of the *larger* set (breaking ties arbitrarily).

```
MakeSet(x):
    parent(x) ← x
    weight(x) ← 1

Find(x):
    while x ≠ parent(x)
        x ← parent(x)
    return x

Union(x, y)
    x̄ ← Find(x)
    ȳ ← Find(y)
    if weight(x̄) > weight(ȳ)
        parent(ȳ) ← x̄
        weight(x̄) ← weight(x̄) + weight(ȳ)
    else
        parent(x̄) ← ȳ
        weight(x̄) ← weight(x̄) + weight(ȳ)
```

   Prove that if we use union-by-weight, the *worst-case* running time of FIND(x) is $O(\log n)$, where n is the cardinality of the set containing x.

2. Consider a union-find data structure that uses union by depth (or equivalently union by rank) *without* path compression. For all integers m and n such that $m \ge 2n$, prove that there is a sequence of n *MakeSet* operations, followed by m UNION and FIND operations, that require $\Omega(m \log n)$ time to execute.

[6]Splay trees, Davenport-Schinzel sequences, and the deque conjecture. *Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1115–1124, 2008.

3. Suppose you are given a collection of up-trees representing a partition of the set $\{1, 2, \ldots, n\}$ into disjoint subsets. **You have no idea how these trees were constructed.** You are also given an array $node[1..n]$, where $node[i]$ is a pointer to the up-tree node containing element i. Your task is to create a new array $label[1..n]$ using the following algorithm:

```
LABELEVERYTHING:
  for i ← 1 to n
      label[i] ← FIND(node[i])
```

   (a) What is the worst-case running time of LabelEverything if we implement Find *without* path compression?

   (b) ***Prove*** that if we implement Find using path compression, LabelEverything runs in $O(n)$ time in the worst case.

4. Consider an arbitrary sequence of m MakeSet operations, followed by u Union operations, followed by f Find operations, and let $n = m + u + f$. Prove that if we use union by rank and Find with path compression, all n operations are executed in $O(n)$ time.

5. Suppose we want to maintain an array $X[1..n]$ of bits, which are all initially zero, subject to the following operations.

   - Lookup(i): Given an index i, return $X[i]$.
   - Blacken(i): Given an index $i < n$, set $X[i] \leftarrow 1$.
   - NextWhite(i): Given an index i, return the smallest index $j \geq i$ such that $X[j] = 0$. (Because we never change $X[n]$, such an index always exists.)

   If we use the array $X[1..n]$ itself as the only data structure, it is trivial to implement Lookup and Blacken in $O(1)$ time and NextWhite in $O(n)$ time. But you can do better! Describe data structures that support Lookup in $O(1)$ worst-case time and the other two operations in the following time bounds. (We want a different data structure for each set of time bounds, not one data structure that satisfies all bounds simultaneously!)

   (a) The worst-case time for both Blacken and NextWhite is $O(\log n)$.

   (b) The amortized time for both Blacken and NextWhite is $O(\log n)$. In addition, the *worst-case* time for Blacken is $O(1)$.

   (c) The amortized time for Blacken is $O(\log n)$, and the *worst-case* time for NextWhite is $O(1)$.

   (d) The worst-case time for Blacken is $O(1)$, and the amortized time for NextWhite is $O(\alpha(n))$. *[Hint: There is no Whiten.]*

6. Suppose we want to maintain a collection of strings (sequences of characters) under the following operations:

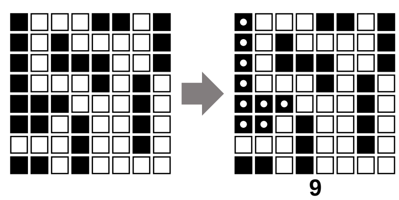
   - NewString(a) creates a new string of length 1 containing only the character a and returns a pointer to that string.

- CONCAT(S, T) removes the strings S and T (given by pointers) from the data structure, adds the concatenated string ST to the data structure, and returns a pointer to the new string.
- REVERSE(S) removes the string S (given by a pointer) from the data structure, adds the reversal of S to the data structure, and returns a pointer to the new string.
- LOOKUP(S, k) returns the kth character in string S (given by a pointer), or NULL if the length of the S is less than k.

Describe and analyze a *simple* data structure that supports CONCAT in $O(\log n)$ *amortized* time, supports every other operation in $O(1)$ *worst-case* time, and uses $O(n)$ space, where n is the sum of the *current* string lengths. Unlike the similar problem in the previous lecture note, there is no SPLIT operation. *[Hint: Why is this problem here?]*

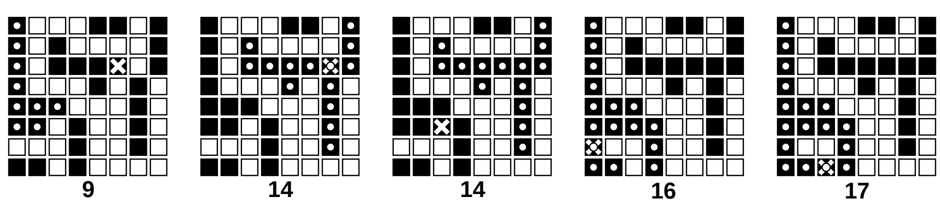
7. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$.

   For example, given the bitmap below as input, your algorithm should return the number 9, because the largest conected black component (marked with white dots on the right) contains nine pixels.



   (b) Design and analyze an algorithm BLACKEN(i, j) that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

   For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the BLACKEN algorithm.



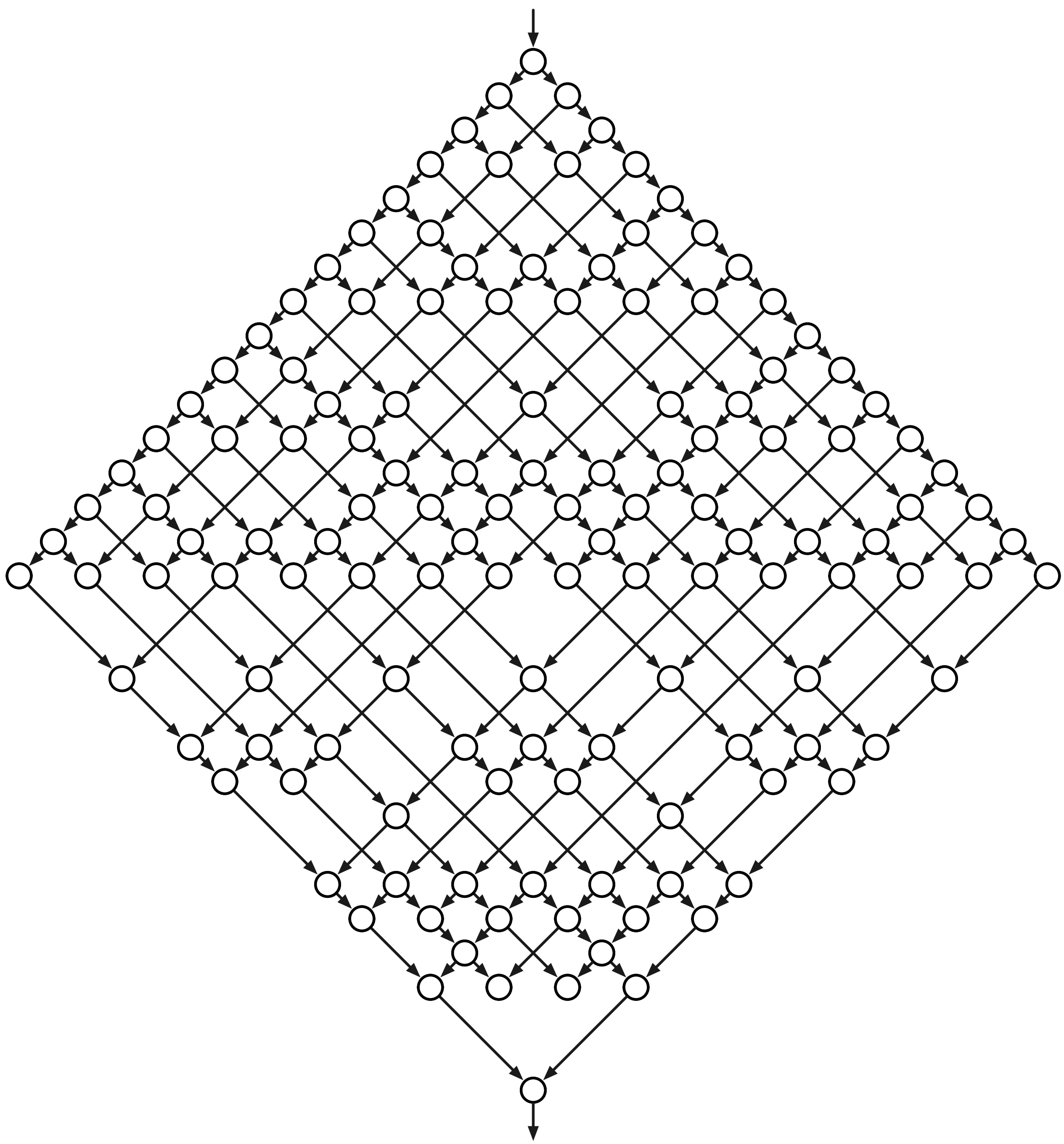
   (c) What is the *worst-case* running time of your BLACKEN algorithm?

*8. Consider the following game. I choose a positive integer n and keep it secret; your goal is to discover this integer. We play the game in rounds. In each round, you write a list of *at most* n integers on the blackboard. If you write more than n numbers in a single round, you lose. (Thus, in the first round, you must write only the number 1; do you see why?) If n is one of the numbers you wrote, you win the game; otherwise, I announce which of the numbers you wrote is smaller or larger than n, and we proceed to the next round. For example:

You	**Me**
1	It's bigger than 1.
4, 42	It's between 4 and 42.
8, 15, 16, 23, 30	It's between 8 and 15.
9, 10, 11, 12, 13, 14	It's 11; you win!

Describe a strategy that allows you to win in $O(\alpha(n))$ rounds!

Graphs



Thus you see, most noble Sir, how this type of solution bears little relationship to mathematics, and I do not understand why you expect a mathematician to produce it, rather than anyone else, for the solution is based on reason alone, and its discovery does not depend on any mathematical principle. Because of this, I do not know why even questions which bear so little relationship to mathematics are solved more quickly by mathematicians than by others.

— Leonhard Euler, describing the Königsburg bridge problem
in a letter to Carl Leonhard Gottlieb Ehler (April 3, 1736)

I study my Bible as I gather apples.
First I shake the whole tree, that the ripest might fall.
Then I climb the tree and shake each limb,
and then each branch and then each twig,
and then I look under each leaf.

— Martin Luther

18 Basic Graph Algorithms

18.1 Definitions

A ***graph*** is normally defined as a pair of sets (V, E), where V is a set of arbitrary objects called ***vertices***[1] or ***nodes***. E is a set of pairs of vertices, which we call ***edges*** or (more rarely) ***arcs***. In an *undirected* graph, the edges are unordered pairs, or just sets of two vertices; I usually write $\boldsymbol{uv}$ instead of $\{u, v\}$ to denote the undirected edge between u and v. In a *directed* graph, the edges are ordered pairs of vertices; I usually write $\boldsymbol{u \to v}$ instead of (u, v) to denote the directed edge from u to v.

The definition of a graph as a pair of *sets* forbids graphs with loops (edges from a vertex to itself) and/or parallel edges (multiple edges with the same endpoints). Graphs *without* loops and parallel edges are often called ***simple*** graphs; non-simple graphs are sometimes called ***multigraphs***. Despite the formal definitional gap, most algorithms for simple graphs extend to non-simple graphs with little or no modification.

Following standard (but admittedly confusing) practice, I'll also use V to denote the *number* of vertices in a graph, and E to denote the *number* of edges. Thus, in any undirected graph we have $0 \le E \le \binom{V}{2}$, and in any directed graph we have $0 \le E \le V(V-1)$.

For any edge uv in an undirected graph, we call u a ***neighbor*** of v and vice versa. The ***degree*** of a node is its number of neighbors. In directed graphs, we have two kinds of neighbors. For any directed edge $u \to v$, we call u a ***predecessor*** of v and v a ***successor*** of u. The ***in-degree*** of a node is the number of predecessors, which is the same as the number of edges going into the node. The ***out-degree*** is the number of successors, or the number of edges going out of the node.

A graph $G' = (V', E')$ is a ***subgraph*** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A ***walk*** in a graph is a sequence of edges, where each successive pair of edges shares one vertex; a walk is called a ***path*** if it visits each vertex at most once. An undirected graph is ***connected*** if there is a walk (and therefore a path) between any two vertices. A disconnected graph consists of several ***components***, which are its maximal connected subgraphs. Two vertices are in the

[1] The singular of 'vertices' is <u>**vertex**</u>. The singular of 'matrices' is <u>**matrix**</u>. Unless you're speaking Italian, there is no such thing as a vertice, a matrice, an indice, an appendice, a helice, an apice, a vortice, a radice, a simplice, a codice, a directrice, a dominatrice, a Unice, a Kleenice, an Asterice, an Obelice, a Dogmatice, a Getafice, a Cacofonice, a Vitalstatistice, a Geriatrice, or Jimi Hendrice! You *will* lose points for using any of these so-called words.

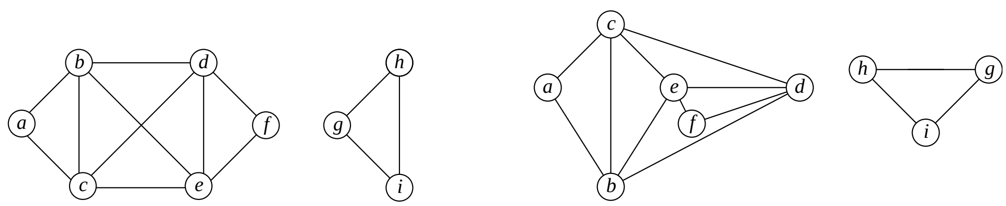
same component if and only if there is a path between them. Components are sometimes called "connected components", but this usage is redundant; components are connected by definition.

A ***cycle*** is a path that starts and ends at the same vertex, and has at least one edge. An undirected graph is ***acyclic*** if no subgraph is a cycle; acyclic graphs are also called ***forests***. A ***tree*** is a connected acyclic graph, or equivalently, one component of a forest. A ***spanning tree*** of a graph G is a subgraph that is a tree and contains every vertex of G. A graph has a spanning tree if and only if it is connected. A ***spanning forest*** of G is a collection of spanning trees, one for each connected component of G.

Directed graphs can contain directed paths and directed cycles. A directed graph is ***strongly connected*** if there is a directed path from any vertex to any other. A directed graph is ***acyclic*** if it does not contain a directed cycle; directed acyclic graphs are often called ***dags***.

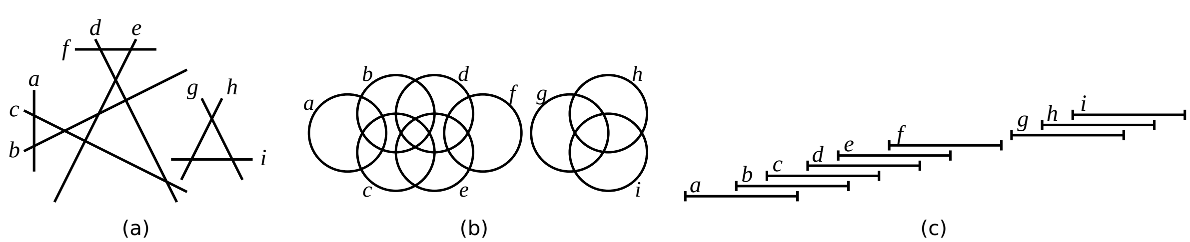
18.2 Abstract Representations and Examples

The most common way to visually represent graphs is with an ***embedding***. An embedding of a graph maps each vertex to a point in the plane (typically drawn as a small circle) and each edge to a curve or straight line segment between the two vertices. A graph is ***planar*** if it has an embedding where no two edges cross. The same graph can have many different embeddings, so it is important not to confuse a particular embedding with the graph itself. In particular, planar graphs can have non-planar embeddings!



A non-planar embedding of a planar graph with nine vertices, thirteen edges, and two components, and a planar embedding of the same graph.

However, embeddings are not the only useful representation of graphs. For example, the ***intersection graph*** of a collection of objects has a node for every object and an edge for every intersecting pair. Whether a particular graph can be represented as an intersection graph depends on what kind of object you want to use for the vertices. Different types of objects—line segments, rectangles, circles, etc.—define different classes of graphs. One particularly useful type of intersection graph is an *interval graph*, whose vertices are intervals on the real line, with an edge between any two intervals that overlap.



The example graph is also the intersection graph of (a) a set of line segments, (b) a set of circles, and (c) a set of intervals on the real line (stacked for visibility).

Another good example is the ***dependency graph*** of a recursive algorithm. Dependency graphs are directed acyclic graphs. The vertices are all the distinct recursive subproblems that arise when executing the algorithm on a particular input. There is an edge from one subproblem to another if evaluating the second subproblem requires a recursive evaluation of the first. For example, for the Fibonacci recurrence

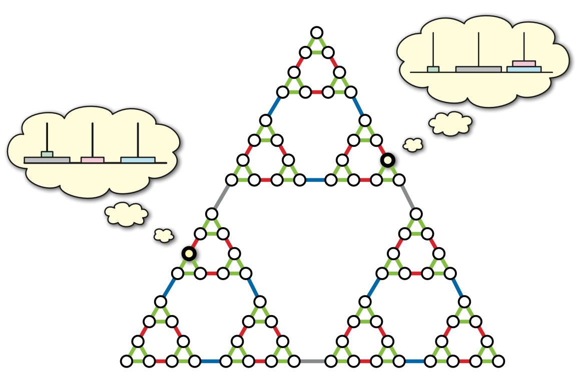
$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{otherwise,} \end{cases}$$

the vertices of the dependency graph are the integers $0, 1, 2, \ldots, n$, and the edges are the pairs $(i-1) \to i$ and $(i-2) \to i$ for every integer i between 2 and n. As a more complex example, consider the following recurrence, which solves a certain sequence-alignment problem called *edit distance*; see the dynamic programming notes for details:

$$Edit(i, j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{Bmatrix} Edit(i-1, j) + 1, \\ Edit(i, j-1) + 1, \\ Edit(i-1, j-1) + \big[A[i] \neq B[j]\big] \end{Bmatrix} & \text{otherwise} \end{cases}$$

The dependency graph of this recurrence is an $m \times n$ grid of vertices (i, j) connected by vertical edges $(i-1, j) \to (i, j)$, horizontal edges $(i, j-1) \to (i, j)$, and diagonal edges $(i-1, j-1) \to (i, j)$. Dynamic programming works efficiently for any recurrence that has a reasonably small dependency graph; a proper evaluation order ensures that each subproblem is visited *after* its predecessors.

Another interesting example is the ***configuration graph*** of a game, puzzle, or mechanism like tic-tac-toe, checkers, the Rubik's Cube, the Towers of Hanoi, or a Turing machine. The vertices of the configuration graph are all the valid configurations of the puzzle; there is an edge from one configuration to another if it is possible to transform one configuration into the other with a simple move. (Obviously, the precise definition depends on what moves are allowed.) Even for reasonably simple mechanisms, the configuration graph can be extremely complex, and we typically only have access to local information about the configuration graph.



The configuration graph of the 4-disk Tower of Hanoi.

Finite-state automata used in formal language theory can be modeled as labeled directed graphs. Recall that a deterministic finite-state automaton is formally defined as a 5-tuple $M = (\Sigma, Q, s, A, \delta)$, where Σ is a finite set called the *alphabet*, Q is a finite set of *states*, $s \in Q$ is the *start state*, $A \subseteq Q$ is the set of *accepting states*, and $\delta : Q \times \Sigma \to Q$ is a *transition function*. But it is often more useful to think of M as a directed graph G_M whose vertices are the states Q, and whose edges have the form $q \to \delta(q, a)$ for every state $q \in Q$ and symbol $a \in \Sigma$. Then basic questions about the language accepted by M can be phrased as questions about the graph G_M. For example, the language accepted by M is empty if and only if there is no path in G_M from the start state/vertex q_0 to an accepting state/vertex.

Finally, sometimes one graph can be used to implicitly represent other larger graphs. A good example of this implicit representation is the subset construction used to convert NFAs into DFAs. The subset construction can be generalized to *arbitrary* directed graphs as follows. Given *any* directed graph $G = (V, E)$, we can define a new directed graph $G' = (2^V, E')$ whose vertices are all *subsets* of vertices in V, and whose edges E' are defined as follows:

$$E' := \left\{ A \to B \;\middle|\; u \to v \in E \text{ for some } u \in A \text{ and } v \in B \right\}$$

We can mechanically translate this definition into an algorithm to construct G' from G, but strictly speaking, this construction is unnecessary, because ***G is already an implicit representation of G′***. Viewed in this light, the *incremental* subset construction used to convert NFAs to DFAs without unreachable states is just a breadth-first search of the implicitly-represented DFA.

It's important not to confuse these examples/representations of graphs with the actual formal *definition*: A graph is a pair of sets (V, E), where V is an arbitrary non-empty finite set, and E is a set of pairs (either ordered or unordered) of elements of V.

18.3 Graph Data Structures

In practice, graphs are represented by two data structures: *adjacency matrices*[2] and *adjacency lists*.

The ***adjacency matrix*** of a graph G is a $V \times V$ matrix, in which each entry indicates whether a particular edge is or is not in the graph:

$$A[i, j] := \big[(i, j) \in E\big].$$

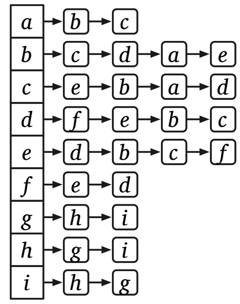
For undirected graphs, the adjacency matrix is always *symmetric*: $A[i, j] = A[j, i]$. Since we don't allow edges from a vertex to itself, the diagonal elements $A[i, i]$ are all zeros.

Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge just by looking in the appropriate slot in the matrix. We can also list all the neighbors of a vertex in $\Theta(V)$ time by scanning the corresponding row (or column). This is optimal in the worst case, since a vertex can have up to $V - 1$ neighbors; however, if a vertex has few neighbors, we may still have to examine every entry in the row to see them all. Similarly, adjacency matrices require $\Theta(V^2)$ space, regardless of how many edges the graph actually has, so it is only space-efficient for very *dense* graphs.

For *sparse* graphs—graphs with relatively few edges—adjacency lists are usually a better choice. An ***adjacency list*** is an array of linked lists, one list per vertex. Each linked list stores the neighbors of the corresponding vertex. For undirected graphs, each edge (u, v) is stored twice, once in u's neighbor list and once in v's neighbor list; for directed graphs, each edge is stored only once. Either way, the overall space required for an adjacency list is $O(V + E)$. Listing the neighbors of a node v takes $O(1 + \deg(v))$ time; just scan the neighbor list. Similarly, we can determine whether (u, v) is an edge in $O(1 + \deg(u))$ time by scanning the neighbor list of u. For

[2] See footnote 1.

	a	b	c	d	e	f	g	h	i
a	0	1	1	0	0	0	0	0	0
b	1	0	1	1	1	0	0	0	0
c	1	1	0	1	1	0	0	0	0
d	0	1	1	0	1	1	0	0	0
e	0	1	1	1	0	1	0	0	0
f	0	0	0	1	1	0	0	0	0
g	0	0	0	0	0	0	0	1	0
h	0	0	0	0	0	0	1	0	1
i	0	0	0	0	0	0	1	1	0



Adjacency matrix and adjacency list representations for the example graph.

undirected graphs, we can improve the time to $O(1 + \min\{\deg(u), \deg(v)\})$ by simultaneously scanning the neighbor lists of both u and v, stopping either we locate the edge or when we fall of the end of a list.

The adjacency list data structure should immediately remind you of hash tables with chaining; the two data structures are identical.[3] Just as with chained hash tables, we can make adjacency lists more efficient by using something other than a linked list to store the neighbors of each vertex. For example, if we use a hash table with constant load factor, when we can detect edges in $O(1)$ time, just as with an adjacency matrix. (Most hash give us only $O(1)$ *expected* time, but we can get $O(1)$ *worst-case* time using cuckoo hashing.)

The following table summarizes the performance of the various standard graph data structures. Stars* indicate expected amortized time bounds for maintaining dynamic hash tables.

	Adjacency matrix	Standard adjacency list (linked lists)	Adjacency list (hash tables)
Space	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Time to test if $uv \in E$	$O(1)$	$O(1+\min\{\deg(u),\deg(v)\}) = O(V)$	$O(1)$
Time to test if $u{\to}v \in E$	$O(1)$	$O(1+\deg(u)) = O(V)$	$O(1)$
Time to list the neighbors of v	$O(V)$	$O(1+\deg(v))$	$O(1+\deg(v))$
Time to list all edges	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Time to add edge uv	$O(1)$	$O(1)$	$O(1)^*$
Time to delete edge uv	$O(1)$	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$

At this point, one might reasonably wonder why anyone would ever use an adjacency matrix; after all, adjacency lists with hash tables support the same operations in the same time, using less space. Similarly, why would anyone use linked lists in an adjacency list structure to store neighbors, instead of hash tables? Although the main reason in practice is almost surely ***tradition***—If it was good enough for your grandfather's code, it should be good enough for yours!—there are some more principled arguments. One reason is that the standard adjacency lists are usually good enough; most graph algorithms never actually ask whether a given edge is present or absent! Another reason is that for sufficiently dense graphs, adjacency matrices are simpler and more efficient in practice, because they avoid the overhead of chasing pointers or computing hash functions.

But perhaps the most compelling reason is that many graphs are *implicitly* represented by adjacency matrices and standard adjacency lists. For example, intersection graphs are usually represented as a list of the underlying geometric objects. As long as we can test whether two

[3]For some reason, adjacency lists are always drawn with *horizontal* lists, while chained hash tables are always drawn with *vertical* lists. Don't ask me; I just work here.

objects intersect in constant time, we can apply any graph algorithm to an intersection graph by *pretending* that it is stored explicitly as an adjacency matrix.

Similarly, any data structure composed from records with pointers between them can be seen as a directed graph; graph algorithms can be applied to these data structures by *pretending* that the graph is stored in a standard adjacency list. Similarly, we can apply any graph algorithm to a configuration graph *as though* it were given to us as a standard adjacency list, provided we can enumerate all possible moves from a given configuration in constant time each. In both of these contexts, we can enumerate the edges leaving any vertex in time proportional to its degree, but we *cannot* necessarily determine in constant time if two vertices are connected. (Is there a pointer from this record to that record? Can we get from this configuration to that configuration in one move?) Thus, a standard adjacency list, with neighbors stored in linked lists, is the appropriate model data structure.

18.4 Traversing Connected Graphs

To keep things simple, we'll consider only undirected graphs for the rest of this lecture, although the algorithms also work for directed graphs with minimal changes.

Suppose we want to visit every node in a connected graph (represented either explicitly or implicitly). Perhaps the simplest graph-traversal algorithm is *depth-first search*. This algorithm can be written either recursively or iteratively. It's exactly the same algorithm either way; the only difference is that we can actually see the "recursion" stack in the non-recursive version. Both versions are initially passed a *source* vertex s.

```
RecursiveDFS(v):
  if v is unmarked
      mark v
      for each edge vw
          RecursiveDFS(w)
```

```
IterativeDFS(s):
  Push(s)
  while the stack is not empty
      v ← Pop
      if v is unmarked
          mark v
          for each edge vw
              Push(w)
```

Depth-first search is just one (perhaps the most common) species of a general family of graph traversal algorithms. The generic graph traversal algorithm stores a set of candidate edges in some data structure that I'll call a "bag". The only important properties of a "bag" are that we can put stuff into it and then later take stuff back out. (In C++ terms, think of the bag as a template for a real data structure.) A stack is a particular type of bag, but certainly not the only one. Here is the generic traversal algorithm:

```
Traverse(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge vw
              put w into the bag
```

This traversal algorithm clearly marks each vertex in the graph *at most* once. In order to show that it visits every node in a connected graph *at least* once, we modify the algorithm slightly; the modifications are highlighted in red. Instead of keeping vertices in the bag, the modified algorithm stores pairs of vertices. This modification allows us to remember, whenever we visit a vertex v for the first time, which previously-visited neighbor vertex put v into the bag. We call this earlier vertex the *parent* of v.

```
TRAVERSE(s):
  put (∅, s) in bag
  while the bag is not empty
      take (p, v) from the bag                 (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge vw                     (†)
              put (v, w) into the bag         (⋆⋆)
```

Lemma 1. *TRAVERSE(s) marks every vertex in any connected graph exactly once, and the set of pairs $(v, parent(v))$ with $parent(v) \neq \varnothing$ defines a spanning tree of the graph.*

Proof: The algorithm marks s. Let v be any vertex other than s, and let $(s, \ldots, u, v)$ be the path from s to v with the minimum number of edges. Since the graph is connected, such a path always exists. (If s and v are neighbors, then $u = s$, and the path has just one edge.) If the algorithm marks u, then it must put (u, v) into the bag, so it must later take (u, v) out of the bag, at which point v must be marked. Thus, by induction on the shortest-path distance from s, the algorithm marks every vertex in the graph, which implies that $parent(v)$ is well-defined for every vertex v.

The algorithm clearly marks every vertex at most once, so it must mark every vertex *exactly* once.

Call any pair $(v, parent(v))$ with $parent(v) \neq \varnothing$ a *parent edge*. For any node v, the path of parent edges $(v, parent(v), parent(parent(v)), \ldots)$ eventually leads back to s, so the set of parent edges form a connected graph. Clearly, both endpoints of every parent edge are marked, and the number of parent edges is exactly one less than the number of vertices. Thus, the parent edges form a spanning tree. □

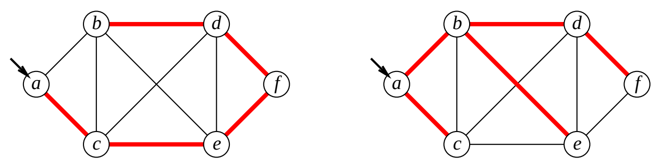
The exact running time of the traversal algorithm depends on how the graph is represented and what data structure is used as the 'bag', but we can make a few general observations. Because each vertex is marked at most once, the for loop (†) is executed at most V times. Each edge uv is put into the bag exactly twice; once as the pair (u, v) and once as the pair (v, u), so line (⋆⋆) is executed at most $2E$ times. Finally, we can't take more things out of the bag than we put in, so line (⋆) is executed at most $2E + 1$ times.

18.5 Examples

Let's first assume that the graph is represented by a standard adjacency list, so that the overhead of the for loop (†) is only constant time per edge.

- If we implement the 'bag' using a *stack*, we recover our original depth-first search algorithm. Each execution of (⋆) or (⋆⋆) takes constant time, so the algorithms runs in ***$O(V + E)$ time*** . If the graph is connected, we have $V \leq E + 1$, and so we can simplify the running time to $O(E)$. The spanning tree formed by the parent edges is called a ***depth-first spanning tree***. The exact shape of the tree depends on the start vertex and on the order that neighbors are visited in the for loop (†), but in general, depth-first spanning trees are long and skinny.

- If we use a *queue* instead of a stack, we get ***breadth-first search***. Again, each execution of ($\star$) or ($\star\star$) takes constant time, so the overall running time for connected graphs is still $\boldsymbol{O(E)}$. In this case, the ***breadth-first spanning tree*** formed by the parent edges contains ***shortest paths*** from the start vertex s to every other vertex in its connected component. We'll see shortest paths again in a future lecture. Again, exact shape of a breadth-first spanning tree depends on the start vertex and on the order that neighbors are visited in the for loop (†), but in general, breadth-first spanning trees are short and bushy.



A depth-first spanning tree and a breadth-first spanning tree
of one component of the example graph, with start vertex a.

- Now suppose the edges of the graph are weighted. If we implement the 'bag' using a *priority queue*, always extracting the minimum-weight edge in line ($\star$), the resulting algorithm is reasonably called ***shortest-first search***. In this case, each execution of ($\star$) or ($\star\star$) takes $O(\log E)$ time, so the overall running time is $\boldsymbol{O(V + E \log E)}$, which simplifies to $\boldsymbol{O(E \log E)}$ if the graph is connected. For this algorithm, the set of parent edges form the ***minimum spanning tree*** of the connected component of s. Surprisingly, as long as all the edge weights are distinct, the resulting tree does *not* depend on the start vertex or the order that neighbors are visited; in this case, there is only one minimum spanning tree. We'll see minimum spanning trees again in the next lecture.

If the graph is represented using an adjacency matrix instead of an adjacency list, finding all the neighbors of each vertex in line (†) takes $O(V)$ time. Thus, depth- and breadth-first search each run in $O(V^2)$ time, and 'shortest-first search' runs in $O(V^2 + E \log E) = O(V^2 \log V)$ time.

18.6 Searching Disconnected Graphs

If the graph is disconnected, then TRAVERSE(s) only visits the nodes in the connected component of the start vertex s. If we want to visit all the nodes in every component, we can use the following 'wrapper' around our generic traversal algorithm. Since TRAVERSE computes a spanning tree of one component, TRAVERSEALL computes a spanning *forest* of the entire graph.

```
TRAVERSEALL(s):
  for all vertices v
      if v is unmarked
          TRAVERSE(v)
```

Surprisingly, a few well-known algorithms textbooks claim that this wrapper can only be used with depth-first search. They're wrong.

Exercises

1. Prove that the following definitions are all equivalent.

    - A tree is a connected acyclic graph.

- A tree is one component of a forest.
- A tree is a connected graph with *at most* $V-1$ edges.
- A tree is a minimal connected graph; removing any edge makes the graph disconnected.
- A tree is an acyclic graph with *at least* $V-1$ edges.
- A tree is a maximal acyclic graph; adding an edge between any two vertices creates a cycle.

2. Prove that any connected acyclic graph with $n \geq 2$ vertices has at least two vertices with degree 1. Do not use the words "tree" or "leaf", or any well-known properties of trees; your proof should follow entirely from the definitions of "connected" and "acyclic".

3. Let G be a connected graph, and let T be a depth-first spanning tree of G rooted at some node v. Prove that if T is also a breadth-first spanning tree of G rooted at v, then $G = T$.

4. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

   (a) Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left. Pretty please.

   (b) Suppose you are given a directed graph representing the pecking relationships among a set of n pigeons. The graph contains one vertex per pigeon, and it contains an edge $i \to j$ if and only if pigeon i pecks pigeon j. Describe and analyze an algorithm to compute a pecking order for the pigeons, as guaranteed by part (a).

5. A graph (V, E) is *bipartite* if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.

   (a) Prove that every tree is a bipartite graph.

   (b) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

6. An ***Euler tour*** of a graph G is a closed walk through G that traverses every edge of G exactly once.

   (a) Prove that a connected graph G has an Euler tour if and only if every vertex has even degree.

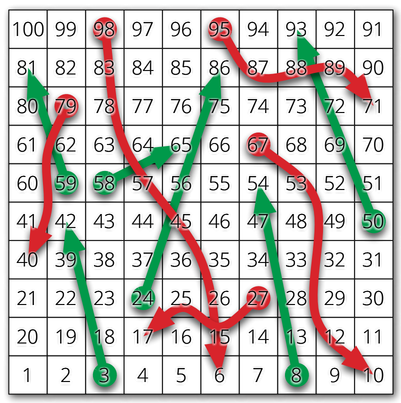
   (b) Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such graph exists.

7. The d-dimensional hypercube is the graph defined as follows. There are $2d$ vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if their labels differ in exactly one bit.

   (a) A Hamiltonian cycle in a graph G is a cycle of edges in G that visits every vertex of G exactly once. Prove that for all $d \geq 2$, the d-dimensional hypercube has a Hamiltonian cycle.

   (b) Which hypercubes have an Euler tour (a closed walk that traverses every edge exactly once)? *[Hint: This is very easy.]*

8. ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.



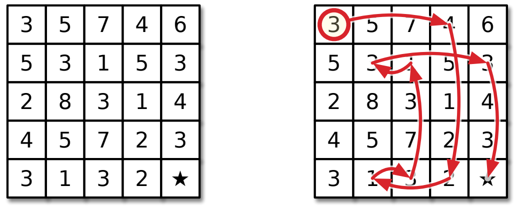
A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k. If the token ends the move at the *top* end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the *bottom* end of a ladder, it climbs up to the top of that ladder.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

9. A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.

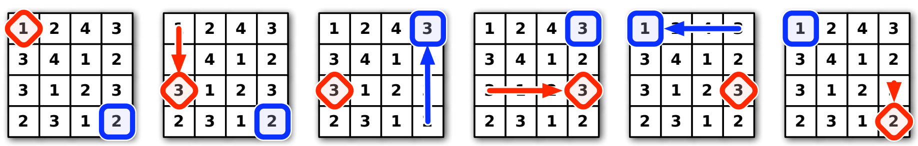
Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution.



A 5 × 5 number maze that can be solved in eight moves.

10. The following puzzle was invented by the infamous Mongolian puzzle-warrior Vidrach Itky Leda in the year 1473. The puzzle consists of an $n \times n$ grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

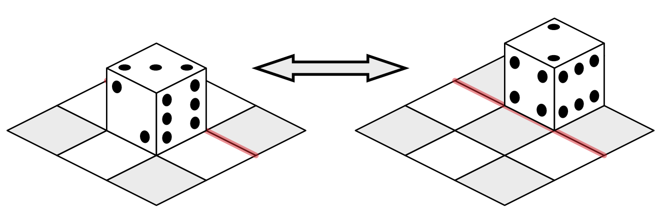
    In a single turn, you may move either token up, right, down, or left *by a distance determined by the **other** token*. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move a token off the grid or to the same square as the other token.



A five-move solution for a 4 × 4 Vidrach Itky Leda puzzle.

    Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle above, your algorithm would return the number 5.

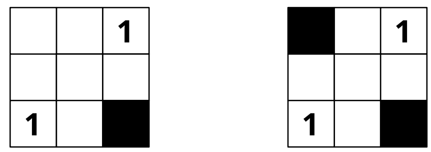
11. A *rolling die maze* is a puzzle involving a standard six-sided die (a cube with numbers on each side) and a grid of squares. You should imagine the grid lying on top of a table; the die always rests on and exactly covers one square. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.



Rolling a die.

    Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number of pips on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

For example, here are two rolling die mazes. Black squares are blocked. The maze on the left can be solved by placing the die on the lower left square with 1 pip on the top face, and then rolling it north, then north, then east, then east. The maze on the right is not solvable.



Two rolling die mazes. Only the maze on the left is solvable.

(a) Suppose the input is a two-dimensional array $L[1..n][1..n]$, where each entry $L[i][j]$ stores the label of the square in the ith row and jth column, where 0 means the square is free and -1 means the square is blocked. Describe and analyze a polynomial-time algorithm to determine whether the given rolling die maze is solvable.

*(b) Now suppose the maze is specified *implicitly* by a list of labeled and blocked squares. Specifically, suppose the input consists of an integer M, specifying the height and width of the maze, and an array $S[1..n]$, where each entry $S[i]$ is a triple (x, y, L) indicating that square (x, y) has label L. As in the explicit encoding, label -1 indicates that the square is blocked; free squares are not listed in S at all. Describe and analyze an efficient algorithm to determine whether the given rolling die maze is solvable. For full credit, the running time of your algorithm should be polynomial in the input size n.

[Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can only be solved if the initial position is chosen correctly.]

12. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[4] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

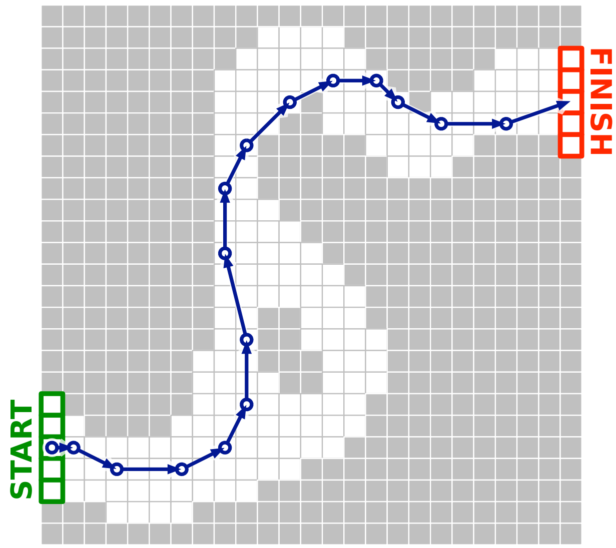
    Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0, 0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position inside the finishing area.

[4] The actual game is a bit more complicated than the version described here. See http://harmmade.com/vectorracer/ for an excellent online version.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the 'starting area' is the first column, and the 'finishing area' is the last column.

Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack. *[Hint: Build a graph. What are the vertices? What are the edges? What problem is this?]*

velocity	position
$(0,0)$	$(1,5)$
$(1,0)$	$(2,5)$
$(2,-1)$	$(4,4)$
$(3,0)$	$(7,4)$
$(2,1)$	$(9,5)$
$(1,2)$	$(10,7)$
$(0,3)$	$(10,10)$
$(-1,4)$	$(9,14)$
$(0,3)$	$(9,17)$
$(1,2)$	$(10,19)$
$(2,2)$	$(12,21)$
$(2,1)$	$(14,22)$
$(2,0)$	$(16,22)$
$(1,-1)$	$(17,21)$
$(2,-1)$	$(19,20)$
$(3,0)$	$(22,20)$
$(3,1)$	$(25,21)$



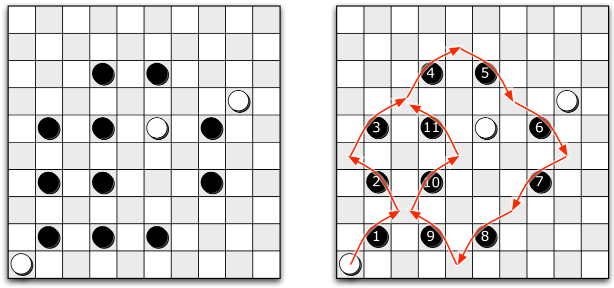
A 16-step Racetrack run, on a 25×25 track. This is *not* the shortest run on this track.

*13. Draughts (also known as checkers) is a game played on an $m \times m$ grid of squares, alternately colored light and dark. (The game is usually played on an 8×8 or 10×10 board, but the rules easily generalize to any board size.) Each dark square is occupied by at most one game piece (usually called a *checker* in the U.S.), which is either black or white; light squares are always empty. One player ('White') moves the white pieces; the other ('Black') moves the black pieces.

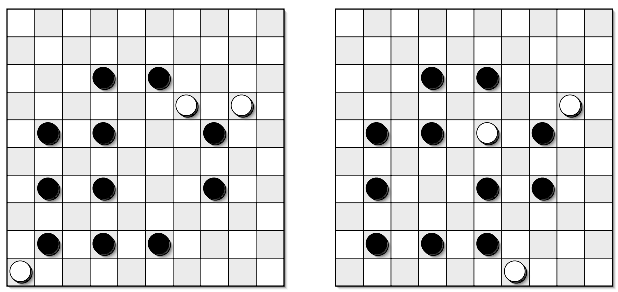
Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[5] Pieces can be moved in any of the four diagonal directions, either one or two steps at a time. On each turn, a player either *moves* one of her pieces one step diagonally into an empty square, or makes a series of *jumps* with one of her checkers. In a single jump, a piece moves to an empty square two steps away in any diagonal direction, but only if the intermediate square is occupied by a piece of the opposite color; this enemy piece is *captured* and immediately removed from the board. Multiple jumps are allowed in a single turn as long as they are made by the same piece. A player wins if her opponent has no pieces left on the board.

[5]Most other variants of draughts have 'flying kings', which behave *very* differently than what's described here. In particular, if we allow flying kings, it is actually NP-hard to determine which move captures the most enemy pieces. The most common international version of draughts also has a forced-capture rule, which *requires* each player to capture the maximum possible number of enemy pieces in each move. Thus, just following the rules is NP-hard!

Describe an algorithm that correctly determines whether White can capture every black piece, thereby winning the game, *in a single turn*. The input consists of the width of the board (m), a list of positions of white pieces, and a list of positions of black pieces. For full credit, your algorithm should run in $O(n)$ time, where n is the total number of pieces. *[Hint: The greedy strategy—make arbitrary jumps until you get stuck—does* ***not*** *always find a winning sequence of jumps even when one exists. See problem* **??**. *Parity, parity, parity.]*



White wins in one turn.



White cannot win in one turn from either of these positions.

Ts'ui Pe must have said once: I am withdrawing to write a book.
And another time: I am withdrawing to construct a labyrinth.
Every one imagined two works;
to no one did it occur that the book and the maze were one and the same thing.

— Jorge Luis Borges, "El jardín de senderos que se bifurcan" (1942)
English translation ("The Garden of Forking Paths") by Donald A. Yates (1958)

"Com'è bello il mondo e come sono brutti i labirinti!" dissi sollevato.
"Come sarebbe bello il mondo se ci fosse una regola per girare nei labirinti,"
rispose il mio maestro.

["How beautiful the world is, and how ugly labyrinths are," I said, relieved.
"How beautiful the world would be if there were a procedure for moving through labyrinths,"
my master replied.]

— Umberto Eco, *Il nome della rosa* (1980)
English translation (*The Name of the Rose*) by William Weaver (1983)

At some point, the learning stops and the pain begins.

— Rao Kosaraju

19 Depth-First Search

Recall from the previous lecture the recursive formulation of depth-first search in undirected graphs.

```
DFS(v):
  if v is unmarked
      mark v
      for each edge vw
          DFS(w)
```

We can make this algorithm slightly faster (in practice) by checking whether a node is marked *before* we recursively explore it. This modification ensures that we call DFS(v) only once for each vertex v. We can further modify the algorithm to define parent pointers and other useful information about the vertices. This additional information is computed by two black-box subroutines PreVisit and PostVisit, which we leave unspecified for now.

```
DFS(v):
  mark v
  PreVisit(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  PostVisit(v)
```

We can search any *connected* graph by unmarking all vertices and then calling DFS(s) for an arbitrary start vertex s. As we argued in the previous lecture, the subgraph of all parent edges $v \rightarrow parent(v)$ defines a spanning tree of the graph, which we consider to be rooted at the start vertex s.

Lemma 1. *Let T be a depth-first spanning tree of a connected undirected graph G, computed by calling DFS(s). For any node v, the vertices that are marked during the execution of DFS(v) are the proper descendants of v in T.*

Proof: T is also the recursion tree for DFS(s). □

Lemma 2. *Let T be a depth-first spanning tree of a connected undirected graph G. For every edge vw in G, either v is an ancestor of w in T, or v is a descendant of w in T.*

Proof: Assume without loss of generality that v is marked before w. Then w is unmarked when DFS(v) is invoked, but marked when DFS(v) returns, so the previous lemma implies that w is a proper descendant of v in T. □

Lemma 2 implies that any depth-first spanning tree T divides the edges of G into two classes: *tree* edges, which appear in T, and *back* edges, which connect some node in T to one of its ancestors.

19.1 Counting and Labeling Components

For graphs that might be disconnected, we can compute a depth-first spanning *forest* by calling the following wrapper function; again, we introduce a generic black-box subroutine PREPROCESS to perform any necessary preprocessing for the POSTVISIT and POSTVISIT functions.

```
DFSALL(G):
  PREPROCESS(G)
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

With very little additional effort, we can count the components of a graph; we simply increment a counter inside the wrapper function. Moreover, we can also record which component contains each vertex in the graph by passing this counter to DFS. The single line $comp(v) \leftarrow count$ is a trivial example of PREVISIT. (And the absence of code after the for loop is a vacuous example of POSTVISIT.)

```
COUNTANDLABEL(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          LABELCOMPONENT(v, count)
  return count
```

```
LABELCOMPONENT(v, count):
  mark v
  comp(v) ← count
  for each edge vw
      if w is unmarked
          LABELCOMPONENT(w, count)
```

It should be emphasized that depth-first search is not specifically required here; any other instantiation of our earlier generic traversal algorithm ("whatever-first search") can be used to count components in the same asymptotic running time. However, most of the other algorithms we consider in this note *do* specifically require *depth*-first search.

19.2 Preorder and Postorder Labeling

You should already be familiar with preorder and postorder traversals of rooted trees, both of which can be computed using from depth-first search. Similar traversal orders can be defined for arbitrary graphs by passing around a counter as follows:

```
PrePostLabel(G):
  for all vertices v
      unmark v
  clock ← 0
  for all vertices v
      if v is unmarked
          clock ← LabelComponent(v, clock)
```

```
LabelComponent(v, clock):
  mark v
  pre(v) ← clock
  clock ← clock + 1
  for each edge vw
      if w is unmarked
          clock ← LabelComponent(w, clock)
  post(v) ← clock
  clock ← clock + 1
  return clock
```

Equivalently, if we're willing to use (shudder) global variables, we can use our generic depth-first-search algorithm with the following subroutines PREPROCESS, PREVISIT, and POSTVISIT.

```
Preprocess(G):
  clock ← 0
```

```
PreVisit(v):
  pre(v) ← clock
  clock ← clock + 1
```

```
PostVisit(v):
  post(v) ← clock
  clock ← clock + 1
```

Consider two vertices u and v, where u is marked after v. Then we must have $pre(u) < pre(v)$. Moreover, Lemma 1 implies that if v is a descendant of u, then $post(u) > post(v)$, and otherwise, $pre(v) > post(u)$. Thus, for any two vertices u and v, the intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or nested; in particular, if uv is an edge, Lemma 2 implies that the intervals must be nested.

19.3 Directed Graphs and Reachability

The recursive algorithm requires only one minor change to handle directed graphs:

```
DFSAll(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

```
DFS(v):
  mark v
  PreVisit(v)
  for each edge v→w
      if w is unmarked
          DFS(w)
  PostVisit(v)
```

However, we can no longer use this modified algorithm to count components. Suppose G is a single directed path. Depending on the order that we choose to visit the nodes in DFSALL, we may discover any number of "components" between 1 and n. All that we can guarantee is that the "component" numbers computed by DFSALL do not increase as we traverse the path. In fact, the real problem is that the *definition* of "component" is only suitable for *undirected* graphs.

Instead, for directed graphs we rely on a more subtle notion of ***reachability***. We say that a node v is *reachable* from another node u in a directed graph G—or more simply, that u can reach v—if and only if there is a directed path in G from u to v. Let ***Reach*(*u*)** denote the set of vertices that are reachable from u (including u itself). A simple inductive argument proves that $Reach(u)$ is precisely the subset of nodes that are marked by calling DFS(u).

19.4 Directed Acyclic Graphs

A ***directed acyclic graph*** or ***dag*** is a directed graph with no directed cycles. Any vertex in a dag that has no incoming vertices is called a ***source***; any vertex with no outgoing edges is called a ***sink***. Every dag has at least one source and one sink (Do you see why?), but may have more than one of each. For example, in the graph with n vertices but no edges, every vertex is a source and every vertex is a sink.

We can check whether a given directed graph G is a dag in $O(V + E)$ time as follows. First, to simplify the algorithm, we add a single artificial source s, with edges from s to every other vertex. Let $G + s$ denote the resulting augmented graph. Because s has no outgoing edges, no directed cycle in $G + s$ goes through s, which implies that $G + s$ is a dag if and only if G is a dag. Then we preform a depth-first search of $G + s$ starting at the new source vertex s; by construction every other vertex is reachable from s, so this search visits every node in the graph.

Instead of vertices being merely marked or unmarked, each vertex has one of three statuses—New, Active, or Done—which depend on whether we have started or finished the recursive depth-first search at that vertex. (Since this algorithm never uses parent pointers, I've removed the line "$parent(w) \leftarrow v$".)

```
IsAcyclic(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← New
  return IsAcyclicDFS(s)
```

```
IsAcyclicDFS(v):
  status(v) ← Active
  for each edge v→w
      if status(w) = Active
          return False
      else if status(w) = New
          if IsAcyclicDFS(w) = False
              return False
  status(v) ← Done
  return True
```

Suppose the algorithm returns False. Then the algorithm must discover an edge $v \to w$ such that $status(w) = \text{Active}$. The active vertices are precisely the vertices currently on the recursion stack, which are all ancestors of the current vertex v. Thus, there is a directed path from w to v, and so the graph has a directed cycle.

On the other hand, suppose G has a directed cycle. Let w be the first vertex in this cycle that we visit, and let $v \to w$ be the edge leading into v in the same cycle. Because there is a directed path from w to v, we must call IsAcyclicDFS(v) during the execution of IsAcyclicDFS(w), unless we discover some other cycle first. During the execution of IsAcyclicDFS(v), we consider the edge $v \to w$, discover that $status(w) = \text{Active}$. The return value False bubbles up through all the recursive calls to the top level.

We conclude that IsAcyclic(G) returns True if and only if G is a dag.

19.5 Topological Sort

A ***topological ordering*** of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \to v$. Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right. A topological ordering is clearly impossible if the graph G has a directed cycle—the rightmost vertex of the cycle would have an edge pointing to the left! On the other hand, every dag has a topological order, which can be computed by either of the following algorithms.

```
TopologicalSort(G):
  n ← |V|
  for i ← 1 to n
      v ← any source in G
      S[i] ← v
      delete v and all edges leaving v
  return S[1..n]
```

```
TopologicalSort(G):
  n ← |V|
  for i ← n down to 1
      v ← any sink in G
      S[1] ← v
      delete v and all edges entering v
  return S[1..n]
```

The correctness of these algorithms follow inductively from the observation that *deleting* a vertex cannot *create* a cycle.

This simple algorithm has two major disadvantages. First, the algorithm actually destroys the input graph. This destruction can be avoided by simply marking the "deleted" vertices, instead of actually deleting them, and defining a vertex to be a source (sink) if none of its incoming (outgoing) edges come from (lead to) an unmarked vertex. The more serious problem is that finding a source vertex seems to require $\Theta(V)$ time in the worst case, which makes the running time of this algorithm $\Theta(V^2)$. In fact, a careful implementation of this algorithm computes a topological ordering in $O(V+E)$ time without removing any edges.

But there is a simpler linear-time algorithm based on our earlier algorithm for deciding whether a directed graph is acyclic. The new algorithm is based on the following observation:

Lemma 3. *For any directed acyclic graph G, the first vertex marked* Done *by* IsAcyclic(G) *must be a sink.*

Proof: Let v be the first vertex marked Done during an execution of IsAcyclic. For the sake of argument, suppose v has an outgoing edge $v \to w$. When IsAcyclicDFS first considers the edge $v \to w$, there are three cases to consider.

- If $status(w) =$ Done, then w is marked Done before v, which contradicts the definition of v.
- If $status(w) =$ New, the algorithm calls TopoSortDFS(w), which (among other computation) marks w Done. Thus, w is marked Done before v, which contradicts the definition of v.
- If $status(w) =$ Active, then G has a directed cycle, contradicting our assumption that G is acyclic.

In all three cases, we have a contradiction, so v must be a sink. □

Thus, to topologically sort a dag G, it suffice to list the vertices in the *reverse* order of being marked Done. For example, we could push each vertex onto a stack when we mark it Done, and then pop every vertex off the stack.

```
TopologicalSort(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← New

  TopoSortDFS(s)

  for i ← 1 to V
      S[i] ← Pop
  return S[1..V]
```

```
TopoSortDFS(v):
  status(v) ← Active
  for each edge v→w
      if status(w) = New
          ProcessBackwardDFS(w)
      else if status(w) = Active
          fail gracefully
  status(v) ← Done
  Push(v)
  return True
```

But maintaining a separate data structure is actually overkill. In most applications of topological sort, an explicit sorted list of the vertices is not our actual goal; instead, we want to performing some fixed computation at each vertex of the graph, either in topological order or in reverse topological order. In this case, it is not necessary to *record* the topological order. To process the graph in *reverse* topological order, we can just process each vertex at the end of its recursive depth-first search.

```
PROCESSBACKWARD(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← NEW
  PROCESSBACKWARDDFS(s)
```

```
PROCESSBACKWARDDFS(v):
  status(v) ← ACTIVE
  for each edge v→w
      if status(w) = NEW
          PROCESSBACKWARDDFS(w)
      else if status(w) = ACTIVE
          fail gracefully
  status(v) ← DONE
  PROCESS(v)
```

If we already *know* that the input graph is acyclic, we can simplify the algorithm by simply marking vertices instead of labeling them ACTIVE or DONE.

```
PROCESSDAGBACKWARD(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      unmark v
  PROCESSDAGBACKWARDDFS(s)
```

```
PROCESSDAGBACKWARDDFS(v):
  mark v
  for each edge v→w
      if w is unmarked
          PROCESSDAGBACKWARDDFS(w)
  PROCESS(v)
```

Except for the addition of the artificial source vertex s, which we need to ensure that every vertex is visited, this is just the standard depth-first search algorithm, with POSTVISIT renamed to PROCESS!

The simplest way to process a dag in *forward* topological order is to construct the ***reversal*** of the input graph, which is obtained by replacing each each $v \rightarrow w$ with its reversal $w \rightarrow v$. Reversing a directed cycle gives us another directed cycle with the opposite orientation, so the reversal of a dag is another dag. Every source in G becomes a sink in the reversal of G and vice versa; it follows inductively that every topological ordering for the reversal of G is the reversal of a topological ordering of G. The reversal of any directed graph can be computed in $O(V + E)$ time; the details of this construction are left as an easy exercise.

19.6 Every Dynamic Programming Algorithm?

Our topological sort algorithm is arguably the model fora wide class of dynamic programming algorithms. Recall that the ***dependency graph*** of a recurrence has a vertex for every recursive subproblem and an edge from one subproblem to another if evaluating the first subproblem requires a recursive evaluation of the second. The dependency graph must be acyclic, or the naïve recursive algorithm would never halt. Evaluating any recurrence with memoization is exactly the same as performing a depth-first search of the dependency graph. In particular, a vertex of the dependency graph is 'marked' if the value of the corresponding subproblem has already been computed, and the black-box subroutine PROCESS is a placeholder for the actual value computation.

However, there are some minor differences between most dynamic programming algorithms and topological sort.

- First, in most dynamic programming algorithms, the dependency graph is *implicit*—the nodes and edges are not given as part of the input. But this difference really is minor; as long as we can enumerate recursive subproblems in constant time each, we can traverse the dependency graph exactly as if it were explicitly stored in an adjacency list.

- More significantly, most dynamic programming recurrences have highly structured dependency graphs. For example, the dependency graph for edit distance is a regular grid with diagonals, and the dependency graph for optimal binary search trees is an upper triangular grid with all possible rightward and upward edges. This regular structure lets us hard-wire a topological order directly into the algorithm, so we don't have to compute it at run time.

Conversely, we can use depth-first search to build dynamic programming algorithms for problems with less structured dependency graphs. For example, consider the ***longest path*** problem, which asks for the path of *maximum* total weight from one node s to another node t in a directed graph G with weighted edges. The longest path problem is NP-hard in general directed graphs, by an easy reduction from the traveling salesman problem, but it is easy to solve in linear time if the input graph G is acyclic, as follows. For any node s, let $LLP(s,t)$ denote the Length of the Longest Path in G from s to t. If G is a dag, this function satisfies the recurrence

$$LLP(s,t) = \begin{cases} 0 & \text{if } s = t, \\ \max_{s\to v} \left(\ell(s\to v) + LLP(v,t)\right) & \text{otherwise,} \end{cases}$$

where $\ell(v\to w)$ is the given weight ("length") of edge $v\to w$. In particular, if s is a *sink* but not equal to t, then $LLP(s,t) = \infty$. The dependency graph for this recurrence is the input graph G itself: subproblem $LLP(u,t)$ depends on subproblem $LLP(v,t)$ if and only if $u\to v$ is an edge in G. Thus, we can evaluate this recursive function in $O(V+E)$ time by performing a depth-first search of G, starting at s.

```
LongestPath(s, t):
  if s = t
      return 0

  if LLP(s) is undefined
      LLP(s) ← ∞
      for each edge s→v
          LLP(s) ← max {LLP(v), ℓ(s→v) + LongestPath(v, t)}
  return LLP(s)
```

A surprisingly large number of dynamic programming problems (but *not* all) can be recast as optimal path problems in the associated dependency graph.

19.7 Strong Connectivity

Let's go back to the proper definition of connectivity in directed graphs. Recall that one vertex u can *reach* another vertex v in a graph G if there is a directed path in G from u to v, and that *Reach*(u) denotes the set of all vertices that u can reach. Two vertices u and v are ***strongly connected*** if u can reach v and v can reach u. Tedious definition-chasing implies that strong connectivity is an equivalence relation over the set of vertices of any directed graph, just as connectivity is for undirected graphs. The equivalence classes of this relation are called the ***strongly connected components*** (or more simply, the ***strong components***) of G. If G has a single strong component, we call it ***strongly connected***. G is a directed acyclic graph if and only if every strong component of G is a single vertex.

It is straightforward to compute the strong component containing a single vertex v in $O(V+E)$ time. First we compute *Reach*(v) by calling *DFS*(v). Then we compute $\mathit{Reach}^{-1}(v) = \{u \mid v \in \mathit{Reach}(u)\}$ by searching the reversal of G. Finally, the strong component of v is the intersection $\mathit{Reach}(v) \cap \mathit{Reach}^{-1}(v)$. In particular, we can determine whether the entire graph is strongly connected in $O(V+E)$ time.

We can compute *all* the strong components in a directed graph by wrapping the single-strong-component algorithm in a wrapper function, just as we did for depth-first search in undirected graphs. However, the resulting algorithm runs in $O(VE)$ time; there are at most V strong components, and each requires $O(E)$ time to discover. Surely we can do better! After all, we only need $O(V+E)$ time to decide whether every strong component is a single vertex.

19.8 Strong Components in Linear Time

For any directed graph G, the ***strong component graph*** $\boldsymbol{scc(G)}$ is another directed graph obtained by contracting each strong component of G to a single (meta-)vertex and collapsing parallel edges. The strong component graph is sometimes also called the *meta-graph* or *condensation* of G. It's not hard to prove (hint, hint) that $scc(G)$ is always a dag. Thus, in principle, it is possible to topologically order the strong components of G; that is, the vertices can be ordered so that every *backward* edge joins two edges in the same strong component.

Let C be any strong component of G that is a sink in $scc(G)$; we call C a *sink component*. Every vertex in C can reach every other vertex in C, so a depth-first search from any vertex in C visits every vertex in C. On the other hand, because C is a sink component, there is no edge from C to any other strong component, so a depth-first search starting in C visits *only* vertices in C. So if we can compute all the strong components as follows:

```
StrongComponents(G):
  count ← 0
  while G is non-empty
      count ← count + 1
      v ← any vertex in a sink component of G
      C ← OneComponent(v, count)
      remove C and incoming edges from G
```

At first glance, finding a vertex in a sink component *quickly* seems quite hard. However, we can quickly find a vertex in a *source* component using the standard depth-first search. A source component is a strong component of G that corresponds to a source in $scc(G)$. Specifically, we compute *finishing times* (otherwise known as post-order labeling) for the vertices of G as follows.

```
DFSAll(G):
  for all vertices v
      unmark v
  clock ← 0
  for all vertices v
      if v is unmarked
          clock ← DFS(v, clock)
```

```
DFS(v, clock):
  mark v
  for each edge v→w
      if w is unmarked
          clock ← DFS(w, clock)
  clock ← clock + 1
  finish(v) ← clock
  return clock
```

Lemma 4. *The vertex with largest finishing time lies in a source component of G.*

Proof: Let v be the vertex with largest finishing time. Then DFS$(v, clock)$ must be the last direct call to DFS made by the wrapper algorithm DFSAll.

Let C be the strong component of G that contains v. For the sake of argument, suppose there is an edge $x \rightarrow y$ such that $x \notin C$ and $y \in C$. Because v and y are strongly connected, y can reach v, and therefore x can reach v. There are two cases to consider.

- If x is already marked when DFS(v) begins, then v must have been marked during the execution of DFS(x), because x can reach v. But then v was already marked when DFS(v) was called, which is impossible.
- If x is not marked when DFS(v) begins, then x must be marked during the execution of DFS(v), which implies that v can reach x. Since x can also reach v, we must have $x \in C$, contradicting the definition of x.

We conclude that C is a source component of G. □

Essentially the same argument implies the following more general result.

Lemma 5. *For any edge $v \rightarrow w$ in G, if* finish(v) < finish(w)*, then v and w are strongly connected in G.*

Proof: Let $v \rightarrow w$ be an arbitrary edge of G. There are three cases to consider.If w is unmarked when DFS(v) begins, then the recursive call to DFS(w) finishes w, which implies that *finish*(w) < *finish*(v). If w is still active when DFS(v) begins, there must be a path from w to v, which implies that v and w are strongly connected. Finally, if w is finished when DFS(v) begins, then clearly *finish*(w) < *finish*(v). □

This observation is consistent with our earlier topological sorting algorithm; for *every* edge $v \rightarrow w$ in a directed acyclic graph, we have *finish*(v) > *finish*(w).

It is easy to check (hint, hint) that any directed G has exactly the same strong components as its reversal *rev*(G); in fact, we have *rev*(*scc*(G)) = *scc*(*rev*(G)). Thus, if we order the vertices of G by their finishing times in DFSALL(*rev*(G)), the *last* vertex in this order lies in a sink component of G. Thus, if we run DFSALL(G), visiting vertices in reverse order of their finishing times in DFSALL(*rev*(G)), then each call to DFS visits exactly one strong component of G.

Putting everything together, we obtain the following algorithm to count and label the strong components of a directed graph in $O(V + E)$ time, first discovered (but never published) by Rao Kosaraju in 1978, and then independently rediscovered by Micha Sharir in 1981. The Kosaraju-Sharir algorithm has two phases. The first phase performs a depth-first search of the reversal of G, pushing each vertex onto a stack when it is finished. In the second phase, we perform another depth-first search of the original graph G, considering vertices in the order they appear on the stack.

```
KosarajuSharir(G):
  ⟨⟨Phase 1: Push in finishing order⟩⟩
  unmark all vertices
  for all vertices v
      if v is unmarked
          clock ← RevPushDFS(v)
  ⟨⟨Phase 2: DFS in stack order⟩⟩
  unmark all vertices
  count ← 0
  while the stack is non-empty
      v ← Pop
      if v is unmarked
          count ← count + 1
          LabelOneDFS(v, count)
```

```
RevPushDFS(v):
  mark v
  for each edge v→u in rev(G)
      if u is unmarked
          RevPushDFS(u)
  Push(v)
```

```
LabelOneDFS(v, count):
  mark v
  label(v) ← count
  for each edge v→w in G
      if w is unmarked
          LabelOneDFS(w, count)
```

With further minor modifications, we can also compute the strongly connected component graph $scc(G)$ in $O(V+E)$ time.

Exercises

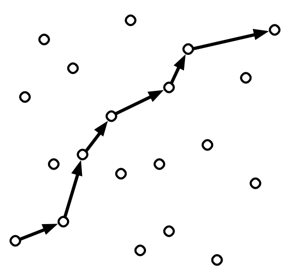
0. (a) Describe an algorithm to compute the reversal $rev(G)$ of a directed graph in $O(V+E)$ time.

   (b) Prove that for any directed graph G, the strong component graph $scc(G)$ is a dag.

   (c) Prove that for any directed graph G, we have $scc(rev(G)) = rev(scc(G))$.

   (d) Suppose S and T are two strongly connected components in a directed graph G. Prove that $finish(u) < finish(v)$ for all vertices $u \in S$ and $v \in T$.

1. A ***polygonal path*** is a sequence of line segments joined end-to-end; the endpoints of these line segments are called the ***vertices*** of the path. The ***length*** of a polygonal path is the sum of the lengths of its segments. A polygonal path with vertices $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ is ***monotonically increasing*** if $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for every index i—informally, each vertex of the path is above and to the right of its predecessor.



A monotonically increasing polygonal path with seven vertices through a set of points

   Suppose you are given a set S of n points in the plane, represented as two arrays $X[1..n]$ and $Y[1..n]$. Describe and analyze an algorithm to compute the length of the maximum-length monotonically increasing path with vertices in S. Assume you have a subroutine $\text{Length}(x, y, x', y')$ that returns the length of the segment from (x, y) to (x', y').

2. Let $G = (V, E)$ be a given directed graph.

   (a) The *transitive closure* G^T is a directed graph with the same vertices as G, that contains any edge $u{\to}v$ if and only if there is a directed path from u to v in G. Describe an efficient algorithm to compute the transitive closure of G.

   (b) A *transitive reduction* G^{TR} is a graph with the smallest possible number of edges whose transitive closure is G^T. (The same graph may have several transitive reductions.) Describe an efficient algorithm to compute the transitive reduction of G.

3. One of the oldest[1] algorithms for exploring graphs was proposed by Gaston Tarry in 1895. The input to Tarry's algorithm is a directed graph G that contains both directions of every edge; that is, for every edge $u{\to}v$ in G, its reversal $v{\to}u$ is also an edge in G.

```
TARRY(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s)
```

```
RECTARRY(v):
  mark v                               ⟨⟨"visit v"⟩⟩
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red                    } ⟨⟨"traverse v→w"⟩⟩
      RECTARRY(w)                      }
  else if there is a green arc v→w
      color v→w red                    } ⟨⟨"traverse v→w"⟩⟩
      RECTARRY(w)                      }
```

We informally say that Tarry's algorithm "visits" vertex v every time it marks v, and it "traverses" edge $v{\to}w$ when it colors that edge red and recursively calls RecTarry(w).

   (a) Describe how to implement Tarry's algorithm so that it runs in $O(V + E)$ time.

   (b) Prove that no directed edge in G is traversed more than once.

   (c) When the algorithm visits a vertex v for the kth time, exactly how many edges into v are red, and exactly how many edges out of v are red? *[Hint: Consider the starting vertex s separately from the other vertices.]*

   (d) Prove each vertex v is visited at most $\deg(v)$ times, except the starting vertex s, which is visited at most $\deg(s) + 1$ times. This claim immediately implies that Tarry(G) terminates.

   (e) Prove that when Tarry(G) ends, the last visited vertex is the starting vertex s.

   (f) For every vertex v that Tarry(G) visits, prove that all edges into v and out of v are red when Tarry(G) halts. *[Hint: Consider the vertices in the order that they are marked for the first time, starting with s, and prove the claim by induction.]*

   (g) Prove that Tarry(G) visits every vertex of G. This claim and the previous claim imply that Tarry(G) traverses every edge of G exactly once.

4. Consider the following variant of Tarry's graph-traversal algorithm; this variant traverses green edges without recoloring them red and assigns two numerical labels to every vertex:

[1] Even older graph-traversal algorithms were described by Charles Trémaux in 1882, by Christian Wiener in 1873, and (implicitly) by Leonhard Euler in 1736. Wiener's algorithm is equivalent to depth-first search in a connected undirected graph.

```
TARRY2(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s, 1)
```

```
RECTARRY2(v, clock):
  if v is unmarked
      pre(v) ← clock;  clock ← clock + 1
      mark v
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red
      RECTARRY2(w, clock)
  else if there is a green arc v→w
      post(v) ← clock;  clock ← clock + 1
      RECTARRY2(w, clock)
```

Prove or disprove the following claim: When TARRY2(G) halts, the green edges define a spanning tree and the labels $pre(v)$ and $post(v)$ define a preorder and postorder labeling that are all consistent with a single depth-first search of G. In other words, prove or disprove that TARRY2 produces the same output as depth-first search.

5. For any two nodes u and v in a directed acyclic graph G, the ***interval*** $\boldsymbol{G[u,v]}$ is the union of all directed paths in G from u to v. Equivalently, $G[u,v]$ consists of all vertices x such that $x \in Reach(u)$ and $v \in Reach(x)$, together with all the edges in G connecting those vertices.

   Suppose we are given a directed acyclic graph G, in which every edge has a numerical weight, which may be positive, negative, or zero. Describe an efficient algorithm to find the maximum-weight interval in G, where the weight of any interval is the sum of the weights of its vertices. *[Hint: Don't try to be clever.]*

6. Let G be a directed acyclic graph with a unique source s and a unique sink t.

   (a) A *Hamiltonian path* in G is a directed path in G that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

   (b) Suppose the *vertices* of G have weights. Describe an efficient algorithm to find the path from s to t with maximum total weight.

   (c) Suppose we are also given an integer ℓ. Describe an efficient algorithm to find the maximum-weight path from s to t, such that the path contains at most ℓ edges. (Assume there is at least one such path.)

   (d) Suppose the vertices of G have integer labels, where $label(s) = -\infty$ and $label(t) = \infty$. Describe an algorithm to find the path from s to t with the maximum number of edges, such that the vertex labels define an increasing sequence.

   (e) Describe an algorithm to compute the number of distinct paths from s to t in G. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

7. Let G and H be directed acyclic graphs, whose vertices have labels from some fixed alphabet, and let $A[1\,..\,\ell]$ be a string over the same alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices.

   (a) Describe an algorithm that either finds a path in G whose label is A or correctly reports that there is no such path.

(b) Describe an algorithm to find the *number* of paths in G whose label is A. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

(c) Describe an algorithm to find the longest path in G whose label is a subsequence of A.

(d) Describe an algorithm to find the *shortest* path in G whose label is a *supersequence* of A.

(e) Describe an algorithm to find a path in G whose label has minimum edit distance from A.

(f) Describe an algorithm to find the longest string that is both a label of a directed path in G and the label of a directed path in H.

(g) Describe an algorithm to find the longest string that is both a *subsequence* of the label of a directed path in G and a *subsequence* of the label of a directed path in H.

(h) Describe an algorithm to find the shortest string that is both a *supersequence* of the label of a directed path in G and a *supersequence* of the label of a directed path in H.

(i) Describe an algorithm to find the longest path in G whose label is a palindrome.

(j) Describe an algorithm to find the longest palindrome that is a subsequence of the label of a path in G.

(k) Describe an algorithm to find the shortest palindrome that is a supersequence of the label of a path in G.

8. Suppose two players are playing a turn-based game on a directed acyclic graph G with a unique source s. Each vertex v of G is labeled with a real number $\ell(v)$, which could be positive, negative, or zero. The game starts with three tokens at s. In each turn, the current player moves one of the tokens along a directed edge from its current node to another node, and the current player's score is increased by $\ell(u) \cdot \ell(v)$, where u and v are the locations of the two tokens that did *not* move. At most one token is allowed on any node except s at any time. The game ends when the current player is unable to move (for example, when all three tokens lie on sinks); at that point, the player with the higher score is the winner.

   Describe an efficient algorithm to determine who wins this game on a given labeled graph, assuming both players play optimally.

*9. Let $x = x_1 x_2 \dots x_n$ be a given n-character string over some finite alphabet Σ, and let A be a deterministic finite-state machine with m states over the same alphabet.

(a) Describe and analyze an algorithm to compute the length of the longest subsequence of x that is accepted by A. For example, if A accepts the language $(\mathtt{AR})^*$ and $x =$ **A**B**R**AC**A**DAB**R**A, your algorithm should output the number 4, which is the length of the subsequence ARAR.

(b) Describe and analyze an algorithm to compute the length of the shortest supersequence of x that is accepted by A. For example, if A accepts the language $(\mathtt{ABCDR})^*$ and $x =$ ABRACADABRA, your algorithm should output the number 25, which is the length of the supersequence **AB**CD**RA**B**C**DR**A**BC**D**R**AB**CD**RA**BCDR.

10. Not *every* dynamic programming algorithm can be modeled as finding an optimal path through a directed acyclic graph; the most obvious counterexample is the optimal binary search tree problem. But every dynamic programming problem does traverse a dependency graph in reverse topological order, performing some additional computation at every vertex.

    (a) Suppose we are given a directed acyclic graph G where every node stores a numerical search key. Describe and analyze an algorithm to find the largest binary search tree that is a subgraph of G.

    (b) Let G be a directed acyclic graph with the following features:

    - G has a single source s and several sinks $t_1, t_2, \ldots, t_k$.
    - Each edge $v \rightarrow w$ has an associated numerical value $p(v \rightarrow w)$ between 0 and 1.
    - For each non-sink vertex v, we have $\sum_w p(v \rightarrow w) = 1$.

    The values $p(v \rightarrow w)$ define a random walk in G from the source s to some sink t_i; after reaching any non-sink vertex v, the walk follows edge $v \rightarrow w$ with probability $p(v \rightarrow w)$. Describe and analyze an algorithm to compute the probability that this random walk reaches sink t_i, for every index i. (Assume that any arithmetic operation requires $O(1)$ time.)

We must all hang together, gentlemen, or else we shall most assuredly hang separately.

— Benjamin Franklin, at the signing of the Declaration of Independence (July 4, 1776)

It is a very sad thing that nowadays there is so little useless information.

— Oscar Wilde

A ship in port is safe, but that is not what ships are for.

— Rear Admiral Grace Murray Hopper

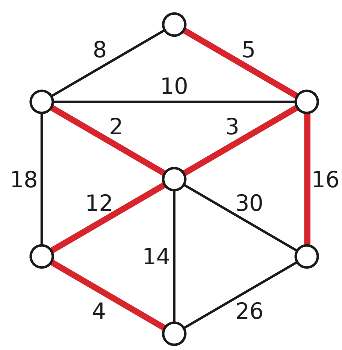
20 Minimum Spanning Trees

20.1 Introduction

Suppose we are given a connected, undirected, *weighted* graph. This is a graph $G = (V, E)$ together with a function $w\colon E \to \mathbb{R}$ that assigns a real *weight* $w(e)$ to each edge e, which may be positive, negative, or zero. Our task is to find the ***minimum spanning tree*** of G, that is, the spanning tree T that minimizes the function

$$w(T) = \sum_{e \in T} w(e).$$

To keep things simple, I'll assume that all the edge weights are distinct: $w(e) \neq w(e')$ for any pair of edges e and e'. Distinct weights guarantee that the minimum spanning tree of the graph is unique. Without this condition, there may be several different minimum spanning trees. For example, if all the edges have weight 1, then *every* spanning tree is a minimum spanning tree with weight $V - 1$.



A weighted graph and its minimum spanning tree.

If we have an algorithm that assumes the edge weights are unique, we can still use it on graphs where multiple edges have the same weight, as long as we have a consistent method for breaking ties. One way to break ties consistently is to use the following algorithm in place of a simple comparison. ShorterEdge takes as input four integers i, j, k, l, and decides which of the two edges (i, j) and (k, l) has "smaller" weight.

```
ShorterEdge(i, j, k, l)
  if w(i, j) < w(k, l)          then return (i, j)
  if w(i, j) > w(k, l)          then return (k, l)
  if min(i, j) < min(k, l)      then return (i, j)
  if min(i, j) > min(k, l)      then return (k, l)
  if max(i, j) < max(k, l)      then return (i, j)
  ⟨⟨if max(i,j) < max(k,l)⟩⟩    return (k, l)
```

20.2 The Only Minimum Spanning Tree Algorithm

There are several different methods for computing minimum spanning trees, but really they are all instances of the following generic algorithm. The situation is similar to the previous lecture, where we saw that depth-first search and breadth-first search were both instances of a single generic traversal algorithm.

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G, which we will call an *intermediate spanning forest*. F is a subgraph of the minimum spanning tree of G, and every component of F is a minimum spanning tree of its vertices. Initially, F consists of n one-node trees. The generic algorithm merges trees together by adding certain edges between them. When the algorithm halts, F consists of a single n-node tree, which must be the minimum spanning tree. Obviously, we have to be careful about *which* edges we add to the evolving forest, since not every edge is in the minimum spanning tree.

The intermediate spanning forest F induces two special types of edges. An edge is ***useless*** if it is not an edge of F, but both its endpoints are in the same component of F. For each component of F, we associate a ***safe*** edge—the minimum-weight edge with exactly one endpoint in that component. Different components might or might not have different safe edges. Some edges are neither safe nor useless—we call these edges *undecided*.

All minimum spanning tree algorithms are based on two simple observations.

Lemma 1. *The minimum spanning tree contains every safe edge.*

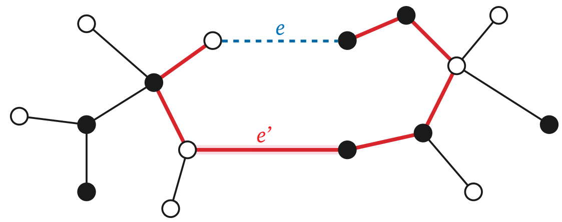
Proof: In fact we prove the following stronger statement: For *any* subset S of the vertices of G, the minimum spanning tree of G contains the minimum-weight edge with exactly one endpoint in S. We prove this claim using a greedy exchange argument.

Let S be an arbitrary subset of vertices of G; let e be the lightest edge with exactly one endpoint in S; and let T be an arbitrary spanning tree that does *not* contain e. Because T is connected, it contains a path from one endpoint of e to the other. Because this path starts at a vertex of S and ends at a vertex not in S, it must contain at least one edge with exactly one endpoint in S; let e' be *any* such edge. Because T is acyclic, removing e' from T yields a spanning *forest* with exactly two components, one containing each endpoint of e. Thus, adding e to this forest gives us a new spanning tree $T' = T - e' + e$. The definition of e implies $w(e') > w(e)$, which implies that T' has smaller total weight than T. We conclude that T is not the minimum spanning tree, which completes the proof. □

Lemma 2. *The minimum spanning tree contains no useless edge.*

Proof: Adding any useless edge to F would introduce a cycle. □

Our generic minimum spanning tree algorithm repeatedly adds one or more safe edges to the evolving forest F. Whenever we add new edges to F, some undecided edges become safe, and



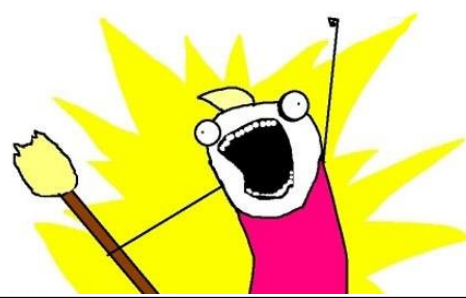
Proving that every safe edge is in the minimum spanning tree. Black vertices are in the subset S.

others become useless. To specify a particular algorithm, we must decide which safe edges to add, and we must describe how to identify new safe and new useless edges, at each iteration of our generic template.

20.3 Borvka's Algorithm

The oldest and arguably simplest minimum spanning tree algorithm was discovered by Borvka in 1926, long before computers even existed, and practically before the invention of graph theory![1] The algorithm was rediscovered by Choquet in 1938; again by Florek, Łukaziewicz, Perkal, Stienhaus, and Zubrzycki in 1951; and again by Sollin some time in the early 1960s. Because Sollin was the only Western computer scientist in this list—Choquet was a civil engineer; Florek and his co-authors were anthropologists—this is often called "Sollin's algorithm", especially in the parallel computing literature.

The Borvka/Choquet/Florek/Łukaziewicz/Perkal/Stienhaus/Zubrzycki/Sollin algorithm can be summarized in one line:



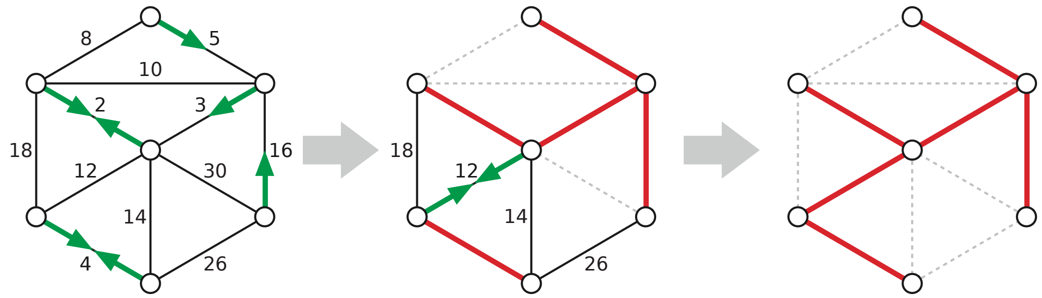
BORVKA: Add ***ALL*** the safe edges[2] and recurse.

We can find all the safe edge in the graph in $O(E)$ time as follows. First, we count the components of F using whatever-first search, using the standard wrapper function. As we count, we label every vertex with its component number; that is, every vertex in the first traversed component gets label 1, every vertex in the second component gets label 2, and so on.

If F has only one component, we're done. Otherwise, we compute an array $S[1..V]$ of edges, where $S[i]$ is the minimum-weight edge with one endpoint in the ith component (or a sentinel value NULL if there are less than i components). To compute this array, we consider each edge uv in the input graph G. If the endpoints u and v have the same label, then uv is useless. Otherwise, we compare the weight of uv to the weights of $S[label(u)]$ and $S[label(v)]$ and update the array entries if necessary.

[1] Leonard Euler published the first graph theory result, his famous theorem about the bridges of Königsburg, in 1736. However, the first textbook on graph theory, written by Dénes König, was not published until 1936.

[2] See also: Allie Brosh, "This is Why I'll Never be an Adult", *Hyperbole and a Half*, June 17, 2010. Actually, just go see everything in *Hyperbole and a Half*. And then go buy the book. And an extra copy for your cat.



Borůvka's algorithm run on the example graph. Thick edges are in F.
Arrows point along each component's safe edge. Dashed (gray) edges are useless.

```
BORVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
      ADDALLSAFEEDGES(E, F, count)
      count ← COUNTANDLABEL(F)
  return F
```

```
ADDALLSAFEEDGES(E, F, count):
  for i ← 1 to count
      S[i] ← NULL        ⟨⟨sentinel: w(NULL) := ∞⟩⟩
  for each edge uv ∈ E
      if label(u) ≠ label(v)
          if w(uv) < w(S[label(u)])
              S[label(u)] ← uv
          if w(uv) < w(S[label(v)])
              S[label(v)] ← uv
  for i ← 1 to count
      if S[i] ≠ NULL
          add S[i] to F
```

Each call to TRAVERSEALL requires $O(V)$ time, because the forest F has at most $V-1$ edges. Assuming the graph is represented by an adjacency list, the rest of each iteration of the main while loop requires $O(E)$ time, because we spend constant time on each edge. Because the graph is connected, we have $V \le E+1$, so each iteration of the while loop takes $O(E)$ time.

Each iteration reduces the number of components of F by at least a factor of two—the worst case occurs when the components coalesce in pairs. Since F initially has V components, the while loop iterates at most $O(\log V)$ times. Thus, the overall running time of Borvka's algorithm is $\boldsymbol{O(E \log V)}$.

Despite its relatively obscure origin, early algorithms researchers were aware of Borvka's algorithm, but dismissed it as being "too complicated"! As a result, despite its simplicity and efficiency, Borvka's algorithm is rarely mentioned in algorithms and data structures textbooks. On the other hand, Borvka's algorithm has several distinct advantages over other classical MST algorithms.

- Borvka's algorithm often runs faster than the $O(E \log V)$ worst-case running time. In arbitrary graphs, the number of components in F can drop by significantly more than a factor of 2 in a single iteration, reducing the number of iterations below the worst-case $\lceil \log_2 V \rceil$. A slight reformulation of Borvka's algorithm (actually closer to Borvka's original presentation) actually runs in $O(E)$ time for a broad class of interesting graphs, including graphs that can be drawn in the plane without edge crossings. In contrast, the time analysis for the other two algorithms applies to *all* graphs.

- Borvka's algorithm allows for significant parallelism; in each iteration, each component of F can be handled in a separate independent thread. This implicit parallelism allows for even faster performance on multicore or distributed systems. In contrast, the other two classical MST algorithms are intrinsically serial.

- There are several more recent minimum-spanning-tree algorithms that are faster even in the worst case than the classical algorithms described here. *All* of these faster algorithms are generalizations of Borvka's algorithm.

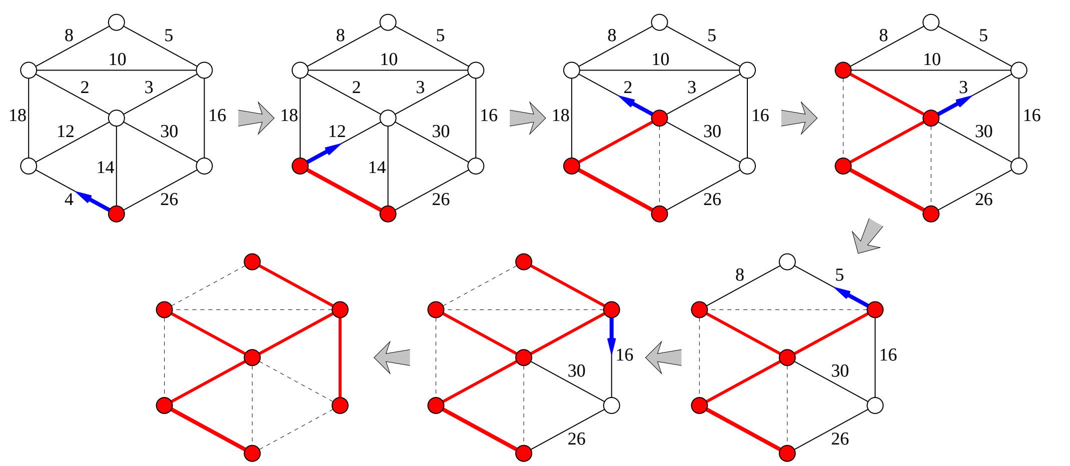
In short, if you ever need to implement a minimum-spanning-tree algorithm, use Borvka. On the other hand, if you want to *prove things about* minimum spanning trees effectively, you really need to know the next two algorithms as well.

20.4 Jarník's ("Prim's") Algorithm

The next oldest minimum spanning tree algorithm was first described by the Czech mathematician Vojtch Jarník in a 1929 letter to Borvka; Jarník published his discovery the following year. The algorithm was independently rediscovered by Kruskal in 1956, by Prim in 1957, by Loberman and Weinberger in 1957, and finally by Dijkstra in 1958. Prim, Loberman, Weinberger, and Dijkstra all (eventually) knew of and even cited Kruskal's paper, but since Kruskal also described two other minimum-spanning-tree algorithms in the same paper, *this* algorithm is usually called "Prim's algorithm", or sometimes "the Prim/Dijkstra algorithm", even though by 1958 Dijkstra already had another algorithm (inappropriately) named after him.

In Jarník's algorithm, the forest F contains only one nontrivial component T; all the other components are isolated vertices. Initially, T consists of an arbitrary vertex of the graph. The algorithm repeats the following step until T spans the whole graph:

$$\boxed{\textsc{Jarník}\text{: Repeatedly add } T\text{'s safe edge to } T.}$$



Jarník's algorithm run on the example graph, starting with the bottom vertex.
At each stage, thick edges are in T, an arrow points along T's safe edge, and dashed edges are useless.

To implement Jarník's algorithm, we keep all the edges adjacent to T in a priority queue. When we pull the minimum-weight edge out of the priority queue, we first check whether both of its endpoints are in T. If not, we add the edge to T and then add the new neighboring edges to the priority queue. In other words, Jarník's algorithm is another instance of the generic graph traversal algorithm we saw last time, using a priority queue as the "bag"! If we implement the algorithm this way, the algorithm runs in $O(E \log E) = \boldsymbol{O(E \log V)}$ time.

*20.5 Improving Jarník's Algorithm

We can improve Jarník's algorithm using a more advanced priority queue data structure called a *Fibonacci heap*, first described by Michael Fredman and Robert Tarjan in 1984. Fibonacci heaps support the standard priority queue operations INSERT, EXTRACTMIN, and DECREASEKEY. However, unlike standard binary heaps, which require $O(\log n)$ time for every operation, Fibonacci heaps support INSERT and DECREASEKEY in constant *amortized* time. The amortized cost of EXTRACTMIN is still $O(\log n)$.

To apply this faster data structure, we keep *vertices* in the priority queue instead of edge, where the key for each vertex v is either the minimum-weight edge between v and the evolving tree T, or ∞ if there is no such edge. We can INSERT all the vertices into the priority queue at the beginning of the algorithm; then, whenever we add a new edge to T, we may need to decrease the keys of some neighboring vertices.

To make the description easier, we break the algorithm into two parts. JARNÍK INIT initializes the priority queue; JARNÍK LOOP is the main algorithm. The input consists of the vertices and edges of the graph, plus the start vertex s. For each vertex v, we maintain both its key $key(v)$ and the incident edge $edge(v)$ such that $w(edge(v)) = key(v)$.

```
JARNÍK(V, E, s):
    JARNÍKINIT(V, E, s)
    JARNÍKLOOP(V, E, s)
```

```
JARNÍKINIT(V, E, s):
    for each vertex v ∈ V \ {s}
        if (v, s) ∈ E
            edge(v) ← (v, s)
            key(v) ← w(v, s)
        else
            edge(v) ← NULL
            key(v) ← ∞
        INSERT(v)
```

```
JARNÍKLOOP(V, E, s):
    T ← ({s}, ∅)
    for i ← 1 to |V| − 1
        v ← EXTRACTMIN
        add v and edge(v) to T
        for each neighbor u of v
            if u ∉ T and key(u) > w(uv)
                edge(u) ← uv
                DECREASEKEY(u, w(uv))
```

The operations INSERT and EXTRACTMIN are each called $O(V)$ times once for each vertex except s, and DECREASEKEY is called $O(E)$ times, at most twice for each edge. Thus, if we use a Fibonacci heap, the improved algorithm runs in ***$O(E + V \log V)$ time***, which is faster than Borvka's algorithm unless $E = O(V)$.

In practice, however, this improvement is rarely faster than the naive implementation using a binary heap, unless the graph is extremely large and dense. The Fibonacci heap algorithms are quite complex, and the hidden constants in both the running time and space are significant—not outrageous, but certainly bigger than the hidden constant 1 in the $O(\log n)$ time bound for binary heap operations.

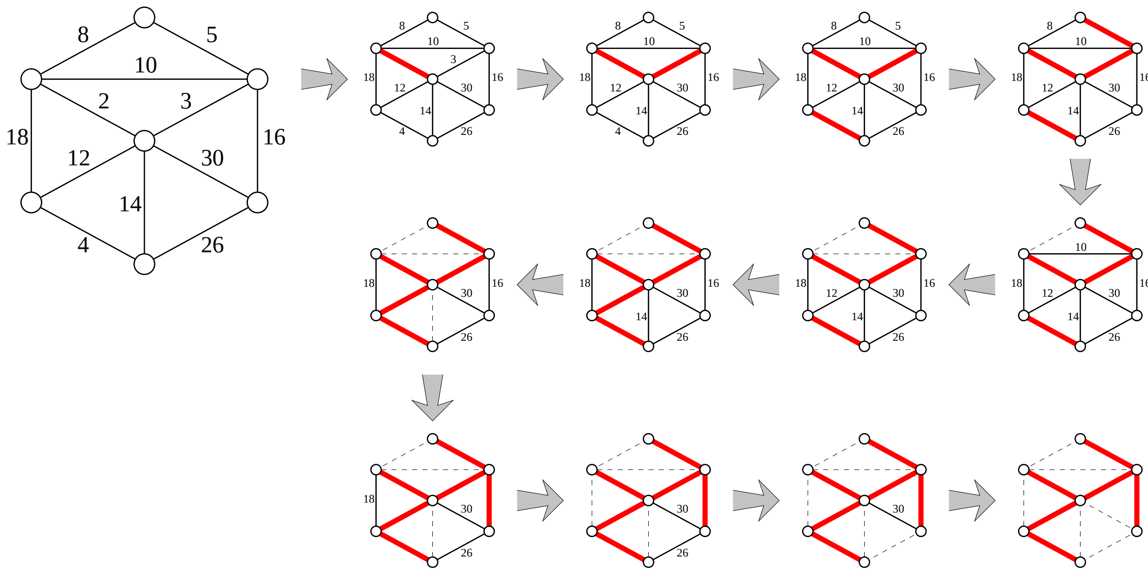
20.6 Kruskal's Algorithm

The last minimum spanning tree algorithm I'll discuss was first described by Kruskal in 1956, in the same paper where he rediscovered Jarnik's algorithm. Kruskal was motivated by "a typewritten translation (of obscure origin)" of Borvka's original paper, claiming that Borvka's algorithm was "unnecessarily elaborate".[3] This algorithm was also rediscovered in 1957 by Loberman and

[3]To be fair, Borvka's original paper *was* unnecessarily elaborate, but in his followup paper, also published in 1927, simplified his algorithm essentially to its current modern form. Kruskal was apparently unaware of Borvka's second paper. Stupid Iron Curtain.

Weinberger, but somehow avoided being renamed after them.

KRUSKAL: Scan all edges in increasing weight order; if an edge is safe, add it to F.



Kruskal's algorithm run on the example graph. Thick edges are in F. Dashed edges are useless.

Since we examine the edges in order from lightest to heaviest, any edge we examine is safe if and only if its endpoints are in different components of the forest F. To prove this, suppose the edge e joins two components A and B but is not safe. Then there would be a lighter edge e' with exactly one endpoint in A. But this is impossible, because (inductively) any previously examined edge has both endpoints in the same component of F.

Just as in Borvka's algorithm, each component of F has a "leader" node. An edge joins two components of F if and only if the two endpoints have different leaders. But unlike Borvka's algorithm, we do not recompute leaders from scratch every time we add an edge. Instead, when two components are joined, the two leaders duke it out in a nationally-televised no-holds-barred steel-cage grudge match.[4] One of the two emerges victorious as the leader of the new larger component. More formally, we will use our earlier algorithms for the UNION-FIND problem, where the vertices are the elements and the components of F are the sets. Here's a more formal description of the algorithm:

```
KRUSKAL(V, E):
  sort E by increasing weight
  F ← (V, ∅)
  for each vertex v ∈ V
      MAKESET(v)
  for i ← 1 to |E|
      uv ← ith lightest edge in E
      if FIND(u) ≠ FIND(v)
          UNION(u, v)
          add uv to F
  return F
```

[4] Live at the Assembly Hall! Only $49.95 on Pay-Per-View![5]

[5] Is Pay-Per-View still a thing?

In our case, the sets are components of F, and $n = V$. Kruskal's algorithm performs $O(E)$ FIND operations, two for each edge in the graph, and $O(V)$ UNION operations, one for each edge in the minimum spanning tree. Using union-by-rank and path compression allows us to perform each UNION or FIND in $O(\alpha(E, V))$ time, where α is the not-quite-constant inverse-Ackerman function. So ignoring the cost of sorting the edges, the running time of this algorithm is $O(E\,\alpha(E, V))$.

We need $O(E \log E) = O(E \log V)$ additional time just to sort the edges. Since this is bigger than the time for the UNION-FIND data structure, the overall running time of Kruskal's algorithm is $\boldsymbol{O(E \log V)}$, exactly the same as Borvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.

Exercises

1. Most classical minimum-spanning-tree algorithms use the notions of "safe" and "useless" edges described in the lecture notes, but there is an alternate formulation. Let G be a weighted undirected graph, where the edge weights are distinct. We say that an edge e is ***dangerous*** if it is the longest edge in some cycle in G, and ***useful*** if it does not lie in any cycle in G.

   (a) Prove that the minimum spanning tree of G contains every useful edge.

   (b) Prove that the minimum spanning tree of G does not contain any dangerous edge.

   (c) Describe and analyze an efficient implementation of the "anti-Kruskal" MST algorithm: Examine the edges of G in *decreasing* order; if an edge is dangerous, remove it from G. *[Hint: It won't be as fast as Kruskal's algorithm.]*

2. Let $G = (V, E)$ be an arbitrary connected graph with weighted edges.

   (a) Prove that for any partition of the vertices V into two disjoint subsets, the minimum spanning tree of G includes the minimum-weight edge with one endpoint in each subset.

   (b) Prove that for any cycle in G, the minimum spanning tree of G *excludes* the maximum-weight edge in that cycle.

   (c) Prove or disprove: The minimum spanning tree of G includes the minimum-weight edge in *every* cycle in G.

3. Throughout this lecture note, we assumed that no two edges in the input graph have equal weights, which implies that the minimum spanning tree is unique. In fact, a weaker condition on the edge weights implies MST uniqueness.

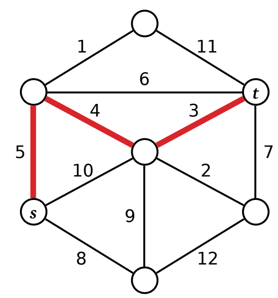
   (a) Describe an edge-weighted graph that has a unique minimum spanning tree, even though two edges have equal weights.

   (b) Prove that an edge-weighted graph G has a *unique* minimum spanning tree if and only if the following conditions hold:

      - For any partition of the vertices of G into two subsets, the minimum-weight edge with one endpoint in each subset is unique.
      - The maximum-weight edge in any cycle of G is unique.

(c) Describe and analyze an algorithm to determine whether or not a graph has a unique minimum spanning tree.

4. Consider a path between two vertices s and t in an undirected weighted graph G. The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t. (If there are no paths from s to t, the bottleneck distance between s and t is ∞.)



The bottleneck distance between s and t is 5.

Describe an algorithm to compute the bottleneck distance between *every* pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.

5. (a) Describe and analyze an algorithm to compute the *maximum*-weight spanning tree of a given edge-weighted graph.

   (b) A *feedback edge set* of an undirected graph G is a subset F of the edges such that every cycle in G contains at least one edge in F. In other words, removing every edge in F makes the graph G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of of a given edge-weighted graph.

6. Suppose we are given both an undirected graph G with weighted edges and a minimum spanning tree T of G.

   (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.

   (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

   In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. *[Hint: Consider the cases $e \in T$ and $e \notin T$ separately.]*

7. (a) Describe and analyze and algorithm to find the *second smallest spanning tree* of a given graph G, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree.

   *(b) Describe and analyze an efficient algorithm to compute, given a weighted undirected graph G and an integer k, the k spanning trees of G with smallest weight.

8. We say that a graph $G = (V, E)$ is *dense* if $E = \Theta(V^2)$. Describe a modification of Jarník's minimum-spanning tree algorithm that runs in $O(V^2)$ time (independent of E) when the input graph is dense, using only simple data structures (and in particular, *without* using a Fibonacci heap).

9. (a) Prove that the minimum spanning tree of a graph is also a spanning tree whose maximum-weight edge is minimal.

   ⋆(b) Describe an algorithm to compute a spanning tree whose maximum-weight edge is minimal, in $O(V + E)$ time. *[Hint: Start by computing the median of the edge weights.]*

10. Consider the following variant of Borvka's algorithm. Instead of counting and labeling components of F to find safe edges, we use a standard disjoint set data structure. Each component of F is represented by an up-tree; each vertex v stores a pointer *parent*(v) to its parent in the up-tree containing v. Each leader vertex $\bar{v}$ also maintains an edge *safe*$(\bar{v})$, which is (eventually) the lightest edge with one endpoint in $\bar{v}$'s component of F.

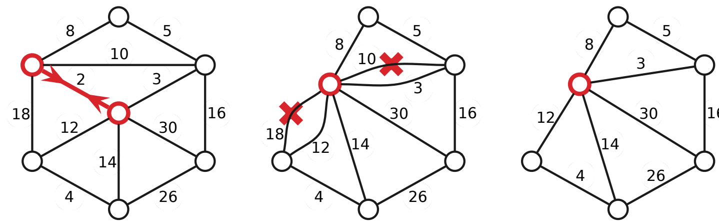
```
BORVKA(V, E):
  F = ∅
  for each vertex v ∈ V
      parent(v) ← v
  while FINDSAFEEDGES(V, E)
      ADDSAFEEDGES(V, E, F)
  return F
```

```
FINDSAFEEDGES(V, E):
  for each vertex v ∈ V
      safe(v) ← NULL
  found ← FALSE
  for each edge uv ∈ E
      ū ← FIND(u);  v̄ ← FIND(v)
      if ū ≠ v̄
          if  w(uv) < w(safe(ū))
              safe(ū) ← uv
          if  w(uv) < w(safe(v̄))
              safe(v̄) ← uv
          found ← TRUE
  return done
```

```
ADDSAFEEDGES(V, E, F):
  for each vertex v ∈ V
      if safe(v) ≠ NULL
          xy ← safe(v)
          if FIND(x) ≠ FIND(y)
              UNION(x, y)
              add xy to F
```

Prove that if FIND uses path compression, then each call to FINDSAFEEDGES and ADDSAFEEDGES requires only $O(V + E)$ time. *[Hint: It doesn't matter how UNION is implemented! What is the depth of the up-trees when FINDSAFEEDGES ends?]*

11. Minimum-spanning tree algorithms are often formulated using an operation called *edge contraction*. To contract the edge uv, we insert a new node, redirect any edge incident to u or v (except uv) to this new node, and then delete u and v. After contraction, there may be multiple parallel edges between the new node and other nodes in the graph; we remove all but the lightest edge between any two nodes.



Contracting an edge and removing redundant parallel edges.

The three classical minimum-spanning tree algorithms can be expressed cleanly in terms of contraction as follows. All three algorithms start by making a clean copy G' of the input graph G and then repeatedly contract safe edges in G; the minimum spanning tree consists of the contracted edges.

- BORVKA: Mark the lightest edge leaving each vertex, contract all marked edges, and recurse.
- JARNÍK: Repeatedly contract the lightest edge incident to some fixed root vertex.
- KRUSKAL: Repeatedly contract the lightest edge in the graph.

(a) Describe an algorithm to execute a single pass of Borvka's contraction algorithm in $O(V + E)$ time. The input graph is represented in an adjacency list.

(b) Consider an algorithm that first performs k passes of Borvka's contraction algorithm, and then runs Jarník's algorithm (*with* a Fibonacci heap) on the resulting contracted graph.

  i. What is the running time of this hybrid algorithm, as a function of V, E, and k?

  ii. For which value of k is this running time minimized? What is the resulting running time?

(c) Call a family of graphs *nice* if it has the following properties:

  - A nice graph with n vertices has only $O(n)$ edges.
  - Contracting an edge of a nice graph yields another nice graph.

  For example, graphs that can be drawn in the plane without crossing edges are nice; Euler's formula implies that any planar graph with n vertices has at most $3n-6$ edges.

  Prove that Borüvka's contraction algorithm computes the minimum spanning tree of any nice n-vertex graph in $O(n)$ time.

Well, ya turn left by the fire station in the village and take the old post road by the reservoir and. . . no, that won't do.

Best to continue straight on by the tar road until you reach the schoolhouse and then turn left on the road to Bennett's Lake until. . . no, that won't work either.

East Millinocket, ya say? Come to think of it, you can't get there from here.

— Robert Bryan and Marshall Dodge,
Bert and I and Other Stories from Down East (1961)

Hey farmer! Where does this road go?
Been livin' here all my life, it ain't gone nowhere yet.
Hey farmer! How do you get to Little Rock?
Listen stranger, you can't get there from here.
Hey farmer! You don't know very much do you?
No, but I ain't lost.

— Michelle Shocked, "Arkansas Traveler" (1992)

21 Shortest Paths

21.1 Introduction

Suppose we are given a weighted *directed* graph $G = (V, E, w)$ with two special vertices, and we want to find the shortest path from a *source* vertex s to a *target* vertex t. That is, we want to find the directed path p starting at s and ending at t that minimizes the function

$$w(p) := \sum_{u\to v\in p} w(u\to v).$$

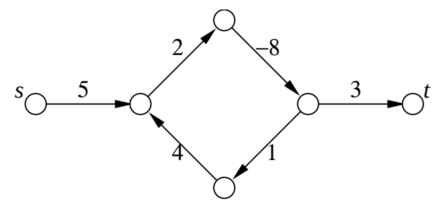
For example, if I want to answer the question "What's the fastest way to drive from my old apartment in Champaign, Illinois to my wife's old apartment in Columbus, Ohio?", I might use a graph whose vertices are cities, edges are roads, weights are driving times, s is Champaign, and t is Columbus.[1] The graph is directed, because driving times along the same road might be different in different directions. (At one time, there was a speed trap on I-70 just east of the Indiana/Ohio border, but only for eastbound traffic.)

Perhaps counter to intuition, we will allow the weights on the edges to be negative. Negative edges make our lives complicated, because the presence of a negative *cycle* might imply that there is no shortest path. In general, a shortest path from s to t exists if and only if there is *at least one* path from s to t, but there is no path from s to t that touches a negative cycle. If any negative cycle is reachable from s and can reach t, we can always find a shorter path by going around the cycle one more time.

Almost every algorithm known for solving this problem actually solves (large portions of) the following more general ***single source shortest path*** or ***SSSP*** problem: Find the shortest path from the source vertex s to *every* other vertex in the graph. This problem is usually solved by finding a ***shortest path tree*** rooted at s that contains all the desired shortest paths.

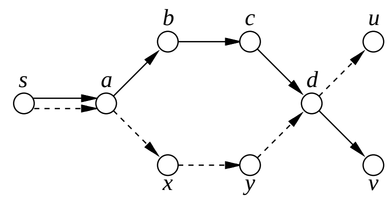
It's not hard to see that if shortest paths are unique, then they form a tree, because any subpath of a shortest path is itself a shortest path. If there are multiple shortest paths to some

[1] West on Church, north on Prospect, east on I-74, south on I-465, east on Airport Expressway, north on I-65, east on I-70, north on Grandview, east on 5th, north on Olentangy River, east on Dodridge, north on High, west on Kelso, south on Neil. Depending on traffic. We both live in Urbana now.



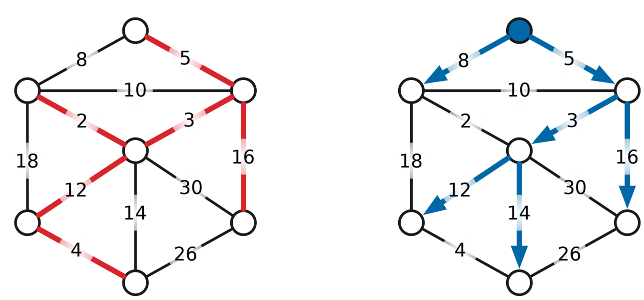
There is no shortest path from s to t.

vertices, we can always choose one shortest path to each vertex so that the union of the paths is a tree. If there are shortest paths to two vertices u and v that diverge, then meet, then diverge again, we can modify one of the paths without changing its length so that the two paths only diverge once.



If $s \to a \to b \to c \to d \to v$ and $s \to a \to x \to y \to d \to u$ are shortest paths,
then $s \to a \to b \to c \to d \to u$ is also a shortest path.

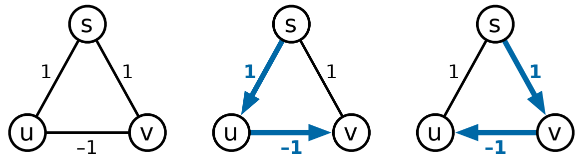
Although they are both optimal spanning trees, shortest-path trees and minimum spanning trees are very different creatures. Shortest-path trees are rooted and directed; minimum spanning trees are unrooted and undirected. Shortest-path trees are most naturally defined for directed graphs; only undirected graphs have minimum spanning trees. If edge weights are distinct, there is only one minimum spanning tree, but every source vertex induces a different shortest-path tree; moreover, it is possible for *every* shortest path tree to use a different set of edges from the minimum spanning tree.



A minimum spanning tree and a shortest path tree (rooted at the top vertex) of the same undirected graph.

21.2 Warning!

Throughout this lecture, we will explicitly consider *only* directed graphs. All of the algorithms described in this lecture also work for undirected graphs with some minor modifications, *but only if negative edges are prohibited*. Dealing with negative edges in undirected graphs is considerably more subtle. We cannot simply replace every undirected edge with a pair of directed edges, because this would transform any negative edge into a short negative cycle. Subpaths of an undirected shortest path that contains a negative edge are *not* necessarily shortest paths; consequently, the set of all undirected shortest paths from a single source vertex may not define a tree, even if shortest paths are unique.



An undirected graph where shortest paths from s are unique but do not define a tree.

A complete treatment of undirected graphs with negative edges is beyond the scope of this lecture (if not the entire course). I will only mention that a *single* shortest path in an undirected graph with negative edges can be computed in $O(VE+V^2 \log V)$ time, by a reduction to maximum weighted matching.

21.3 The Only SSSP Algorithm

Just like graph traversal and minimum spanning trees, there are several different SSSP algorithms, but they are all special cases of the a single generic algorithm, first proposed by Lester Ford in 1956, and independently by George Dantzig in 1957.[2] Each vertex v in the graph stores two values, which (inductively) describe a *tentative* shortest path from s to v.

- $dist(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.
- $pred(v)$ is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path, or NULL if there is no such vertex.

In fact, the predecessor pointers automatically define a tentative shortest path *tree*; they play exactly the same role as the parent pointers in our generic graph traversal algorithm. At the beginning of the algorithm, we already know that $dist(s) = 0$ and $pred(s) = \text{NULL}$. For every vertex $v \neq s$, we initially set $dist(v) = \infty$ and $pred(v) = \text{NULL}$ to indicate that we do not know of *any* path from s to v.

During the execution of the algorithm, we call an edge $u \to v$ ***tense*** if $dist(u) + w(u \to v) < dist(v)$. If $u \to v$ is tense, the tentative shortest path $s \rightsquigarrow v$ is clearly incorrect, because the path $s \rightsquigarrow u \to v$ is shorter. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

```
RELAX(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

When there are no tense edges, the algorithm halts, and we have our desired shortest path tree.

The correctness of Ford's generic relaxation algorithm follows from the following series of claims:

1. For every vertex v, the distance $dist(v)$ is either ∞ or the length of some walk from s to v. This claim can be proved by induction on the number of relaxations.
2. If the graph has no negative cycles, then $dist(v)$ is either ∞ or the length of some *simple path* from s to v. Specifically, if $dist(v)$ is the length of a walk from s to v that contains a directed cycle, that cycle must have negative weight. This claim implies that if G has no negative cycles, the relaxation algorithm eventually halts, because there are only a finite number of simple paths in G.

[2] Specifically, Dantzig showed that the shortest path problem can be phrased as a linear programming problem, and then described an interpretation of his simplex method in terms of the original graph. His description is equivalent to Ford's relaxation strategy.

3. If no edge in G is tense, then for every vertex v, the distance $dist(v)$ is the length of the predecessor path $s \to \cdots pred(pred(v)) \to pred(v) \to v$. Specifically, if v violates this condition but its predecessor $pred(v)$ does not, the edge $pred(v) \to v$ is tense.

4. If no edge in G is tense, then for every vertex v, the path $s \to \cdots pred(pred(v)) \to pred(v) \to v$ is a shortest path from s to v. Specifically, if v violates this condition but its predecessor u *in some shortest path* does not, the edge $u \to v$ is tense. This claim also implies that if the G has a negative cycle, then some edge is *always* tense, so the generic algorithm never halts.

So far I haven't said anything about how we detect which edges can be relaxed, or in what order we relax them. To make this easier, we refine the relaxation algorithm slightly, into something closely resembling the generic graph traversal algorithm. We maintain a "bag" of vertices, initially containing just the source vertex s. Whenever we take a vertex u from the bag, we scan all of its outgoing edges, looking for something to relax. Finally, whenever we successfully relax an edge $u \to v$, we put v into the bag. Unlike our generic graph traversal algorithm, we do not mark vertices when we visit them; the same vertex could be visited many times, and the same edge could be relaxed many times.

```
INITSSSP(s):
  dist(s) ← 0
  pred(s) ← NULL
  for all vertices v ≠ s
      dist(v) ← ∞
      pred(v) ← NULL
```

```
GENERICSSSP(s):
  INITSSSP(s)
  put s in the bag
  while the bag is not empty
      take u from the bag
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
              put v in the bag
```

Just as with graph traversal, different "bag" data structures for the give us different algorithms. There are three obvious choices to try: a stack, a queue, and a priority queue. Unfortunately, if we use a stack, the resulting algorithm performs $\Theta(2^V)$ relaxation steps in the worst case! (Proving this is a good homework problem.) The other two possibilities are much more efficient.

21.4 Dijkstra's Algorithm

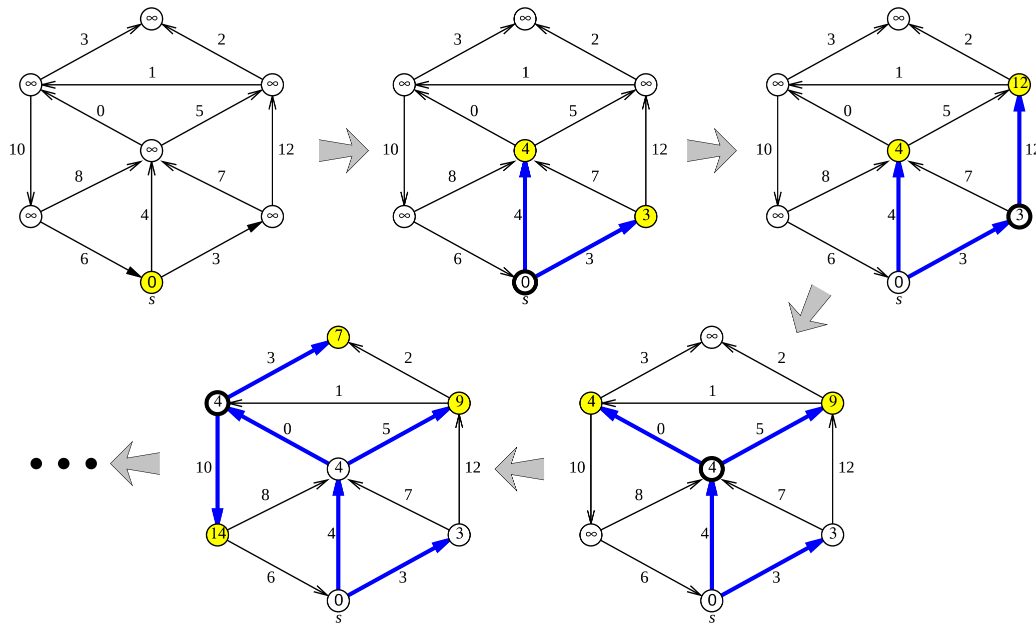
If we implement the bag using a priority queue, where the key of a vertex v is its tentative distance $dist(v)$, we obtain an algorithm first "published" in 1957 by a team of researchers at the Case Institute of Technology, in an annual project report for the Combat Development Department of the US Army Electronic Proving Ground. The same algorithm was later independently rediscovered and actually publicly published by Edsger Dijkstra in 1959. A nearly identical algorithm was also described by George Dantzig in 1958.

Dijkstra's algorithm, as it is universally known[3], is particularly well-behaved if the graph has no negative-weight edges. In this case, it's not hard to show (by induction, of course) that the vertices are scanned in increasing order of their shortest-path distance from s. It follows that each vertex is scanned at most once, and thus that each edge is relaxed at most once. Since the key of each vertex in the heap is its tentative distance from s, the algorithm performs a DECREASEKEY operation every time an edge is relaxed. Thus, the algorithm performs at most E DECREASEKEYS.

[3] I will follow this common convention, despite the historical inaccuracy, partly because I don't think anybody wants to read about the "Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-Seitz algorithm", and partly because papers that aren't actually *publically* published don't count.

Similarly, there are at most V INSERT and EXTRACTMIN operations. Thus, if we store the vertices in a Fibonacci heap, the total running time of Dijkstra's algorithm is $\boldsymbol{O(E + V \log V)}$; if we use a regular binary heap, the running time is $\boldsymbol{O(E \log V)}$.

This analysis assumes that no edge has negative weight. Dijkstra's algorithm (in the form I'm presenting here[4]) is still *correct* if there are negative edges, but the worst-case running time could be exponential. (Proving this unfortunate fact is a good homework problem.) On the other hand, in practice, Dijkstra's algorithm is usually quite fast even for graphs with negative edges.



Four phases of Dijkstra's algorithm run on a graph with no negative edges.
At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.

21.5 The A^* Heuristic

A slight generalization of Dijkstra's algorithm, commonly known as the A^* algorithm, is frequently used to find a shortest path from a single source node s to a single target node t. This heuristic was first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael. A^* uses a black-box function GUESSDISTANCE(v, t) that returns an estimate of the distance from v to t. The only difference between Dijkstra and A^* is that the key of a vertex v is $dist(v) +$ GUESSDISTANCE(v, t).

The function GUESSDISTANCE is called *admissible* if GUESSDISTANCE(v, t) never overestimates the actual shortest path distance from v to t. If GUESSDISTANCE is admissible and the actual edge weights are all non-negative, the A^* algorithm computes the actual shortest path from s to t at least as quickly as Dijkstra's algorithm. In practice, the closer GUESSDISTANCE(v, t) is to the real distance from v to t, the faster the algorithm. However, in the worst case, the running time is still $O(E + V \log V)$.

The heuristic is especially useful in situations where the actual graph is not known. For example, A^* can be used to find optimal solutions to many puzzles (15-puzzle, Freecell, Shanghai,

[4]Most algorithms textbooks, Wikipedia, and even Dijkstra's original paper present a version of Dijkstra's algorithm that gives incorrect results for graphs with negative edges, because it *never* visits the same vertex more than once. I've taken the liberty of correcting Dijkstra's mistake. Even Dijkstra would agree that a correct algorithm that is sometimes slow (and in practice, *rarely* slow) is better than a fast algorithm that doesn't always work.

Sokoban, Atomix, Rush Hour, Rubik's Cube, Racetrack, ...) and other path planning problems where the starting and goal configurations are given, but the graph of all possible configurations and their connections is not given explicitly.

21.6 Shimbel's Algorithm

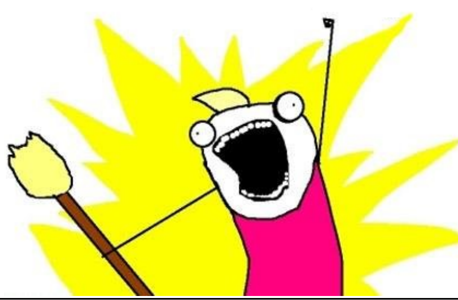
If we replace the heap in Dijkstra's algorithm with a FIFO queue, we obtain an algorithm first sketched by Shimbel in 1954, described in more detail by Moore in 1957, then independently rediscovered by Woodbury and Dantzig in 1957 and again by Bellman in 1958. Because Bellman explicitly used Ford's formulation of relaxing edges, this algorithm is almost universally called "Bellman-Ford", although some early sources refer to "Bellman-Shimbel". Shimbel's algorithm is efficient even if there are negative edges, and it can be used to quickly detect the presence of negative cycles. If there are no negative edges, however, Dijkstra's algorithm is faster. (In fact, in practice, Dijkstra's algorithm is often faster even for graphs with negative edges.)

The easiest way to analyze the algorithm is to break the execution into *phases*, by introducing an imaginary *token*. Before we even begin, we insert the token into the queue. The current phase ends when we take the token out of the queue; we begin the next phase by reinserting the token into the queue. The 0th phase consists entirely of scanning the source vertex s. The algorithm ends when the queue contains *only* the token. A simple inductive argument (hint, hint) implies the following invariant for every integer i and vertex v:

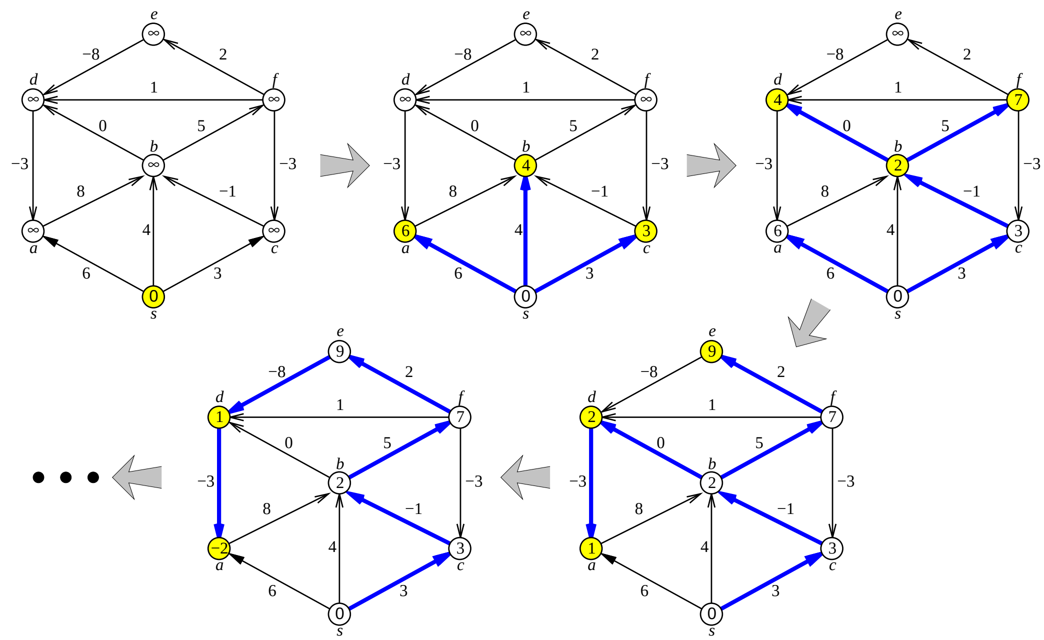
> **After i phases of the algorithm, *dist*(v) is *at most* the length of the shortest walk from s to v consisting of *at most* i edges.**

Since a shortest path can only pass through each vertex once, either the algorithm halts before the Vth phase, or the graph contains a negative cycle. In each phase, we scan each vertex at most once, so we relax each edge at most once, so the running time of a single phase is $O(E)$. Thus, the overall running time of Shimbel's algorithm is $\boldsymbol{O(VE)}$.

Once we understand how the phases of Shimbel's algorithm behave, we can simplify the algorithm considerably by producing the same behavior on purpose. Instead of performing a partial breadth-first search of the graph in each phase, we can simply scan through the adjacency list directly, relaxing every tense edge we find in the graph.



SHIMBEL: Relax ***ALL*** the tense edges and recurse.



Four phases of Shimbel's algorithm run on a directed graph with negative edges.
Nodes are taken from the queue in the order $s \diamond a\ b\ c \diamond d\ f\ b \diamond a\ e\ d \diamond d\ a \diamond \diamond$, where $\diamond$ is the end-of-phase token.
Shaded vertices are in the queue at the end of each phase. The bold edges describe the evolving shortest path tree.

```
ShimbelSSSP(s)
  InitSSSP(s)
  repeat V times:
      for every edge u→v
          if u→v is tense
              Relax(u→v)
  for every edge u→v
      if u→v is tense
          return "Negative cycle!"
```

This is how most textbooks present "Bellman-Ford".[5] The $O(VE)$ running time of this formulation of the algorithm should be obvious, but it may be less clear that the algorithm is still correct. In fact, correctness follows from exactly the same invariant as before:

> **After i phases of the algorithm, *dist*(v) is *at most* the length of the shortest walk from s to v consisting of *at most* i edges.**

As before, it is straightforward to prove by induction (hint, hint) that this invariant holds for every integer i and vertex v.

21.7 Shimbel's Algorithm as Dynamic Programming

Shimbel's algorithm can also be recast as a dynamic programming algorithm. Let $dist_i(v)$ denote the length of the shortest path $s \rightsquigarrow v$ consisting of at most i edges. It's not hard to see that this

[5]In fact, this is essentially the formulation proposed by both Shimbel and Bellman. Bob Tarjan recognized in the early 1980s that Shimbel's algorithm is equivalent to Dijkstra's algorithm with a queue instead of a heap.

function obeys the following recurrence:

$$dist_i(v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min \left\{ \begin{array}{c} dist_{i-1}(v), \\ \min\limits_{u \to v \in E} (dist_{i-1}(u) + w(u \to v)) \end{array} \right\} & \text{otherwise} \end{cases}$$

For the moment, let's assume the graph has no negative cycles; our goal is to compute $dist_{V-1}(t)$. We can clearly memoize this two-parameter function into a two-dimensional array. A straightforward dynamic programming evaluation of this recurrence looks like this:

```
ShimbelDP(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i−1,v]
          for every edge u→v
              if dist[i,v] > dist[i−1,u] + w(u→v)
                  dist[i,v] ← dist[i−1,u] + w(u→v)
```

Now let us make two minor changes to this algorithm. First, we remove one level of indentation from the last three lines. This may change the order in which we examine edges, but the modified algorithm still computes $dist_i(v)$ for all i and v. Second, we change the indices in the last two lines from $i-1$ to i. This change may cause the distances $dist[i,v]$ to approach the true shortest-path distances more quickly than before, but the algorithm is still correct.

```
ShimbelDP2(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i−1,v]
      for every edge u→v
          if dist[i,v] > dist[i,u] + w(u→v)
              dist[i,v] ← dist[i,u] + w(u→v)
```

Now notice that the iteration index i is completely redundant! We really only need to keep a one-dimensional array of distances, which means we don't need to scan the vertices in each iteration of the main loop.

```
ShimbelDP3(s)
  dist[s] ← 0
  for every vertex v ≠ s
      dist[v] ← ∞
  for i ← 1 to V − 1
      for every edge u→v
          if dist[v] > dist[u] + w(u→v)
              dist[v] ← dist[u] + w(u→v)
```

The resulting algorithm is almost identical to our earlier algorithm SHIMBELSSSP! The first three lines initialize the shortest path distances, and the last two lines check whether an edge is tense, and if so, relaxes it. The only feature missing from the new algorithm is explicit maintenance of predecessors, but that's easy to add.

Exercises

0. Prove that the following invariant holds for every integer i and every vertex v: After i phases of Shimbel's algorithm (in either formulation), $dist(v)$ is at most the length of the shortest path $s \rightsquigarrow v$ consisting of at most i edges.

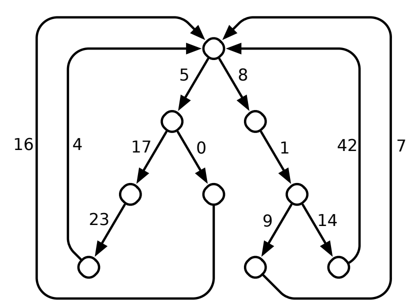
1. Let G be a directed graph with edge weights (which may be positive, negative, or zero), and let s be an arbitrary vertex of G.

   (a) Suppose every vertex v stores a number $dist(v)$. Describe and analyze an algorithm to determine whether $dist(v)$ is the shortest-path distance from s to v, for every vertex v.

   (b) Suppose instead that every vertex $v \neq s$ stores a pointer $pred(v)$ to another vertex in G. Describe and analyze an algorithm to determine whether these predecessor pointers define a single-source shortest path tree rooted at s.

   Do ***not*** assume that G contains no negative cycles.

2. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight.



A looped tree.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?

   (b) Describe and analyze a faster algorithm.

3. Suppose we are given an undirected graph G in which every *vertex* has a positive weight.

   (a) Describe and analyze an algorithm to find a *spanning tree* of G with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.)

   (b) Describe and analyze an algorithm to find a *path* in G from one given vertex s to another given vertex t with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.)

4. For any edge e in any graph G, let $G \setminus e$ denote the graph obtained by deleting e from G.

   (a) Suppose we are given a directed graph G in which the shortest path σ from vertex s to vertex t passes through *every* vertex of G. Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G, in $O(E \log V)$ time. Your algorithm should output a set of E shortest-path distances, one for each edge of the input graph. You may assume that all edge weights are non-negative. *[Hint: If we delete an edge of the original shortest path, how do the old and new shortest paths overlap?]*

   *(b) Let s and t be *arbitrary* vertices in an arbitrary *undirected* graph G. Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G, in $O(E \log V)$ time. Again, you may assume that all edge weights are non-negative.

5. Let $G = (V, E)$ be a connected directed graph with non-negative edge weights, let s and t be vertices of G, and let H be a subgraph of G obtained by deleting some edges. Suppose we want to reinsert exactly one edge from G back into H, so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm that chooses the best edge to reinsert, in $O(E \log V)$ time.

6. When there is more than one shortest path from one node s to another node t, it is often convenient to choose a shortest path with the fewest edges; call this the ***best path*** from s to t. Suppose we are given a directed graph G with positive edge weights and a source vertex s in G. Describe and analyze an algorithm to compute *best* paths in G from s to every other vertex.

*7. (a) Prove that Ford's generic shortest-path algorithm (while the graph contains a tense edge, relax it) can take exponential time in the worst case when implemented with a stack instead of a priority queue (like Dijkstra) or a queue (like Shimbel). Specifically, for every positive integer n, construct a weighted directed n-vertex graph G_n, such that the stack-based shortest-path algorithm call RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: Towers of Hanoi.]*

   (b) Prove that Dijkstra's shortest-path algorithm can require exponential time in the worst case when edges are allowed to have negative weight. Specifically, for every positive integer n, construct a weighted directed n-vertex graph G_n, such that Dijkstra's algorithm calls RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: This is relatively easy if you've already solved part (a).]*

8. (a) Describe and analyze a modification of Shimbel's shortest-path algorithm that actually returns a negative cycle if any such cycle is reachable from s, or a shortest-path tree if there is no such cycle. The modified algorithm should still run in $O(VE)$ time.

   (b) Describe and analyze a modification of Shimbel's shortest-path algorithm that computes the correct shortest path distances from s to every other vertex of the input graph, even if the graph contains negative cycles. Specifically, if any walk from s to v contains a negative cycle, your algorithm should end with $dist(v) = -\infty$; otherwise, $dist(v)$ should contain the length of the shortest path from s to v. The modified algorithm should still run in $O(VE)$ time.

*(c) Repeat parts (a) and (b), but for Ford's generic shortest-path algorithm. You may assume that the unmodified algorithm halts in $O(2^V)$ steps if there is no negative cycle; your modified algorithms should also run in $O(2^V)$ time.

*9. Describe and analyze an efficient algorithm to compute the *number* of shortest paths between two specified vertices s and t in a directed graph G whose edges have positive weights. *[Hint: Which edges of G can lie on a shortest path from s to t?]*

10. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where *exactly* should you meet?

    You are given a weighted graph $G = (V, E)$, where the vertices V represent cities and the edges E represent roads that directly connect cities. Each edge e has a weight $w(e)$ equal to the time required to travel between the two cities. You are also given a vertex p, representing your starting location, and a vertex q, representing your friend's starting location.

    Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as quickly as possible.

11. After a grueling algorithms midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between bus lines at least once.

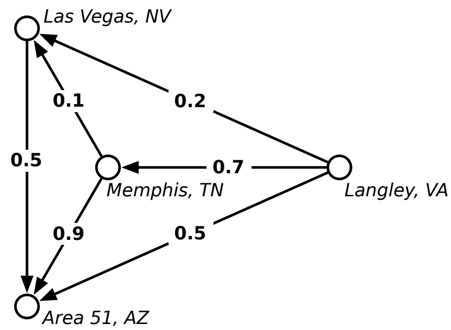
    Describe and analyze an algorithm to determine the sequence of bus rides that will get you home as early as possible, assuming there are b different bus lines, and each bus stops n times per day. Your goal is to minimize your *arrival time*, not the time you actually spend traveling. Assume that the buses run exactly on schedule, that you have an accurate watch, and that you are too tired to walk between bus stops.

12. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

    Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights. *[Hint: Modify the input data and apply Dijkstra's algorithm.]*

13. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.



For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted! (That's how they got Elvis, you know.) Although this example is a dag, your algorithm must handle *arbitrary* directed graphs.

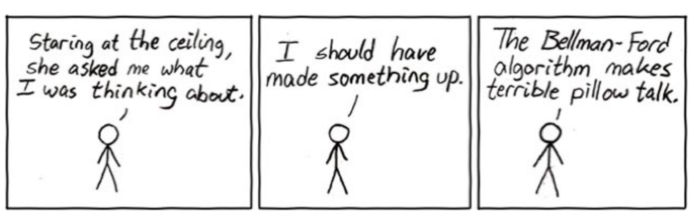
14. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, "Get out... while... you...", thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

    You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

    On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can't stand each other's company, so you'll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

    You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger's heart, you wisely decide to leave the park immediately.

    Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along the path is as small as possible. *Be sure to account for **both** the vertex probabilities **and** the edge probabilities!*



— Randall Munroe, *xkcd* (http://xkcd.com/69/)

The tree which fills the arms grew from the tiniest sprout;
the tower of nine storeys rose from a (small) heap of earth;
the journey of a thousand li commenced with a single step.

— Lao-Tzu, *Tao Te Ching*, chapter 64 (6th century BC),
translated by J. Legge (1891)

And I would walk five hundred miles,
And I would walk five hundred more,
Just to be the man who walks a thousand miles
To fall down at your door.

— The Proclaimers, "Five Hundred Miles (I'm Gonna Be)",
Sunshine on Leith (2001)

Almost there... Almost there...

— Red Leader [Drewe Henley], *Star Wars* (1977)

22 All-Pairs Shortest Paths

In the previous lecture, we saw algorithms to find the shortest path from a source vertex s to a target vertex t in a directed graph. As it turns out, the best algorithms for this problem actually find the shortest path from s to every possible target (or from every possible source to t) by constructing a shortest path tree. The shortest path tree specifies two pieces of information for each node v in the graph:

- $dist(v)$ is the length of the shortest path (if any) from s to v;
- $pred(v)$ is the second-to-last vertex (if any) the shortest path (if any) from s to v.

In this lecture, we want to generalize the shortest path problem even further. In the *all pairs shortest path* problem, we want to find the shortest path from *every* possible source to *every* possible destination. Specifically, for every pair of vertices u and v, we need to compute the following information:

- $dist(u, v)$ is the length of the shortest path (if any) from u to v;
- $pred(u, v)$ is the second-to-last vertex (if any) on the shortest path (if any) from u to v.

For example, for any vertex v, we have $dist(v, v) = 0$ and $pred(v, v) = \textsc{Null}$. If the shortest path from u to v is only one edge long, then $dist(u, v) = w(u \to v)$ and $pred(u, v) = u$. If there is *no* shortest path from u to v—either because there's no path at all, or because there's a negative cycle—then $dist(u, v) = \infty$ and $pred(v, v) = \textsc{Null}$.

The output of our shortest path algorithms will be a pair of $V \times V$ arrays encoding all V^2 distances and predecessors. Many maps include a distance matrix—to find the distance from (say) Champaign to (say) Columbus, you would look in the row labeled 'Champaign' and the column labeled 'Columbus'. In these notes, I'll focus almost exclusively on computing the distance array. The predecessor array, from which you would compute the actual shortest paths, can be computed with only minor additions to the algorithms I'll describe (hint, hint).

22.1 Lots of Single Sources

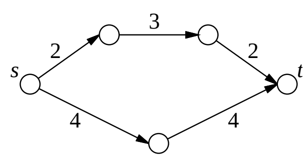
The obvious solution to the all-pairs shortest path problem is just to run a single-source shortest path algorithm V times, once for every possible source vertex! Specifically, to fill in the one-dimensional subarray $dist[s,\cdot]$, we invoke either Dijkstra's or Shimbel's algorithm starting at the source vertex s.

```
OBVIOUSAPSP(V, E, w):
  for every vertex s
      dist[s,·] ← SSSP(V, E, w, s)
```

The running time of this algorithm depends on which single-source shortest path algorithm we use. If we use Shimbel's algorithm, the overall running time is $\boldsymbol{\Theta(V^2E)} = O(V^4)$. If all the edge weights are non-negative, we can use Dijkstra's algorithm instead, which decreases the running time to $\boldsymbol{\Theta(VE + V^2 \log V)} = O(V^3)$. For graphs with negative edge weights, Dijkstra's algorithm can take exponential time, so we can't get this improvement directly.

22.2 Reweighting

One idea that occurs to most people is increasing the weights of all the edges by the same amount so that all the weights become positive, and then applying Dijkstra's algorithm. Unfortunately, this simple idea doesn't work. Different paths change by different amounts, which means the shortest paths in the reweighted graph may not be the same as in the original graph.



Increasing all the edge weights by 2 changes the shortest path s to t.

However, there is a more complicated method for reweighting the edges in a graph. Suppose each vertex v has some associated *cost* $c(v)$, which might be positive, negative, or zero. We can define a new weight function w' as follows:

$$w'(u \to v) = c(u) + w(u \to v) - c(v)$$

To give some intuition, imagine that when we leave vertex u, we have to pay an exit tax of $c(u)$, and when we enter v, we get $c(v)$ as an entrance gift.

Now it's not too hard to show that the shortest paths with the new weight function w' are exactly the same as the shortest paths with the original weight function w. In fact, for *any* path $u \rightsquigarrow v$ from one vertex u to another vertex v, we have

$$w'(u \rightsquigarrow v) = c(u) + w(u \rightsquigarrow v) - c(v).$$

We pay $c(u)$ in exit fees, plus the original weight of of the path, minus the $c(v)$ entrance gift. At every intermediate vertex x on the path, we get $c(x)$ as an entrance gift, but then immediately pay it back as an exit tax!

22.3 Johnson's Algorithm

Johnson's all-pairs shortest path algorithm finds a cost $c(v)$ for each vertex, so that when the graph is reweighted, every edge has non-negative weight.

Suppose the graph has a vertex s that has a path to every other vertex. Johnson's algorithm computes the shortest paths from s to every other vertex, using Shimbel's algorithm (which doesn't care if the edge weights are negative), and then sets $c(v) \leftarrow dist(s,v)$, so the new weight of every edge is

$$w'(u \to v) = dist(s,u) + w(u \to v) - dist(s,v).$$

Why are all these new weights non-negative? Because otherwise, Shimbel's algorithm wouldn't be finished! Recall that an edge $u \to v$ is *tense* if $dist(s,u) + w(u \to v) < dist(s,v)$, and that single-source shortest path algorithms eliminate all tense edges. The only exception is if the graph has a negative cycle, but then shortest paths aren't defined, and Johnson's algorithm simply aborts.

But what if the graph *doesn't* have a vertex s that can reach everything? No matter where we start Shimbel's algorithm, some of those vertex costs will be infinite. Johnson's algorithm avoids this problem by adding a new vertex s to the graph, with zero-weight edges going from s to every other vertex, but *no* edges going back into s. This addition doesn't change the shortest paths between any other pair of vertices, because there are no paths into s.

So here's Johnson's algorithm in all its glory.

```
JohnsonAPSP(V, E, w):
    create a new vertex s
    for every vertex v
        w(s→v) ← 0
        w(v→s) ← ∞
    dist[s,·] ← Shimbel(V, E, w, s)
    if Shimbel found a negative cycle
        fail gracefully
    for every edge (u,v) ∈ E
        w'(u→v) ← dist[s,u] + w(u→v) − dist[s,v]
    for every vertex u
        dist[u,·] ← Dijkstra(V, E, w', u)
        for every vertex v
            dist[u,v] ← dist[u,v] − dist[s,u] + dist[s,v]
```

The algorithm spends $\Theta(V)$ time adding the artificial start vertex s, $\Theta(VE)$ time running Shimbel, $O(E)$ time reweighting the graph, and then $\Theta(VE + V^2 \log V)$ running V passes of Dijkstra's algorithm. Thus, the overall running time is $\boldsymbol{\Theta(VE + V^2 \log V)}$.

22.4 Dynamic Programming

There's a completely different solution to the all-pairs shortest path problem that uses dynamic programming instead of a single-source algorithm. For *dense* graphs where $E = \Omega(V^2)$, the dynamic programming approach eventually leads to the same $O(V^3)$ running time as Johnson's algorithm, but with a much simpler algorithm. In particular, the new algorithm avoids Dijkstra's algorithm, which gets its efficiency from Fibonacci heaps, which are rather easy to screw up in the implementation. **In the rest of this lecture, I will assume that the input graph contains no negative cycles.**

As usual for dynamic programming algorithms, we first need to come up with a recursive formulation of the problem. Here is an "obvious" recursive definition for $dist(u,v)$:

$$dist(u,v) = \begin{cases} 0 & \text{if } u = v \\ \min\limits_{x \to v} \big(dist(u,x) + w(x \to v) \big) & \text{otherwise} \end{cases}$$

In other words, to find the shortest path from u to v, we consider all possible last edges $x \to v$ and recursively compute the shortest path from u to x. **Unfortunately, this recurrence doesn't work!** To compute $dist(u,v)$, we may need to compute $dist(u,x)$ for every other vertex x. But to compute $dist(u,x)$, we may need to compute $dist(u,v)$. We're stuck in an infinite loop!

To avoid this circular dependency, we need an additional parameter that decreases at each recursion, eventually reaching zero at the base case. One possibility is to include the number of edges in the shortest path as this third magic parameter, just as we did in the dynamic programming formulation of Shimbel's algorithm. Let $dist(u,v,k)$ denote the length of the shortest path from u to v that uses *at most k edges*. Since we know that the shortest path between any two vertices has at most $V-1$ vertices, $dist(u,v,V-1)$ is the actual shortest-path distance. As in the single-source setting, we have the following recurrence:

$$dist(u,v,k) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{if } k = 0 \text{ and } u \neq v \\ \min\limits_{x \to v} \big(dist(u,x,k-1) + w(x \to v)\big) & \text{otherwise} \end{cases}$$

Turning this recurrence into a dynamic programming algorithm is straightforward. To make the algorithm a little shorter, let's assume that $w(v \to v) = 0$ for every vertex v. Assuming the graph is stored in an adjacency list, the resulting algorithm runs in $\boldsymbol{\Theta(V^2E)}$ ***time***.

```
DynamicProgrammingAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v, 0] ← 0
          else
              dist[u, v, 0] ← ∞
  for k ← 1 to V − 1
      for all vertices u
          dist[u, u, k] ← 0
          for all vertices v ≠ u
              dist[u, v, k] ← ∞
              for all edges x→v
                  if dist[u, v, k] > dist[u, x, k − 1] + w(x→v)
                      dist[u, v, k] ← dist[u, x, k − 1] + w(x→v)
```

This algorithm was first sketched by Shimbel in 1955; in fact, this algorithm is just running V different instances of Shimbel's single-source algorithm, one for each possible source vertex. Just as in the dynamic programming development of Shimbel's single-source algorithm, we don't actually need the inner loop over vertices v, and we only need a two-dimensional table. After the kth iteration of the main loop in the following algorithm, $dist[u,v]$ lies between the true shortest path distance from u to v and the value $dist[u,v,k]$ computed in the previous algorithm.

```
SHIMBELAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v] ← 0
          else
              dist[u, v] ← ∞
  for k ← 1 to V − 1
      for all vertices u
          for all edges x→v
              if dist[u, v] > dist[u, x] + w(x→v)
                  dist[u, v] ← dist[u, x] + w(x→v)
```

22.5 Divide and Conquer

But we can make a more significant improvement. The recurrence we just used broke the shortest path into a slightly shorter path and a single edge, by considering all predecessors. Instead, let's break it into two shorter paths at the *middle* vertex of the path. This idea gives us a different recurrence for $dist(u, v, k)$. Once again, to simplify things, let's assume $w(v\to v) = 0$.

$$dist(u, v, k) = \begin{cases} w(u\to v) & \text{if } k = 1 \\ \min\limits_x \big(dist(u, x, k/2) + dist(x, v, k/2)\big) & \text{otherwise} \end{cases}$$

This recurrence only works when k is a power of two, since otherwise we might try to find the shortest path with a fractional number of edges! But that's not really a problem, since $dist(u, v, 2^{\lceil \lg V \rceil})$ gives us the overall shortest distance from u to v. Notice that we use the base case $k = 1$ instead of $k = 0$, since we can't use half an edge.

Once again, a dynamic programming solution is straightforward. Even before we write down the algorithm, we can tell the running time is $\mathbf{\Theta(V^3 \log V)}$—we consider V possible values of u, v, and x, but only $\lceil \lg V \rceil$ possible values of k.

```
FASTDYNAMICPROGRAMMINGAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)
  for i ← 1 to ⌈lg V⌉          ⟨⟨k = 2^i⟩⟩
      for all vertices u
          for all vertices v
              dist[u, v, i] ← ∞
              for all vertices x
                  if dist[u, v, i] > dist[u, x, i − 1] + dist[x, v, i − 1]
                      dist[u, v, i] ← dist[u, x, i − 1] + dist[x, v, i − 1]
```

This algorithm is ***not*** the same as V invocations of any single-source algorithm; in particular, the innermost loop does not simply relax tense edges. However, we can remove the last dimension of the table, using $dist[u, v]$ everywhere in place of $dist[u, v, i]$, just as in Shimbel's single-source algorithm, thereby reducing the space from $O(V^3)$ to $O(V^2)$.

```
FASTSHIMBELAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)
  for i ← 1 to ⌈lg V⌉
      for all vertices u
          for all vertices v
              for all vertices x
                  if dist[u, v] > dist[u, x] + dist[x, v]
                      dist[u, v] ← dist[u, x] + dist[x, v]
```

This faster algorithm was discovered by Leyzorek *et al.* in 1957, in the same paper where they describe Dijkstra's algorithm.

22.6 Aside: 'Funny' Matrix Multiplication

There is a very close connection (first observed by Shimbel, and later independently by Bellman) between computing shortest paths in a directed graph and computing powers of a square matrix. Compare the following algorithm for multiplying two $n \times n$ matrices A and B with the inner loop of our first dynamic programming algorithm. (I've changed the variable names in the second algorithm slightly to make the similarity clearer.)

```
MATRIXMULTIPLY(A, B):
  for i ← 1 to n
      for j ← 1 to n
          C[i, j] ← 0
          for k ← 1 to n
              C[i, j] ← C[i, j] + A[i, k] · B[k, j]
```

```
APSPINNERLOOP:
  for all vertices u
      for all vertices v
          D′[u, v] ← ∞
          for all vertices x
              D′[u, v] ← min{D′[u, v], D[u, x] + w[x, v]}
```

The *only* difference between these two algorithms is that we use addition instead of multiplication and minimization instead of addition. For this reason, the shortest path inner loop is often referred to as 'funny' matrix multiplication.

DYNAMICPROGRAMMINGAPSP is the standard iterative algorithm for computing the $(V-1)$th 'funny power' of the weight matrix w. The first set of for loops sets up the 'funny identity matrix', with zeros on the main diagonal and infinity everywhere else. Then each iteration of the second main for loop computes the next 'funny power'. FASTDYNAMICPROGRAMMINGAPSP replaces this iterative method for computing powers with repeated squaring, exactly like we saw at the beginning of the semester. The fast algorithm is simplified slightly by the fact that unless there are negative cycles, every 'funny power' after the Vth is the same.

There are faster methods for multiplying matrices, similar to Karatsuba's divide-and-conquer algorithm for multiplying integers. (Google for 'Strassen's algorithm'.) Unfortunately, these algorithms us subtraction, and there's no 'funny' equivalent of subtraction. (What's the inverse operation for min?) So at least for general graphs, there seems to be no way to speed up the inner loop of our dynamic programming algorithms.

Fortunately, this isn't true. There a beautiful randomized algorithm, discovered by Alon, Galil, Margalit, and Noar[1], that computes all-pairs shortest paths in undirected graphs in $O(M(V)\log^2 V)$ expected time, where $M(V)$ is the time to multiply two $V \times V$ integer matrices. A simplified version of this algorithm for *unweighted* graphs was discovered by Seidel.[2]

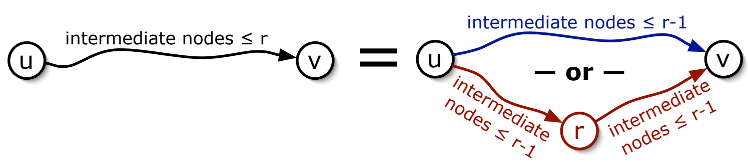
22.7 Floyd-(Roy-Kleene-)Warshall

Our fast dynamic programming algorithm is still a factor of $O(\log V)$ slower than Johnson's algorithm. A different formulation that removes this logarithmic factor was proposed in 1962 by Robert Floyd, slightly generalizing an algorithm of Stephen Warshall published earlier in the same year. (In fact, Warshall's algorithm was independently discovered by Bernard Roy in 1959, but the underlying technique was used even earlier by Stephen Kleene[3] in 1951.) Warshall's (and Roy's and Kleene's) insight was to use a different third parameter in the dynamic programming recurrence.

Number the vertices arbitrarily from 1 to V. For every pair of vertices u and v and every integer r, we define a path $\pi(u,v,r)$ as follows:

> $\pi(u,v,r)$:= the shortest path from u to v where every intermediate vertex (that is, every vertex except u and v) is numbered at most r.

If $r = 0$, we aren't allowed to use any intermediate vertices, so $\pi(u,v,0)$ is just the edge (if any) from u to v. If $r > 0$, then either $\pi(u,v,r)$ goes through the vertex numbered r, or it doesn't. If $\pi(u,v,r)$ does contain vertex r, it splits into a subpath from u to r and a subpath from r to v, where every intermediate vertex in these two subpaths is numbered at most $r-1$. Moreover, the subpaths are as short as possible with this restriction, so they must be $\pi(u,r,r-1)$ and $\pi(r,v,r-1)$. On the other hand, if $\pi(u,v,r)$ does not go through vertex r, then every intermediate vertex in $\pi(u,v,r)$ is numbered at most $r-1$; since $\pi(u,v,r)$ must be the *shortest* such path, we have $\pi(u,v,r) = \pi(u,v,r-1)$.



Recursive structure of the restricted shortest path $\pi(u,v,r)$.

This recursive structure implies the following recurrence for the length of $\pi(u,v,r)$, which we will denote by $dist(u,v,r)$:

$$dist(u,v,r) = \begin{cases} w(u\to v) & \text{if } r = 0 \\ \min\left\{dist(u,v,r-1),\ dist(u,r,r-1) + dist(r,v,r-1)\right\} & \text{otherwise} \end{cases}$$

[1]Noga Alon, Zvi Galil, Oded Margalit*, and Moni Naor. Witnesses for Boolean matrix multiplication and for shortest paths. *Proc. 33rd FOCS* 417-426, 1992. See also Noga Alon, Zvi Galil, Oded Margalit*. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences* 54(2):255–262, 1997.

[2]Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995. This is one of the few algorithms papers where (in the conference version at least) the algorithm is completely described and analyzed *in the abstract* of the paper.

[3]Pronounced "clay knee", not "clean" or "clean-ee" or "clay-nuh" or "dimaggio".

We need to compute the shortest path distance from u to v with no restrictions, which is just $dist(u, v, V)$. Once again, we should immediately see that a dynamic programming algorithm will implement this recurrence in $\boldsymbol{\Theta(V^3)}$ time.

```
FloydWarshall(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)
  for r ← 1 to V
      for all vertices u
          for all vertices v
              if dist[u, v, r − 1] < dist[u, r, r − 1] + dist[r, v, r − 1]
                  dist[u, v, r] ← dist[u, v, r − 1]
              else
                  dist[u, v, r] ← dist[u, r, r − 1] + dist[r, v, r − 1]
```

Just like our earlier algorithms, we can simplify the algorithm by removing the third dimension of the memoization table. Also, because the vertex numbering was chosen arbitrarily, there's no reason to refer to it explicitly in the pseudocode.

```
FloydWarshall2(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)
  for all vertices r
      for all vertices u
          for all vertices v
              if dist[u, v] > dist[u, r] + dist[r, v]
                  dist[u, v] ← dist[u, r] + dist[r, v]
```

Now compare this algorithm with FastShimbelAPSP. Instead of $O(\log V)$ passes through all triples of vertices, FloydWarshall2 only requires a single pass, but only because it uses a different nesting order for the three for-loops!

22.8 Converting DFAs to regular expressions

Floyd's algorithm is a special case of a more general method for solving problems involving paths between vertices in graphs. The earliest example (that I know of) of this technique is an 1951 algorithm of Stephen Kleene to convert a deterministic finite automaton into an equivalent regular expression.

Recall that a deterministic finite automaton (DFA) formally consists of the following components:

- A finite set Σ, called the ***alphabet***, and whose elements we call ***symbols***.
- A finite set Q, whose elements are called ***states***.
- An ***initial state*** $s \in Q$.
- A subset $A \subseteq Q$ of ***accepting states***.
- A ***transition function*** $\delta : Q \times \Sigma \to Q$.

The ***extended transition function*** $\delta^*: Q \times \Sigma^* \to Q$ is recursively defined as follows:

$$\delta^*(q, w) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*(\delta(q, a), x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^*. \end{cases}$$

Finally, a DFA ***accepts*** a string $w \in \Sigma^*$ if and only if $\delta^*(s, w) \in A$.

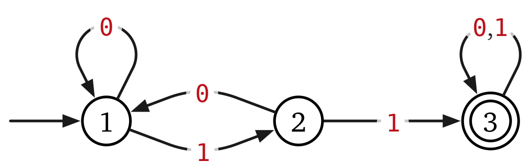
Equivalently, a DFA is a directed (multi-)graph with labeled edges whose vertices are the states, such that each vertex (state) has exactly one outgoing edge (transition) labeled with each symbol in Σ. There is a special "start" vertex s, and a subset A of the vertices are marked as "accepting". For any string $w \in \Sigma^*$, there is a unique walk starting at s whose sequence of edge labels is w. The DFA accepts w if and only if this walk ends at a state in A.

Kleene described the following algorithm to convert DFAs into equivalent regular expressions. Suppose we are given a DFA M with n states, where (without loss of generality) each state is identified by an integer between 1 and n. Let $\boldsymbol{L(i, j, r)}$ denote the set of all strings that describe walks in M that start at state i and end at state j, such that every intermediate state has index at most r. Thus, the language accepted by M is precisely

$$L(M) = \bigcup_{q \in A} L(s, q, n).$$

We prove inductively that every language $L(i, j, r)$ is regular, by recursively constructing a regular expression $R(i, j, r)$ that represents $L(i, j, r)$. There are two cases to consider.

- First, suppose $r = 0$. The language $L(i, j, 0)$ contains the labels walks from state i to state j that do not pass through *any* intermediate states. Thus, every string in $L(i, j, 0)$ has length at most 1. Specifically, for any symbol $a \in \Sigma$, we have $a \in L(i, j, 0)$ if and only if $\delta(i, a) = j$, and we have $\varepsilon \in L(i, j, 0)$ if and only if $i = j$. Thus, $L(i, j, 0)$ is always finite, and therefore regular.

  An example DFA

  For example, the DFA sown on the next page defines the following regular languages $L(i, j, 0)$.

  $$\begin{array}{lll} R[1,1,0] = \varepsilon + 0 & R[2,1,0] = 0 & R[3,1,0] = \varnothing \\ R[1,2,0] = 1 & R[2,2,0] = \varepsilon & R[3,2,0] = \varnothing \\ R[1,3,0] = \varnothing & R[2,3,0] = 1 & R[3,3,0] = \varepsilon + 0 + 1 \end{array}$$

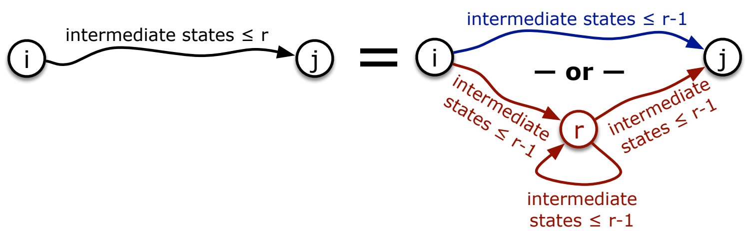
- Now suppose $r > 0$. Each string $w \in L(i, j, r)$ describes a walk from state i to state j where every intermediate state has index at most r. If this walk does not pass through state r, then $w \in L(i, j, r-1)$ by definition. Otherwise, we can split w into a sequence of substrings $w = w_1 \cdot w_2 \cdots w_\ell$ at the points where the walk visits state r. These substrings have the following properties:

  – The prefix w_1 describes a walk from state i to state r and thus belongs to $L(i, r, r-1)$.

- The suffix w_ℓ describes a walk from state r to state j and thus belongs to $L(r, j, r-1)$.
- For every other index k, the substring w_k describes a walk from state r to state r and thus belongs to $L(r, r, r-1)$.

We conclude that

$$L(i, j, r) \;=\; L(i, j, r-1) \,\cup\, L(i, r, r-1) \bullet L(r, r, r-1)^* \bullet L(r, j, r-1).$$



Recursive structure of the regular language $L(i, j, r)$.

Putting these pieces together, we can recursively define a regular ***expression*** $R(i, j, r)$ that describes the language $L(i, j, r)$, as follows:

$$R(i,j,r) := \begin{cases} \varepsilon + \sum_{\delta(i,a)=j} a & \text{if } r = 0 \text{ and } i = j \\ \sum_{\delta(i,a)=j} a & \text{if } r = 0 \text{ and } i \neq j \\ R(i,j,r-1) + R(i,r,r-1) \bullet R(r,r,r-1)^* \bullet R(r,j,r-1) & \text{otherwise} \end{cases}$$

Kleene's algorithm evaluates this recurrence bottom-up using the natural dynamic programming algorithm. We memoize the previous recurrence into a three-dimensional array $R[1..n, 1..n, 0..n]$, which we traverse by increasing r in the outer loop, and in arbitrary order in the inner two loops.

```
KLEENE(Σ, n, δ, F):
  ⟨⟨Base cases⟩⟩
  for i ← 1 to n
      for j ← 1 to n
          if i = j  then  R[i, j, 0] ← ε  else  R[i, j, 0] ← ∅
          for all symbols a ∈ Σ
              if δ[i, a] = j
                  R[i, j, 0] ← R[i, j, 0] + a
  ⟨⟨Recursive cases⟩⟩
  for r ← 1 to n
      for i ← 1 to n
          for j ← 1 to n
              R[i, j, r] ← R[i, j, r−1] + R[i, r, r−1] • R[r, r, r−1]* • R[r, j, r−1]
  ⟨⟨Assemble the final result⟩⟩
  R ← ∅
  for q ← 0 to n−1
      if q ∈ F
          R ← R + R[1, q, n−1]
      return R
```

For purposes of analysis, let's assume the alphabet Σ has constant size. Assuming each alternation $(+)$, concatenation $(\bullet)$, and Kleene closure $(^*)$ operation requires constant time, the entire algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

However, regular expressions over an alphabet Σ are normally represented either as standard strings (arrays) over the larger alphabet $\Sigma \cup \{+, \bullet, *, (,), \varepsilon\}$, or as regular expression *trees*, whose internal nodes are $+$, $\bullet$, and $*$ operators and whose leaves are symbols and εs. In either representation, the regular expressions in Kleene's algorithm grow in size by roughly a factor of 4 in each iteration of the outer loop, at least in the worst case. Thus, in the worst case, each regular expression $R[i, j, r]$ has size $O(4^r)$, the size of the final output expression is $O(4^n n)$, and entire algorithm runs in ***$O(4^n n^2)$ time***.

So we shouldn't do this. After all, the running time is exponential, and exponential time is bad. Right? Moreover, this exponential dependence is unavoidable; Hermann Gruber and Markus Holzer proved in 2008[4] that there are n-state DFAs over the binary alphabet $\{0, 1\}$ such that any equivalent regular expression has length $2^{\Omega(n)}$.

Well, maybe it's not so bad. The output regular expression has exponential size *because it contains multiple copies of the same subexpressions*; similarly, the regular expression tree has exponential size *because it contains multiples copies of several subtrees*. But it's precisely this exponential behavior that we use dynamic programming to avoid! In fact, it's not hard to modify Kleene's algorithm to compute a ***regular expression dag*** of size $O(n^3)$, ***in $O(n^3)$ time***, that (intuitively) contains each subexpression $R[i, j, r]$ only once. This regular expression dag has exactly the same relationship to the regular expression *tree* as the dependency graph of Kleene's algorithm has to the recursion tree of its underlying recurrence.

Exercises

1. All of the algorithms discussed in this lecture fail if the graph contains a negative cycle. Johnson's algorithm detects the negative cycle in the initialization phase (via Shimbel's algorithm) and aborts; the dynamic programming algorithms just return incorrect results. However, all of these algorithms can be modified to return correct shortest-path distances, even in the presence of negative cycles. Specifically, if there is a path from vertex u to a negative cycle and a path from that negative cycle to vertex v, the algorithm should report that $dist[u, v] = -\infty$. If there is no directed path from u to v, the algorithm should return $dist[u, v] = \infty$. Otherwise, $dist[u, v]$ should equal the length of the shortest directed path from u to v.

   (a) Describe how to modify Johnson's algorithm to return the correct shortest-path distances, even if the graph has negative cycles.

   (b) Describe how to modify the Floyd-Warshall algorithm (FloydWarshall2) to return the correct shortest-path distances, even if the graph has negative cycles.

2. All of the shortest-path algorithms described in this note can also be modified to return an explicit description of some negative cycle, instead of simply reporting that a negative cycle exists.

   (a) Describe how to modify Johnson's algorithm to return either the matrix of shortest-path distances or a negative cycle.

   (b) Describe how to modify the Floyd-Warshall algorithm (FloydWarshall2) to return either the matrix of shortest-path distances or a negative cycle.

[4]Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. *Proc. 35th ICALP*, 39–50, 2008.

If the graph contains more than one negative cycle, your algorithms may choose one arbitrarily.

3. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero. Suppose the vertices of G are partitioned into k disjoint subsets $V_1, V_2, \ldots, V_k$; that is, every vertex of G belongs to exactly one subset V_i. For each i and j, let $\delta(i, j)$ denote the minimum shortest-path distance between vertices in V_i and vertices in V_j:
$$\delta(i, j) = \min\left\{ dist(u, v) \;\middle|\; u \in V_i \text{ and } v \in V_j \right\}.$$
Describe an algorithm to compute $\delta(i, j)$ for all i and j in time $O(V^2 + kE \log E)$.

4. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero.

   (a) How could we delete an arbitrary vertex v from this graph, without changing the shortest-path distance between any other pair of vertices? Describe an algorithm that constructs a directed graph $G' = (V', E')$ with weighted edges, where $V' = V \setminus \{v\}$, and the shortest-path distance between any two nodes in H is equal to the shortest-path distance between the same two nodes in G, in $O(V^2)$ time.

   (b) Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances from v to every other vertex, and from every other vertex to v, in the original graph G, in $O(V^2)$ time.

   (c) Combine parts (a) and (b) into another all-pairs shortest path algorithm that runs in $O(V^3)$ time. (The resulting algorithm is *not* the same as Floyd-Warshall!)

5. In this problem we will discover how you, too, can be employed by Wall Street and cause a major economic collapse! The *arbitrage* business is a money-making scheme that takes advantage of differences in currency exchange. In particular, suppose that 1 US dollar buys 120 Japanese yen; 1 yen buys 0.01 euros; and 1 euro buys 1.2 US dollars. Then, a trader starting with \$1 can convert his money from dollars to yen, then from yen to euros, and finally from euros back to dollars, ending with \$1.44! The cycle of currencies $\$ \to ¥ \to € \to \$$ is called an ***arbitrage cycle***. Of course, finding and exploiting arbitrage cycles before the prices are corrected requires extremely fast algorithms.

   Suppose n different currencies are traded in your currency market. You are given the matrix $R[1..n, 1..n]$ of exchange rates between every pair of currencies; for each i and j, one unit of currency i can be traded for $R[i, j]$ units of currency j. (Do *not* assume that $R[i, j] \cdot R[j, i] = 1$.)

   (a) Describe an algorithm that returns an array $V[1..n]$, where $V[i]$ is the maximum amount of currency i that you can obtain by trading, starting with one unit of currency 1, assuming there are no arbitrage cycles.

   (b) Describe an algorithm to determine whether the given matrix of currency exchange rates creates an arbitrage cycle.

   (c) Modify your algorithm from part (b) to actually return an arbitrage cycle, if it exists.

*6. Let $G = (V, E)$ be an undirected, unweighted, connected, n-vertex graph, represented by the adjacency matrix $A[1..n, 1..n]$. In this problem, we will derive Seidel's sub-cubic algorithm to compute the $n \times n$ matrix $D[1..n, 1..n]$ of shortest-path distances using fast matrix multiplication. Assume that we have a subroutine MatrixMultiply that multiplies two $n \times n$ matrices in $\Theta(n^\omega)$ time, for some unknown constant $\omega \geq 2$.[5]

(a) Let G^2 denote the graph with the same vertices as G, where two vertices are connected by a edge if and only if they are connected by a path of length at most 2 in G. Describe an algorithm to compute the adjacency matrix of G^2 using a single call to MatrixMultiply and $O(n^2)$ additional time.

(b) Suppose we discover that G^2 is a complete graph. Describe an algorithm to compute the matrix D of shortest path distances in $O(n^2)$ additional time.

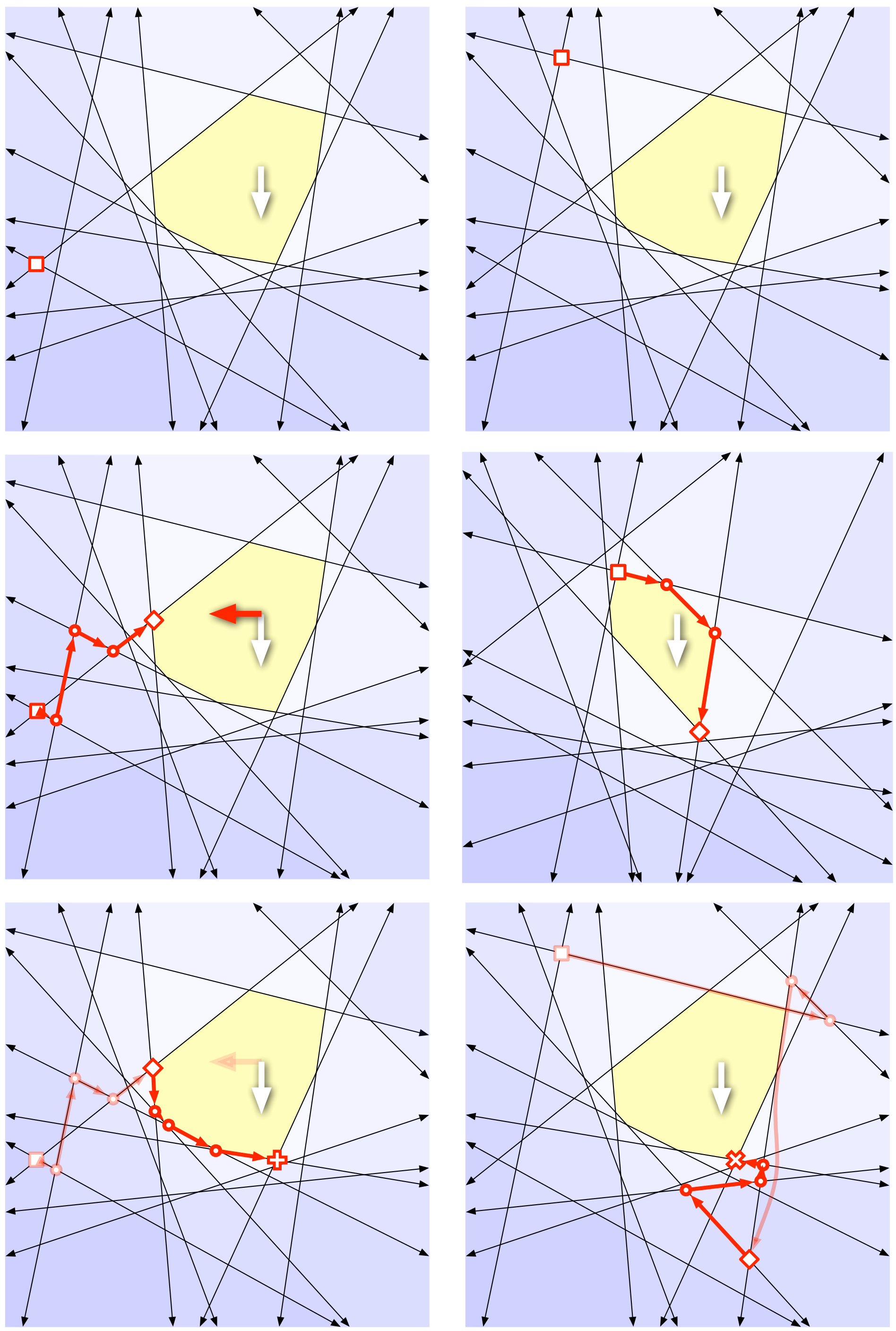
(c) Let D^2 denote the (recursively computed) matrix of shortest-path distances in G^2. Prove that the shortest-path distance from node i to node j is either $2 \cdot D^2[i, j]$ or $2 \cdot D^2[i, j] - 1$.

(d) Suppose G^2 is not a complete graph. Let $X = D^2 \cdot A$, and let $\deg(i)$ denote the degree of vertex i in the original graph G. Prove that the shortest-path distance from node i to node j is $2 \cdot D^2[i, j]$ if and only if $X[i, j] \geq D^2[i, j] \cdot \deg(i)$.

(e) Describe an algorithm to compute the matrix of shortest-path distances in G in $O(n^\omega \log n)$ time.

[5]The matrix multiplication algorithm you already know runs in $\Theta(n^3)$ time, but this is not the fastest algorithm known. The current record is $\omega \approx 2.3727$, due to Virginia Vassilevska Williams. Determining the smallest possible value of ω is a long-standing open problem; many people believe there is an undiscovered $O(n^2)$-time algorithm for matrix multiplication.

Optimization

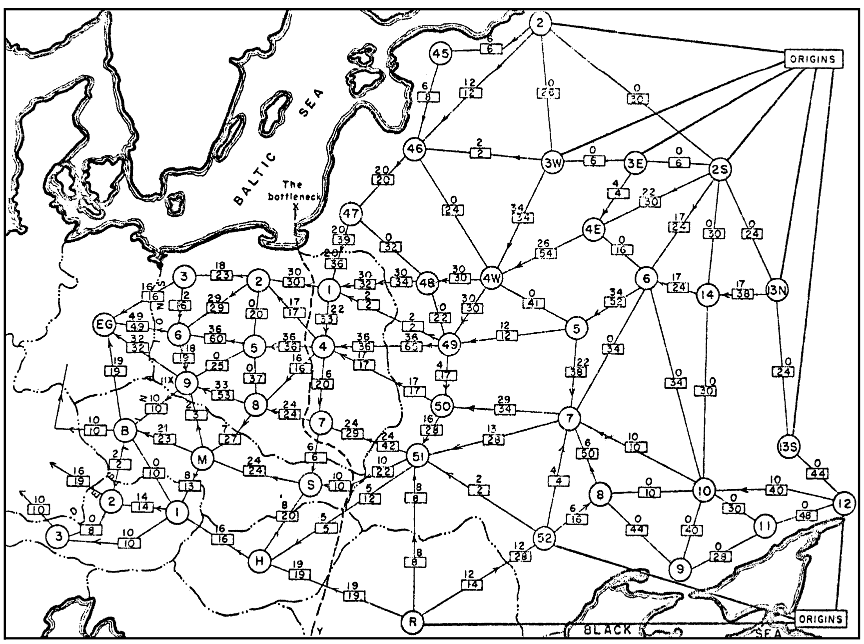


A process cannot be understood by stopping it. Understanding must move with the flow of the process, must join it and flow with it.
— The First Law of Mentat, in Frank Herbert's *Dune* (1965)

There's a difference between knowing the path and walking the path.
— Morpheus [Laurence Fishburne], *The Matrix* (1999)

23 Maximum Flows and Minimum Cuts

In the mid-1950s, Air Force researcher Theodore E. Harris and retired army general Frank S. Ross published a classified report studying the rail network that linked the Soviet Union to its satellite countries in Eastern Europe. The network was modeled as a graph with 44 vertices, representing geographic regions, and 105 edges, representing links between those regions in the rail network. Each edge was given a weight, representing the rate at which material could be shipped from one region to the next. Essentially by trial and error, they determined both the maximum amount of stuff that could be moved from Russia into Europe, as well as the cheapest way to disrupt the network by removing links (or in less abstract terms, blowing up train tracks), which they called 'the bottleneck'. Their results, including the drawing of the network below, were only declassified in 1999.[1]



Harris and Ross's map of the Warsaw Pact rail network

This one of the first recorded applications of the *maximum flow* and *minimum cut* problems. For both problems, the input is a directed graph $G = (V, E)$, along with special vertices s and t called the *source* and *target*. As in the previous lectures, I will use $u \rightarrow v$ to denote the directed edge from vertex u to vertex v. Intuitively, the maximum flow problem asks for the largest

[1]Both the map and the story were taken from Alexander Schrijver's fascinating survey 'On the history of combinatorial optimization (till 1960)'.

amount of material that can be transported from s to t; the minimum cut problem asks for the minimum damage needed to separate s from t.

23.1 Flows

An ***(s,t)-flow*** (or just a ***flow*** if the source and target are clear from context) is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies the following ***conservation constraint*** at every vertex v except possibly s and t:

$$\sum_u f(u \to v) = \sum_w f(v \to w).$$

In English, the total flow into v is equal to the total flow out of v. To keep the notation simple, we define $f(u \to v) = 0$ if there is no edge $u \to v$ in the graph. The ***value*** of the flow f, denoted $|f|$, is the total net flow out of the source vertex s:

$$|f| := \sum_w f(s \to w) - \sum_u f(u \to s).$$

It's not hard to prove that $|f|$ is also equal to the total net flow *into* the target vertex t, as follows. To simplify notation, let $\partial f(v)$ denote the total net flow out of any vertex v:

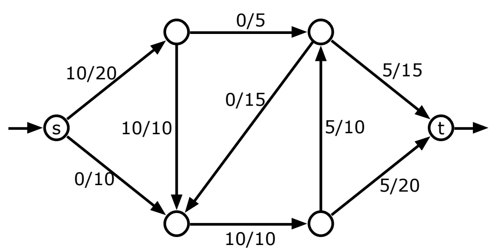
$$\partial f(v) := \sum_u f(u \to v) - \sum_w f(v \to w).$$

The conservation constraint implies that $\partial f(v) = 0$ or every vertex v except s and t, so

$$\sum_v \partial f(v) = \partial f(s) + \partial f(t).$$

On the other hand, any flow that leaves one vertex must enter another vertex, so we must have $\sum_v \partial f(v) = 0$. It follows immediately that $|f| = \partial f(s) = -\partial f(t)$.

Now suppose we have another function $c : E \to \mathbb{R}_{\geq 0}$ that assigns a non-negative ***capacity*** $c(e)$ to each edge e. We say that a flow f is ***feasible*** (with respect to c) if $f(e) \leq c(e)$ for every edge e. Most of the time we will consider only flows that are feasible with respect to some fixed capacity function c. We say that a flow f ***saturates*** edge e if $f(e) = c(e)$, and ***avoids*** edge e if $f(e) = 0$. The ***maximum flow problem*** is to compute a feasible (s,t)-flow in a given directed graph, with a given capacity function, whose value is as large as possible.



An (s,t)-flow with value 10. Each edge is labeled with its flow/capacity.

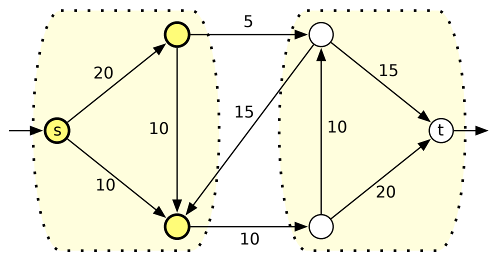
23.2 Cuts

An ***(s,t)-cut*** (or just ***cut*** if the source and target are clear from context) is a partition of the vertices into disjoint subsets S and T—meaning $S \cup T = V$ and $S \cap T = \varnothing$—where $s \in S$ and $t \in T$.

If we have a capacity function $c\colon E \to \mathbb{R}_{\geq 0}$, the ***capacity*** of a cut is the sum of the capacities of the edges that start in S and end in T:

$$\|S, T\| := \sum_{v \in S} \sum_{w \in T} c(v \to w).$$

(Again, if $v \to w$ is not an edge in the graph, we assume $c(v \to w) = 0$.) Notice that the definition is asymmetric; edges that start in T and end in S are unimportant. The ***minimum cut problem*** is to compute an (s,t)-cut whose capacity is as large as possible.



An (s,t)-cut with capacity 15. Each edge is labeled with its capacity.

Intuitively, the minimum cut is the cheapest way to disrupt all flow from s to t. Indeed, it is not hard to show that **the value of *any* feasible (s,t)-flow is at most the capacity of *any* (s,t)-cut.** Choose your favorite flow f and your favorite cut (S,T), and then follow the bouncing inequalities:

$$
\begin{aligned}
|f| &= \sum_w f(s \to w) - \sum_u f(u \to s) && \text{by definition} \\
&= \sum_{v \in S} \left(\sum_w f(v \to w) - \sum_u f(u \to v) \right) && \text{by the conservation constraint} \\
&= \sum_{v \in S} \left(\sum_{w \in T} f(v \to w) - \sum_{u \in T} f(u \to v) \right) && \text{removing duplicate edges} \\
&\leq \sum_{v \in S} \sum_{w \in T} f(v \to w) && \text{since } f(u \to v) \geq 0 \\
&\leq \sum_{v \in S} \sum_{w \in T} c(v \to w) && \text{since } f(u \to v) \leq c(v \to w) \\
&= \|S, T\| && \text{by definition}
\end{aligned}
$$

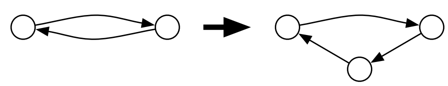
Our derivation actually implies the following stronger observation: **$|f| = \|S,T\|$ if and only if f saturates every edge from S to T and avoids every edge from T to S.** Moreover, if we have a flow f and a cut (S,T) that satisfies this equality condition, f must be a maximum flow, and (S,T) must be a minimum cut.

23.3 The Maxflow Mincut Theorem

Surprisingly, for any weighted directed graph, there is always a flow f and a cut (S,T) that satisfy the equality condition. This is the famous *max-flow min-cut theorem*, first proved by Lester Ford (of shortest path fame) and Delbert Ferguson in 1954 and independently by Peter Elias, Amiel Feinstein, and and Claude Shannon (of information theory fame) in 1956.

The Maxflow Mincut Theorem. *In any flow network with source s and target t, the value of the maximum (s,t)-flow is* equal *to the capacity of the minimum (s,t)-cut.*

Ford and Fulkerson proved this theorem as follows. Fix a graph G, vertices s and t, and a capacity function $c\colon E \to \mathbb{R}_{\geq 0}$. The proof will be easier if we assume that the capacity function is ***reduced***: For any vertices u and v, either $c(u{\to}v) = 0$ or $c(v{\to}u) = 0$, or equivalently, if an edge appears in G, then its reversal does not. This assumption is easy to enforce. Whenever an edge $u{\to}v$ and its reversal $v{\to}u$ are both the graph, replace the edge $u{\to}v$ with a path $u{\to}x{\to}v$ of length two, where x is a new vertex and $c(u{\to}x) = c(x{\to}v) = c(u{\to}v)$. The modified graph has the same maximum flow value and minimum cut capacity as the original graph.

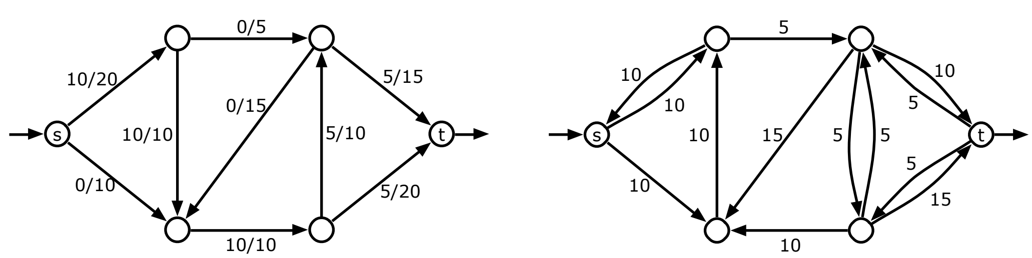


Enforcing the one-direction assumption.

Let f be a feasible flow. We define a new capacity function $c_f : V \times V \to \mathbb{R}$, called the ***residual capacity***, as follows:

$$c_f(u{\to}v) = \begin{cases} c(u{\to}v) - f(u{\to}v) & \text{if } u{\to}v \in E \\ f(v{\to}u) & \text{if } v{\to}u \in E \\ 0 & \text{otherwise} \end{cases}.$$

Since $f \geq 0$ and $f \leq c$, the residual capacities are always non-negative. It is possible to have $c_f(u{\to}v) > 0$ even if $u{\to}v$ is not an edge in the original graph G. Thus, we define the ***residual graph*** $G_f = (V, E_f)$, where E_f is the set of edges whose residual capacity is positive. Notice that the residual capacities are *not* necessarily reduced; it is quite possible to have both $c_f(u{\to}v) > 0$ and $c_f(v{\to}u) > 0$.



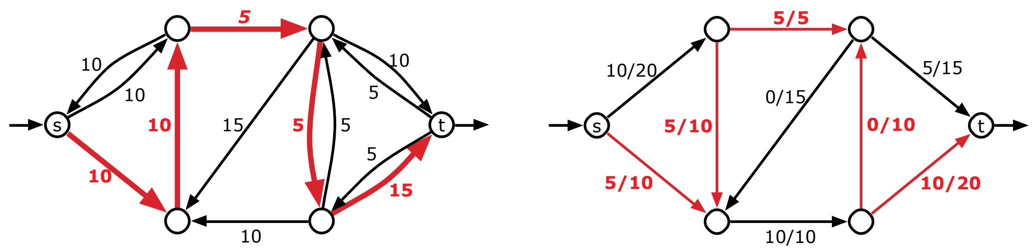
A flow f in a weighted graph G and the corresponding residual graph G_f.

Suppose there is no path from the source s to the target t in the residual graph G_f. Let S be the set of vertices that are reachable from s in G_f, and let $T = V \setminus S$. The partition (S,T) is clearly an (s,t)-cut. For every vertex $u \in S$ and $v \in T$, we have

$$c_f(u{\to}v) = (c(u{\to}v) - f(u{\to}v)) + f(v{\to}u) = 0,$$

which implies that $c(u{\to}v) - f(u{\to}v) = 0$ and $f(v{\to}u) = 0$. In other words, our flow f saturates every edge from S to T and avoids every edge from T to S. It follows that $|f| = \|S,T\|$. Moreover, f is a maximum flow and (S,T) is a minimum cut.

On the other hand, suppose there is a path $s = v_0{\to}v_1{\to}\cdots{\to}v_r = t$ in G_f. We refer to $v_0{\to}v_1{\to}\cdots{\to}v_r$ as an ***augmenting path***. Let $F = \min_i c_f(v_i{\to}v_{i+1})$ denote the maximum amount



An augmenting path in G_f with value $F = 5$ and the augmented flow f'.

of flow that we can push through the augmenting path in G_f. We define a new flow function $f' : E \to \mathbb{R}$ as follows:

$$f'(u\to v) = \begin{cases} f(u\to v) + F & \text{if } u\to v \text{ is in the augmenting path} \\ f(u\to v) - F & \text{if } v\to u \text{ is in the augmenting path} \\ f(u\to v) & \text{otherwise} \end{cases}$$

To prove that the flow f' is feasible with respect to the original capacities c, we need to verify that $f' \geq 0$ and $f' \leq c$. Consider an edge $u\to v$ in G. If $u\to v$ is in the augmenting path, then $f'(u\to v) > f(u\to v) \geq 0$ and

$$\begin{aligned} f'(u\to v) &= f(u\to v) + F && \text{by definition of } f' \\ &\leq f(u\to v) + c_f(u\to v) && \text{by definition of } F \\ &= f(u\to v) + c(u\to v) - f(u\to v) && \text{by definition of } c_f \\ &= c(u\to v) && \text{Duh.} \end{aligned}$$

On the other hand, if the reversal $v\to u$ is in the augmenting path, then $f'(u\to v) < f(u\to v) \leq c(u\to v)$, which implies that

$$\begin{aligned} f'(u\to v) &= f(u\to v) - F && \text{by definition of } f' \\ &\geq f(u\to v) - c_f(v\to u) && \text{by definition of } F \\ &= f(u\to v) - f(u\to v) && \text{by definition of } c_f \\ &= 0 && \text{Duh.} \end{aligned}$$

Finally, we observe that (without loss of generality) only the first edge in the augmenting path leaves s, so $|f'| = |f| + F > 0$. In other words, f is *not* a maximum flow.

This completes the proof!

23.4 Ford and Fulkerson's augmenting-path algorithm

Ford and Fulkerson's proof of the Maxflow-Mincut Theorem translates immediately to an algorithm to compute maximum flows: Starting with the zero flow, repeatedly augment the flow along **any** path from s to t in the residual graph, until there is no such path.

This algorithm has an important but straightforward corollary:

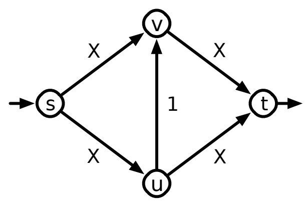
Integrality Theorem. *If all capacities in a flow network are integers, then there is a maximum flow such that the flow through every edge is an integer.*

Proof: We argue by induction that after each iteration of the augmenting path algorithm, all flow values and residual capacities are integers. Before the first iteration, residual capacities are the original capacities, which are integral by definition. In each later iteration, the induction hypothesis implies that the capacity of the augmenting path is an integer, so augmenting changes the flow on each edge, and therefore the residual capacity of each edge, by an integer.

In particular, the algorithm increases the overall value of the flow by a positive integer, which implies that the augmenting path algorithm halts and returns a maximum flow. □

If every edge capacity is an integer, the algorithm halts after $|f^*|$ iterations, where f^* is the actual maximum flow. In each iteration, we can build the residual graph G_f and perform a whatever-first-search to find an augmenting path in $O(E)$ time. Thus, for networks with integer capacities, the Ford-Fulkerson algorithm runs in $\boldsymbol{O(E|f^*|)}$ time in the worst case.

The following example shows that this running time analysis is essentially tight. Consider the 4-node network illustrated below, where X is some large integer. The maximum flow in this network is clearly $2X$. However, Ford-Fulkerson might alternate between pushing 1 unit of flow along the augmenting path $s \to u \to v \to t$ and then pushing 1 unit of flow along the augmenting path $s \to v \to u \to t$, leading to a running time of $\Theta(X) = \Omega(|f^*|)$.



A bad example for the Ford-Fulkerson algorithm.

Ford and Fulkerson's algorithm works quite well in many practical situations, or in settings where the maximum flow value $|f^*|$ is small, but without further constraints on the augmenting paths, this is *not* an efficient algorithm in general. The example network above can be described using only $O(\log X)$ bits; thus, the running time of Ford-Fulkerson is actually *exponential* in the input size.

23.5 Irrational Capacities

If we multiply all the capacities by the same (positive) constant, the maximum flow increases everywhere by the same constant factor. It follows that if all the edge capacities are *rational*, then the Ford-Fulkerson algorithm eventually halts, although still in exponential time.

However, if we allow *irrational* capacities, the algorithm can actually loop forever, always finding smaller and smaller augmenting paths! Worse yet, this infinite sequence of augmentations may not even converge to the maximum flow, or even to a significant fraction of the maximum flow! Perhaps the simplest example of this effect was discovered by Uri Zwick.

Consider the six-node network shown on the next page. Six of the nine edges have some large integer capacity X, two have capacity 1, and one has capacity $\phi = (\sqrt{5}-1)/2 \approx 0.618034$, chosen so that $1-\phi = \phi^2$. To prove that the Ford-Fulkerson algorithm can get stuck, we can watch the residual capacities of the three horizontal edges as the algorithm progresses. (The residual capacities of the other six edges will always be at least $X-3$.)

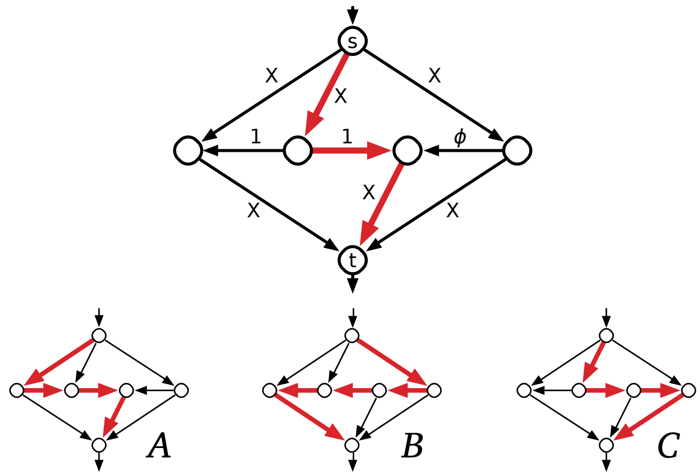
Suppose the Ford-Fulkerson algorithm starts by choosing the central augmenting path, shown in the large figure on the next page. The three horizontal edges, in order from left to right, now have residual capacities 1, 0, and ϕ. Suppose inductively that the horizontal residual capacities are ϕ^{k-1}, 0, ϕ^k for some non-negative integer k.

1. Augment along B, adding ϕ^k to the flow; the residual capacities are now $\phi^{k+1}, \phi^k, 0$.
2. Augment along C, adding ϕ^k to the flow; the residual capacities are now $\phi^{k+1}, 0, \phi^k$.
3. Augment along B, adding ϕ^{k+1} to the flow; the residual capacities are now $0, \phi^{k+1}, \phi^{k+2}$.
4. Augment along A, adding ϕ^{k+1} to the flow; the residual capacities are now $\phi^{k+1}, 0, \phi^{k+2}$.

It follows by induction that after $4n+1$ augmentation steps, the horizontal edges have residual capacities $\phi^{2n-2}, 0, \phi^{2n-1}$. As the number of augmentations grows to infinity, the value of the flow converges to

$$1 + 2\sum_{i=1}^{\infty} \phi^i = 1 + \frac{2}{1-\phi} = 4 + \sqrt{5} < 7,$$

even though the maximum flow value is clearly $2X + 1 \gg 7$.



Uri Zwick's non-terminating flow example, and three augmenting paths.

Picky students might wonder at this point why we care about irrational capacities; after all, computers can't represent anything but (small) integers or (dyadic) rationals exactly. Good question! One reason is that the integer restriction is literally *artificial*; it's an *artifact* of actual computational hardware[2], not an inherent feature of the abstract mathematical problem. Another reason, which is probably more convincing to most practical computer scientists, is that the behavior of the algorithm with irrational inputs tells us something about its worst-case behavior *in practice* given floating-point capacities—terrible! Even with very reasonable capacities, a careless implementation of Ford-Fulkerson could enter an infinite loop simply because of round-off error.

23.6 Edmonds and Karp's Algorithms

Ford and Fulkerson's algorithm does not specify which path in the residual graph to augment, and the poor behavior of the algorithm can be blamed on poor choices for the augmenting path. In the early 1970s, Jack Edmonds and Richard Karp analyzed two natural rules for choosing augmenting paths, both of which led to more efficient algorithms.

[2]...or perhaps the laws of physics. Yeah, whatever. Like *reality* actually matters in this class.

23.6.1 Fat Pipes

Edmonds and Karp's first rule is essentially a greedy algorithm:

Choose the augmenting path with largest bottleneck value.

It's a fairly easy to show that the maximum-bottleneck (s,t)-path in a directed graph can be computed in $O(E \log V)$ time using a variant of Jarník's minimum-spanning-tree algorithm, or of Dijkstra's shortest path algorithm. Simply grow a directed spanning tree T, rooted at s. Repeatedly find the highest-capacity edge leaving T and add it to T, until T contains a path from s to t. Alternately, one could emulate Kruskal's algorithm—insert edges one at a time in decreasing capacity order until there is a path from s to t—although this is less efficient, at least when the graph is directed.

We can now analyze the algorithm in terms of the value of the maximum flow f^*. Let f be any flow in G, and let f' be the maximum flow *in the current residual graph* G_f. (At the beginning of the algorithm, $G_f = G$ and $f' = f^*$.) Let e be the bottleneck edge in the next augmenting path. Let S be the set of vertices reachable from s through edges in G_f with capacity greater than $c_f(e)$ and let $T = V \setminus S$. By construction, T is non-empty, and every edge from S to T has capacity at most $c_f(e)$. Thus, the capacity of the cut (S,T) is at most $c_f(e) \cdot E$. On the other hand, the maxflow-mincut theorem implies that $\|S,T\| \geq |f'|$. We conclude that $c(e) \geq |f'|/E$.

The preceding argument implies that augmenting f along the maximum-bottleneck path in G_f multiplies the maximum flow value in G_f by a factor of at most $1 - 1/E$. In other words, the residual maximum flow value *decays exponentially* with the number of iterations. After $E \cdot \ln|f^*|$ iterations, the maximum flow value in G_f is at most

$$|f^*| \cdot (1 - 1/E)^{E \cdot \ln|f^*|} < |f^*| \, e^{-\ln|f^*|} = 1.$$

(That's Euler's constant e, not the edge e. Sorry.) In particular, *if all the capacities are integers*, then after $E \cdot \ln|f^*|$ iterations, the maximum capacity of the residual graph is *zero* and f is a maximum flow.

We conclude that for graphs with integer capacities, the Edmonds-Karp 'fat pipe' algorithm runs in $\boldsymbol{O(E^2 \log E \log|f^*|)}$ time, which is actually a polynomial function of the input size.

23.6.2 Short Pipes

The second Edmonds-Karp rule was actually proposed by Ford and Fulkerson in their original max-flow paper; a variant of this rule was independently considered by the Russian mathematician Yefim Dinits around the same time as Edmonds and Karp.

Choose the augmenting path with the smallest number of edges.

The shortest augmenting path can be found in $O(E)$ time by running breadth-first search in the residual graph. Surprisingly, the resulting algorithm halts after a polynomial number of iterations, independent of the actual edge capacities!

The proof of this polynomial upper bound relies on two observations about the evolution of the residual graph. Let f_i be the current flow after i augmentation steps, let G_i be the corresponding residual graph. In particular, f_0 is zero everywhere and $G_0 = G$. For each vertex v, let $\mathit{level}_i(v)$ denote the unweighted shortest path distance from s to v in G_i, or equivalently, the *level* of v in a breadth-first search tree of G_i rooted at s.

Our first observation is that these levels can only increase over time.

Lemma 1. $\text{level}_{i+1}(v) \geq \text{level}_i(v)$ *for all vertices v and integers i.*

Proof: The claim is trivial for $v = s$, since $level_i(s) = 0$ for all i. Choose an arbitrary vertex $v \neq s$, and let $s \to \cdots \to u \to v$ be a shortest path from s to v in G_{i+1}. (If there is no such path, then $level_{i+1}(v) = \infty$, and we're done.) Because this is a shortest path, we have $level_{i+1}(v) = level_{i+1}(u) + 1$, and the inductive hypothesis implies that $level_{i+1}(u) \geq level_i(u)$.

We now have two cases to consider. If $u \to v$ is an edge in G_i, then $level_i(v) \leq level_i(u) + 1$, because the levels are defined by breadth-first traversal.

On the other hand, if $u \to v$ is not an edge in G_i, then $v \to u$ must be an edge in the ith augmenting path. Thus, $v \to u$ must lie on the shortest path from s to t in G_i, which implies that $level_i(v) = level_i(u) - 1 \leq level_i(u) + 1$.

In both cases, we have $level_{i+1}(v) = level_{i+1}(u) + 1 \geq level_i(u) + 1 \geq level_i(v)$. □

Whenever we augment the flow, the bottleneck edge in the augmenting path disappears from the residual graph, and some other edge in the *reversal* of the augmenting path may (re-)appear. Our second observation is that an edge cannot appear or disappear too many times.

Lemma 2. *During the execution of the Edmonds-Karp short-pipe algorithm, any edge $u \to v$ disappears from the residual graph G_f at most $V/2$ times.*

Proof: Suppose $u \to v$ is in two residual graphs G_i and G_{j+1}, but not in any of the intermediate residual graphs $G_{i+1}, \ldots, G_j$, for some $i < j$. Then $u \to v$ must be in the ith augmenting path, so $level_i(v) = level_i(u) + 1$, and $v \to u$ must be on the jth augmenting path, so $level_j(v) = level_j(u) - 1$. By the previous lemma, we have

$$level_j(u) = level_j(v) + 1 \geq level_i(v) + 1 = level_i(u) + 2.$$

In other words, the distance from s to u increased by at least 2 between the disappearance and reappearance of $u \to v$. Since every level is either less than V or infinite, the number of disappearances is at most $V/2$. □

Now we can derive an upper bound on the number of iterations. Since each edge can disappear at most $V/2$ times, there are at most $EV/2$ edge disappearances overall. But at least one edge disappears on each iteration, so the algorithm must halt after at most $EV/2$ iterations. Finally, since each iteration requires $O(E)$ time, this algorithm runs in ***$O(VE^2)$ time*** overall.

23.7 Further Progress

This is nowhere near the end of the story for maximum-flow algorithms. Decades of further research have led to a number of even faster algorithms, some of which are summarized in the table below.[3] All of the algorithms listed below compute a maximum flow in several iterations. Each algorithm has two variants: a simpler version that performs each iteration by brute force, and a faster variant that uses sophisticated data structures to maintain a spanning tree of the flow network, so that each iteration can be performed (and the spanning tree updated) in logarithmic time. There is no reason to believe that the best algorithms known so far are optimal; indeed, maximum flows are still a very active area of research.

[3] To keep the table short, I have deliberately omitted algorithms whose running time depends on the maximum capacity, the sum of the capacities, or the maximum flow value. Even with this restriction, the table is incomplete!

Technique	Direct	With dynamic trees	Sources
Blocking flow	$O(V^2E)$	$O(VE\log V)$	[Dinits; Sleator and Tarjan]
Network simplex	$O(V^2E)$	$O(VE\log V)$	[Dantzig; Goldfarb and Hao; Goldberg, Grigoriadis, and Tarjan]
Push-relabel (generic)	$O(V^2E)$	—	[Goldberg and Tarjan]
Push-relabel (FIFO)	$O(V^3)$	$O(V^2\log(V^2/E))$	[Goldberg and Tarjan]
Push-relabel (highest label)	$O(V^2\sqrt{E})$	—	[Cheriyan and Maheshwari; Tunçel]
Pseudoflow	$O(V^2E)$	$O(VE\log V)$	[Hochbaum]
Compact abundance graphs		$O(VE)$	[Orlin 2012]

Several purely combinatorial maximum-flow algorithms and their running times.

The fastest known maximum flow algorithm, announced by James Orlin in 2012, runs in ***$O(VE)$ time***. The details of Orlin's algorithm are far beyond the scope of this course; in addition to his own new techniques, Orlin uses several existing algorithms and data structures as black boxes, most of which are themselves quite complicated. Nevertheless, for purposes of analyzing algorithms that use maximum flows, this is the time bound you should cite. So write the following sentence on your cheat sheets and cite it in your homeworks:

Maximum flows can be computed in $O(VE)$ time.

Exercises

1. Suppose you are given a directed graph $G = (V, E)$, two vertices s and t, a capacity function $c : E \to \mathbb{R}^+$, and a second function $f : E \to \mathbb{R}$. Describe an algorithm to determine whether f is a maximum (s,t)-flow in G.

2. Let (S, T) and (S', T') be minimum (s,t)-cuts in some flow network G. Prove that $(S \cap S', T \cup T')$ and $(S \cup S', T \cap T')$ are also minimum (s,t)-cuts in G.

3. Suppose (S, T) is the *unique* minimum (s,t)-cut in some flow network. Prove that (S, T) is also a minimum (x, y)-cut for all vertices $x \in S$ and $y \in T$.

4. Cuts are sometimes defined as subsets of the edges of the graph, instead of as partitions of its vertices. In this problem, you will prove that these two definitions are *almost* equivalent.

   We say that a subset X of (directed) edges *separates s and t* if every directed path from s to t contains at least one (directed) edge in X. For any subset S of *vertices*, let δS denote the set of directed edges leaving S; that is, $\delta S := \{u{\to}v \mid u \in S, v \notin S\}$.

   (a) Prove that if (S, T) is an (s,t)-cut, then δS separates s and t.

   (b) Let X be an arbitrary subset of edges that separates s and t. Prove that there is an (s,t)-cut (S, T) such that $\delta S \subseteq X$.

   (c) Let X be a *minimal* subset of edges that separates s and t. (Such a set of edges is sometimes called a ***bond***.) Prove that there is an (s,t)-cut (S, T) such that $\delta S = X$.

5. A flow f is ***acyclic*** if the subgraph of directed edges with positive flow contains no directed cycles.

   (a) Prove that for any flow f, there is an acyclic flow with the same value as f. (In particular, this implies that some maximum flow is acyclic.)

   (b) A *path flow* assigns positive values only to the edges of one simple directed path from s to t. Prove that every acyclic flow can be written as the sum of $O(E)$ path flows.

   (c) Describe a flow in a directed graph that *cannot* be written as the sum of path flows.

   (d) A *cycle flow* assigns positive values only to the edges of one simple directed cycle. Prove that every flow can be written as the sum of $O(E)$ path flows and cycle flows.

   (e) Prove that every flow with value 0 can be written as the sum of $O(E)$ cycle flows. (Zero-value flows are also called *circulations*.)

6. Suppose instead of capacities, we consider networks where each edge $u{\to}v$ has a non-negative ***demand*** $d(u{\to}v)$. Now an (s,t)-flow f is *feasible* if and only if $f(u{\to}v) \geq d(u{\to}v)$ for every edge $u{\to}v$. (Feasible flow values can now be arbitrarily large.) A natural problem in this setting is to find a feasible (s,t)-flow of *minimum* value.

   (a) Describe an efficient algorithm to compute a feasible (s,t)-flow, given the graph, the demand function, and the vertices s and t as input. *[Hint: Find a flow that is non-zero everywhere, and then scale it up to make it feasible.]*

   (b) Suppose you have access to a subroutine MaxFlow that computes *maximum* flows in networks with edge capacities. Describe an efficient algorithm to compute a *minimum* flow in a given network with edge demands; your algorithm should call MaxFlow exactly once.

   (c) State and prove an analogue of the max-flow min-cut theorem for this setting. (Do minimum flows correspond to maximum cuts?)

7. For any flow network G and any vertices u and v, let $bottleneck_G(u,v)$ denote the maximum, over all paths π in G from u to v, of the minimum-capacity edge along π.

   (a) Describe and analyze an algorithm to compute $bottleneck_G(s,t)$ in $O(E \log V)$ time.

   (b) Describe an algorithm to construct a spanning tree T of G such that $bottleneck_T(u,v) = bottleneck_G(u,v)$ for *all* vertices u and v. (Edges in T inherit their capacities from G.)

8. Describe an efficient algorithm to determine whether a given flow network contains a *unique* maximum flow.

9. Suppose you have already computed a maximum flow f^* in a flow network G with ***integer*** edge capacities.

   (a) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is increased by 1.

   (b) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is decreased by 1.

Both algorithms should be significantly faster than recomputing the maximum flow from scratch.

10. Let G be a network with integer edge capacities. An edge in G is *upper-binding* if increasing its capacity by 1 also increases the value of the maximum flow in G. Similarly, an edge is *lower-binding* if decreasing its capacity by 1 also decreases the value of the maximum flow in G.

    (a) Does every network G have at least one upper-binding edge? Prove your answer is correct.

    (b) Does every network G have at least one lower-binding edge? Prove your answer is correct.

    (c) Describe an algorithm to find all upper-binding edges in G, given both G and a maximum flow in G as input, in $O(E)$ time.

    (d) Describe an algorithm to find all lower-binding edges in G, given both G and a maximum flow in G as input, in $O(EV)$ time.

11. A given flow network G may have more than one minimum (s,t)-cut. Let's define the ***best*** minimum (s,t)-cut to be any minimum cut with the smallest number of edges.

    (a) Describe an efficient algorithm to determine whether a given flow network contains a *unique* minimum (s,t)-cut.

    (b) Describe an efficient algorithm to find the best minimum (s,t)-cut when the capacities are integers.

    (c) Describe an efficient algorithm to find the best minimum (s,t)-cut for *arbitrary* edge capacities.

    (d) Describe an efficient algorithm to determine whether a given flow network contains a unique *best* minimum (s,t)-cut.

12. A new assistant professor, teaching maximum flows for the first time, suggests the following greedy modification to the generic Ford-Fulkerson augmenting path algorithm. Instead of maintaining a residual graph, just reduce the capacity of edges along the augmenting path! In particular, whenever we saturate an edge, just remove it from the graph.

```
GreedyFlow(G, c, s, t):
  for every edge e in G
      f(e) ← 0

  while there is a path from s to t
      π ← an arbitrary path from s to t
      F ← minimum capacity of any edge in π
      for every edge e in π
          f(e) ← f(e) + F
          if c(e) = F
              remove e from G
          else
              c(e) ← c(e) − F

  return f
```

(a) Show that this algorithm does *not* always compute a maximum flow.

(b) Prove that for any flow network, if the Greedy Path Fairy tells you precisely which path π to use at each iteration, then GreedyFlow does compute a maximum flow. (Sadly, the Greedy Path Fairy does not actually exist.)

13. We can speed up the Edmonds-Karp 'fat pipe' heuristic, at least for integer capacities, by relaxing our requirements for the next augmenting path. Instead of finding the augmenting path with maximum bottleneck capacity, we find a path whose bottleneck capacity is at least half of maximum, using the following ***capacity scaling*** algorithm.

    The algorithm maintains a bottleneck threshold Δ; initially, Δ is the maximum capacity among all edges in the graph. In each *phase*, the algorithm augments along paths from s to t in which every edge has residual capacity at least Δ. When there is no such path, the phase ends, we set $\Delta \leftarrow \lfloor \Delta/2 \rfloor$, and the next phase begins.

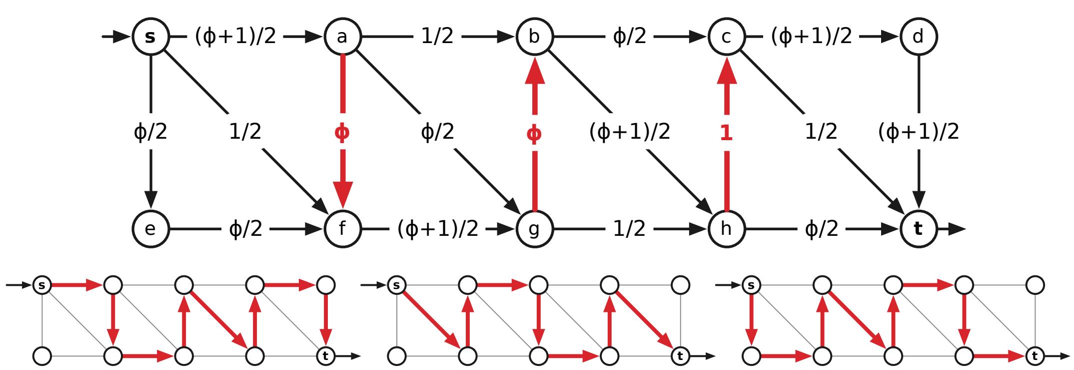
    (a) How many phases will the algorithm execute in the worst case, if the edge capacities are integers?

    (b) Let f be the flow at the end of a phase for a particular value of Δ. Let S be the nodes that are reachable from s in the residual graph G_f using only edges with residual capacity at least Δ, and let $T = V \setminus S$. Prove that the capacity (with respect to G's original edge capacities) of the cut (S, T) is at most $|f| + E \cdot \Delta$.

    (c) Prove that in each phase of the scaling algorithm, there are at most $2E$ augmentations.

    (d) What is the overall running time of the scaling algorithm, assuming all the edge capacities are integers?

14. In 1980 Maurice Queyranne published the following example of a flow network where Edmonds and Karp's "fat pipe" heuristic does not halt. Here, as in Zwick's bad example for the original Ford-Fulkerson algorithm, ϕ denotes the inverse golden ratio $(\sqrt{5}-1)/2$. The three vertical edges play essentially the same role as the horizontal edges in Zwick's example.



Queyranne's network, and a sequence of "fat-pipe" augmentations.

    (a) Show that the following infinite sequence of path augmentations is a valid execution of the Edmonds-Karp algorithm. (See the figure above.)

```
QueyranneFatPipes:
  for i ← 1 to ∞
      push φ^(3i−2) units of flow along  s→a→f→g→b→h→c→d→t
      push φ^(3i−1) units of flow along  s→f→a→b→g→h→c→t
      push φ^(3i)   units of flow along  s→e→f→a→g→b→c→h→t
  forever
```

(b) Describe a sequence of $O(1)$ path augmentations that yields a maximum flow in Queyranne's network.

15. An ***(s, t)-series-parallel*** graph is an directed acyclic graph with two designated vertices s (the *source*) and t (the *target* or *sink*) and with one of the following structures:

- **Base case:** A single directed edge from s to t.
- **Series:** The union of an (s, u)-series-parallel graph and a (u, t)-series-parallel graph that share a common vertex u but no other vertices or edges.
- **Parallel:** The union of two smaller (s, t)-series-parallel graphs with the same source s and target t, but with no other vertices or edges in common.

Describe an efficient algorithm to compute a maximum flow from s to t in an (s, t)-series-parallel graph with arbitrary edge capacities.

> *For a long time it puzzled me how something so expensive, so leading edge, could be so useless, and then it occurred to me that a computer is a stupid machine with the ability to do incredibly smart things, while computer programmers are smart people with the ability to do incredibly stupid things. They are, in short, a perfect match.*
>
> — Bill Bryson, *Notes from a Big Country* (1999)

24 Applications of Maximum Flow

24.1 Edge-Disjoint Paths

One of the easiest applications of maximum flows is computing the maximum number of edge-disjoint paths between two specified vertices s and t in a directed graph G using maximum flows. A set of paths in G is *edge-disjoint* if each edge in G appears in at most one of the paths; several edge-disjoint paths may pass through the same vertex, however.

If we give each edge capacity 1, then the maxflow from s to t assigns a flow of either 0 or 1 to every edge. Since any vertex of G lies on at most two saturated edges (one in and one out, or none at all), the subgraph S of saturated edges is the union of several edge-disjoint paths and cycles. Moreover, the number of paths is exactly equal to the value of the flow. Extracting the actual paths from S is easy—just follow any directed path in S from s to t, remove that path from S, and recurse.

Conversely, we can transform any collection of k edge-disjoint paths into a flow by pushing one unit of flow along each path from s to t; the value of the resulting flow is exactly k. It follows that any maxflow algorithm actually computes the largest possible set of edge-disjoint paths.

If we use Orlin's algorithm to compute the maximum (s,t)-flow, we can compute edge-disjoint paths in $\boldsymbol{O(VE)}$ ***time***, but Orlin's algorithm is overkill for this simple application. The cut $(\{s\}, V \setminus \{s\})$ has capacity at most $V-1$, so the maximum flow has value at most $V-1$. Thus, Ford and Fulkerson's original augmenting path algorithm also runs in $O(|f^*|E) = O(VE)$ time.

The same algorithm can also be used to find edge-disjoint paths in *undirected* graphs. We simply replace every undirected edge in G with a pair of directed edges, each with unit capacity, and compute a maximum flow from s to t in the resulting directed graph G' using the Ford-Fulkerson algorithm. For any edge uv in G, if our max flow saturates both directed edges $u{\to}v$ and $v{\to}u$ in G', we can remove *both* edges from the flow without changing its value. Thus, without loss of generality, the maximum flow assigns a direction to every saturated edge, and we can extract the edge-disjoint paths by searching the graph of directed saturated edges.

24.2 Vertex Capacities and Vertex-Disjoint Paths

Suppose we have capacities on the vertices as well as the edges. Here, in addition to our other constraints, we require that for any vertex v other than s and t, the total flow into v (and therefore the total flow out of v) is at most some non-negative value $c(v)$. How can we compute a maximum flow with these new constraints?

The simplest method is to transform the input into a traditional flow network, with only edge capacities. Specifically, we replace every vertex v with two vertices v_{in} and v_{out}, connected by an edge $v_{\text{in}}{\to}v_{\text{out}}$ with capacity $c(v)$, and then replace every directed edge $u{\to}v$ with the edge $u_{\text{out}}{\to}v_{\text{in}}$ (keeping the same capacity). Finally, we compute the maximum flow from s_{out} to t_{in} in this modified flow network.

It is now easy to compute the maximum number of *vertex*-disjoint paths from s to t in any directed graph. Simply give every vertex capacity 1, and compute a maximum flow!

★★★ Figure!

24.3 Maximum Matchings in Bipartite Graphs

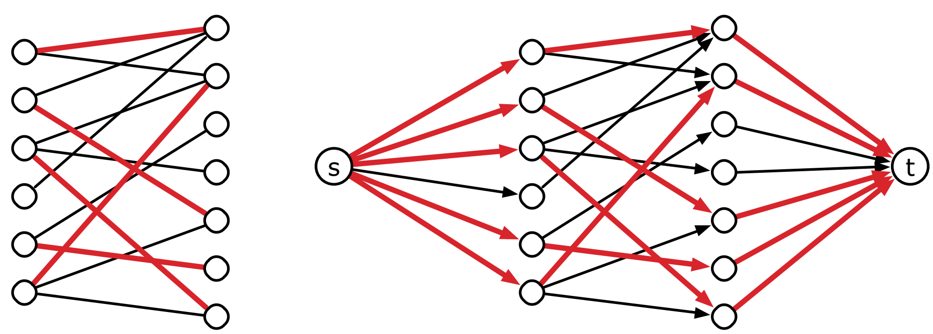
Another natural application of maximum flows is finding large *matchings* in bipartite graphs. A matching is a subgraph in which every vertex has degree at most one, or equivalently, a collection of edges such that no two share a vertex. The problem is to find the matching with the maximum number of edges in a given bipartite graph.

We can solve this problem by reducing it to a maximum flow problem as follows. Let G be the given bipartite graph with vertex set $U \cup W$, such that every edge joins a vertex in U to a vertex in W. We create a new *directed* graph G' by (1) orienting each edge from U to W, (2) adding two new vertices s and t, (3) adding edges from s to every vertex in U, and (4) adding edges from each vertex in W to t. Finally, we assign every edge in G' a capacity of 1.

Any matching M in G can be transformed into a flow f_M in G' as follows: For each edge uw in M, push one unit of flow along the path $s \to u \to w \to t$. These paths are disjoint except at s and t, so the resulting flow satisfies the capacity constraints. Moreover, the value of the resulting flow is equal to the number of edges in M.

Conversely, consider any (s,t)-flow f in G' computed using the Ford-Fulkerson augmenting path algorithm. Because the edge capacities are integers, the Ford-Fulkerson algorithm assigns an integer flow to every edge. (This is easy to verify by induction, hint, hint.) Moreover, since each edge has *unit* capacity, the computed flow either saturates $(f(e) = 1)$ or avoids $(f(e) = 0)$ every edge in G'. Finally, since at most one unit of flow can enter any vertex in U or leave any vertex in W, the saturated edges from U to W form a matching in G. The size of this matching is exactly $|f|$.

Thus, the size of the maximum matching in G is equal to the value of the maximum flow in G', and provided we compute the maxflow using augmenting paths, we can convert the actual maxflow into a maximum matching in $O(E)$ time. Again, we can compute the maximum flow in ***$O(VE)$ time*** using either Orlin's algorithm or off-the-shelf Ford-Fulkerson.



A maximum matching in a bipartite graph G, and the corresponding maximum flow in G'.

24.4 Assignment Problems

Maximum-cardinality matchings are a special case of a general family of so-called *assignment* problems.[1] An unweighted *binary* assignment problem involves two disjoint finite sets X and Y,

[1]Most authors refer to finding a maximum-weight matching in a bipartite graph as *the* assignment problem.

which typically represent two different kinds of resources, such as web pages and servers, jobs and machines, rows and columns of a matrix, hospitals and interns, or customers and ice cream flavors. Our task is to choose the largest possible collection of pairs (x, y) as possible, where $x \in X$ and $y \in Y$, subject to several constraints of the following form:

- Each element $x \in X$ can appear in at most $c(x)$ pairs.
- Each element $y \in Y$ can appear in at most $c(y)$ pairs.
- Each pair $(x, y) \in X \times Y$ can appear in the output at most $c(x, y)$ times.

Each upper bound $c(x)$, $c(y)$, and $c(x, y)$ is either a (typically small) non-negative integer or ∞. Intuitively, we create each pair in our output by *assigning* an element of X to an element of Y.

The maximum-matching problem is a special case, where $c(z) = 1$ for all $z \in X \cup Y$, and each $c(x, y)$ is either 0 or 1, depending on whether the pair xy defines an edge in the underlying bipartite graph.

Here is a slightly more interesting example. A nearby school, famous for its onerous administrative hurdles, decides to organize a dance. Every pair of students (one boy, one girl) who wants to dance must register in advance. School regulations limit each boy-girl pair to at most three dances together, and limits each student to at most ten dances overall. How can we maximize the number of dances? This is a binary assignment problem for the set X of girls and the set Y of boys, where for each girl x and boy y, we have $c(x) = c(y) = 10$ and either $c(x, y) = 3$ (if x and y registered to dance) or $c(x, y) = 0$ (if they didn't register).

Every binary assignment problem can be reduced to a standard maximum flow problem as follows. We construct a flow network $G = (V, E)$ with vertices $X \cup Y \cup \{s, t\}$ and the following edges:

- an edge $s{\rightarrow}x$ with capacity $c(x)$ for each $x \in X$,
- an edge $y{\rightarrow}t$ with capacity $c(y)$ for each $y \in Y$.
- an edge $x{\rightarrow}y$ with capacity $c(x, y)$ for each $x \in X$ and $y \in Y$, and

Because all the edges have integer capacities, the any augmenting-path algorithm constructs an integer maximum flow f^*, which can be decomposed into the sum of $|f^*|$ paths of the form $s{\rightarrow}x{\rightarrow}y{\rightarrow}t$ for some $x \in X$ and $y \in Y$. For each such path, we report the pair (x, y). (Thus, the pair (x, y) appears in our output collection exactly $f(x{\rightarrow}y)$ times.

It is easy to verify (hint, hint) that this collection of pairs satisfies all the necessary constraints. Conversely, any legal collection of r pairs can be transformed into a feasible integer flow in G with value r. Thus, the largest legal collection of pairs corresponds to a maximum flow in G. So our algorithm is correct. If we use Orlin's algorithm to compute the maximum flow, this assignment algorithm runs in $\boldsymbol{O(VE) = O(n^3)}$ ***time***, where $n = |X| + |Y|$.

24.5 Baseball Elimination

Every year millions of baseball fans eagerly watch their favorite team, hoping they will win a spot in the playoffs, and ultimately the World Series. Sadly, most teams are "mathematically eliminated" days or even weeks before the regular season ends. Often, it is easy to spot when a team is eliminated—they can't win enough games to catch up to the current leader in their division. But sometimes the situation is more subtle.

For example, here are the actual standings from the American League East on August 30, 1996.

Team	Won–Lost	Left	NYY	BAL	BOS	TOR	DET
New York Yankees	75–59	28		3	8	7	3
Baltimore Orioles	71–63	28	3		2	7	4
Boston Red Sox	69–66	27	8	2		0	0
Toronto Blue Jays	63–72	27	7	7	0		0
Detroit Tigers	49–86	27	3	4	0	0	

Detroit is clearly behind, but some die-hard Tigers fans may hold out hope that their team can still win. After all, if Detroit wins all 27 of their remaining games, they will end the season with 76 wins, more than any other team has now. So as long as every other team loses every game. . . but that's not possible, because some of those other teams still have to play each other. Here is one complete argument:[2]

> *By winning all of their remaining games, Detroit can finish the season with a record of 76 and 86. If the Yankees win just 2 more games, then they will finish the season with a 77 and 85 record which would put them ahead of Detroit. So, let's suppose the Tigers go undefeated for the rest of the season and the Yankees fail to win another game.*
>
> *The problem with this scenario is that New York still has 8 games left with Boston. If the Red Sox win all of these games, they will end the season with at least 77 wins putting them ahead of the Tigers. Thus, the only way for Detroit to even have a chance of finishing in first place, is for New York to win exactly one of the 8 games with Boston and lose all their other games. Meanwhile, the Sox must loss all the games they play agains teams other than New York. This puts them in a 3-way tie for first place. . . .*
>
> *Now let's look at what happens to the Orioles and Blue Jays in our scenario. Baltimore has 2 games left with with Boston and 3 with New York. So, if everything happens as described above, the Orioles will finish with at least 76 wins. So, Detroit can catch Baltimore only if the Orioles lose all their games to teams other than New York and Boston. In particular, this means that Baltimore must lose all 7 of its remaining games with Toronto. The Blue Jays also have 7 games left with the Yankees and we have already seen that for Detroit to finish in first place, Toronto must will all of these games. But if that happens, the Blue Jays will win at least 14 more games giving them at final record of 77 and 85 or better which means they will finish ahead of the Tigers. So, no matter what happens from this point in the season on, Detroit can not finish in first place in the American League East.*

There has to be a better way to figure this out!

Here is a more abstract formulation of the problem. Our input consists of two arrays $W[1..n]$ and $G[1..n, 1..n]$, where $W[i]$ is the number of games team i has already won, and $G[i, j]$ is the number of upcoming games between teams i and j. We want to determine whether team n can end the season with the most wins (possibly tied with other teams).[3]

In the mid-1960s, Benjamin Schwartz showed that this question can be modeled as an assignment problem: We want to ***assign*** *a winner to each game*, so that team n comes in first place. We have an assignment problem! Let $R[i] = \sum_j G[i, j]$ denote the number of remaining games for team i. We will assume that team n wins all $R[n]$ of its remaining games. Then team n can come in first place if and only if every other team i wins at most $W[n] + R[n] - W[i]$ of its $R[i]$ remaining games.

Since we want to ***assign*** winning teams to games, we start by building a bipartite graph, whose nodes represent the games and the teams. We have $\binom{n}{2}$ *game* nodes $g_{i,j}$, one for each pair $1 \le i < j < n$, and $n-1$ *team* nodes t_i, one for each $1 \le i < n$. For each pair i, j, we add edges $g_{i,j} \to t_i$ and $g_{i,j} \to t_j$ with *infinite* capacity. We add a source vertex s and edges $s \to g_{i,j}$ with capacity $G[i, j]$ for each pair i, j. Finally, we add a target node t and edges $t_i \to t$ with capacity $W[n] - W[i] + R[n]$ for each team i.

[2]Both the example and this argument are taken from http://riot.ieor.berkeley.edu/~baseball/detroit.html.

[3]We assume here that no games end in a tie (always true for Major League Baseball), and that every game is actually played (not always true).

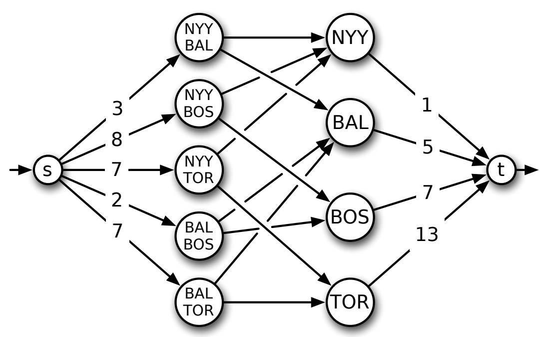
Theorem. *Team n can end the season in first place if and only if there is a feasible flow in this graph that saturates every edge leaving s.*

Proof: Suppose it is possible for team n to end the season in first place. Then every team $i < n$ wins at most $W[n]+R[n]-W[i]$ of the remaining games. For each game between team i and team j that team i wins, add one unit of flow along the path $s \to g_{i,j} \to t_i \to t$. Because there are exactly $G[i,j]$ games between teams i and j, every edge leaving s is saturated. Because each team i wins at most $W[n]+R[n]-W[i]$ games, the resulting flow is feasible.

Conversely, Let f be a feasible flow that saturates every edge out of s. Suppose team i wins exactly $f(g_{i,j} \to t_i)$ games against team j, for all i and j. Then teams i and j play $f(g_{i,j} \to t_i) + f(g_{i,j} \to t_j) = f(s \to g_{i,j}) = G[i,j]$ games, so every upcoming game is played. Moreover, each team i wins a total of $\sum_j f(g_{i,j} \to t_i) = f(t_i \to t) \leq W[n]+R[n]-W[i]$ upcoming games, and therefore at most $W[n]+R[n]$ games overall. Thus, if team n win all their upcoming games, they end the season in first place. □

So, to decide whether our favorite team can win, we construct the flow network, compute a maximum flow, and report whether than maximum flow saturates the edges leaving s. The flow network has $O(n^2)$ vertices and $O(n^2)$ edges, and it can be constructed in $O(n^2)$ time. Using Orlin's algorithm, we can compute the maximum flow in $O(VE) = \boldsymbol{O(n^4)}$ ***time***.

The graph derived from the 1996 American League East standings is shown below. The total capacity of the edges leaving s is 27 (there are 27 remaining games), but the total capacity of the edges entering t is only 26. So the maximum flow has value at most 26, which means that Detroit is mathematically eliminated.



The flow graph for the 1996 American League East standings. Unlabeled edges have infinite capacity.

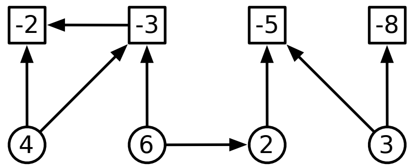
More recently, Kevin Wayne[4] proved that one can determine ***all*** the teams that are mathematically eliminated in only $\boldsymbol{O(n^3)}$ ***time***, essentially using a single maximum-flow computation.

24.6 Project Selection

In our final example, suppose we are given a set of n projects that we could possibly perform; for simplicity, we identify each project by an integer between 1 and n. Some projects cannot be started until certain other projects are completed. This set of dependencies is described by a directed acyclic graph, where an edge $i \to j$ indicates that project i depends on project j. Finally, each project i has an associated *profit* p_i which is given to us if the project is completed; however, some projects have negative profits, which we interpret as positive *costs*. We can choose to finish

[4]Kevin D. Wayne. A new property and a faster algorithm for baseball elimination. *SIAM J. Discrete Math* 14(2):223–229, 2001.

any subset X of the projects that includes all its dependents; that is, for every project $x \in X$, every project that x depends on is also in X. Our goal is to find a valid subset of the projects whose total profit is as large as possible. In particular, if all of the jobs have negative profit, the correct answer is to do nothing.



A dependency graph for a set of projects. Circles represent profitable projects; squares represent costly projects.

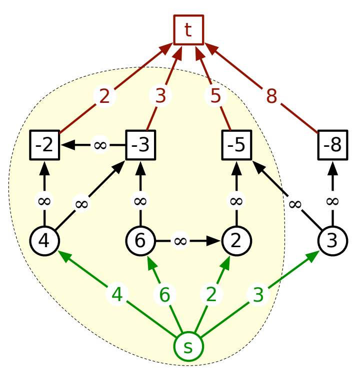
At a high level, our task to partition the projects into two subsets S and T, the jobs we *Select* and the jobs we *Turn down*. So intuitively, we'd like to model our problem as a minimum cut problem in a certain graph. But in which graph? How do we enforce prerequisites? We want to *maximize* profit, but we only know how to find *minimum* cuts. And how do we convert negative profits into positive capacities?

We define a new graph G by adding a source vertex s and a target vertex t to the dependency graph, with an edge $s \to j$ for every profitable job (with $p_j > 0$), and an edge $i \to t$ for every costly job (with $p_i < 0$). Intuitively, we can think of s as a new job ("To the bank!") with profit/cost 0 that we must perform last. We assign edge capacities as follows:

- $c(s \to j) = p_j$ for every profitable job j;
- $c(i \to t) = -p_i$ for every costly job i;
- $c(i \to j) = \infty$ for every dependency edge $i \to j$.

All edge-capacities are positive, so this is a legal input to the maximum cut problem.

Now consider an (s,t)-cut (S,T) in G. If the capacity $\|S,T\|$ is finite, then for every dependency edge $i \to j$, projects i and j are on the same side of the cut, which implies that S is a valid solution. Moreover, we claim that selecting the jobs in S earns us a total profit of $C - \|S,T\|$, where C is the sum of all the positive profits. This claim immediately implies that we can *maximize* our total profit by computing a *minimum* cut in G.



The flow network for the example dependency graph, along with its minimum cut.
The cut has capacity 13 and $C = 15$, so the total profit for the selected jobs is 2.

We prove our key claim as follows. For any subset A of projects, we define three functions:

$$\begin{aligned}
cost(A) &:= \sum_{i\in A\colon p_i<0} -p_i \;=\; \sum_{i\in A} c(i\to t)\\
benefit(A) &:= \sum_{j\in A\colon p_i>0} p_j \;=\; \sum_{j\in A} c(s\to j)\\
profit(A) &:= \sum_{i\in A} p_i \;=\; benefit(A) - cost(A).
\end{aligned}$$

By definition, $C = benefit(S) + benefit(T)$. Because the cut (S, T) has finite capacity, only edges of the form $s\to j$ and $i\to t$ can cross the cut. By construction, every edge $s\to j$ points to a profitable job and each edge $i\to t$ points from a costly job. Thus, $\|S, T\| = cost(S) + benefit(T)$. We immediately conclude that $C - \|S, T\| = benefit(S) - cost(S) = profit(S)$, as claimed.

Exercises

1. Given an undirected graph $G = (V, E)$, with three vertices u, v, and w, describe and analyze an algorithm to determine whether there is a path from u to w that passes through v.

2. Let $G = (V, E)$ be a directed graph where for each vertex v, the in-degree and out-degree of v are equal. Let u and v be two vertices G, and suppose G contains k edge-disjoint paths from u to v. Under these conditions, must G also contain k edge-disjoint paths from v to u? Give a proof or a counterexample with explanation.

3. Consider a directed graph $G = (V, E)$ with multiple source vertices $s_1, s_2, \ldots, s_\sigma$ and multiple target vertices $t_1, t_1, \ldots, t_\tau$, where no vertex is both a source and a target. A *multiterminal flow* is a function $f : E \to \mathbb{R}_{\geq 0}$ that satisfies the flow conservation constraint at every vertex that is neither a source nor a target. The value $|f|$ of a multiterminal flow is the total excess flow out of *all* the source vertices:

$$|f| := \sum_{i=1}^{\sigma} \left(\sum_{w} f(s_i\to w) - \sum_{u} f(u\to s_i) \right)$$

   As usual, we are interested in finding flows with maximum value, subject to capacity constraints on the edges. (In particular, we don't care how much flow moves from any particular source to any particular target.)

   (a) Consider the following algorithm for computing multiterminal flows. The variables f and f' represent flow functions. The subroutine MaxFlow(G, s, t) solves the standard maximum flow problem with source s and target t.

```
MaxMultiFlow(G, s[1..σ], t[1..τ]):
  f ← 0                          ⟨⟨Initialize the flow⟩⟩
  for i ← 1 to σ
      for j ← 1 to τ
          f' ← MaxFlow(G_f, s[i], t[j])
          f ← f + f'             ⟨⟨Update the flow⟩⟩
  return f
```

Prove that this algorithm correctly computes a maximum multiterminal flow in G.

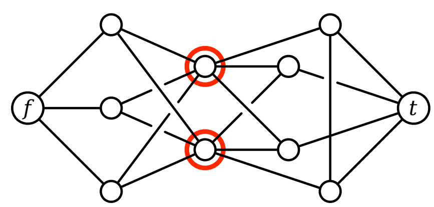
(b) Describe a more efficient algorithm to compute a maximum multiterminal flow in G.

4. A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover all the vertices. Describe and analyze an efficient algorithm to find a cycle cover for a given graph, or correctly report that no cycle cover exists. *[Hint: Use bipartite matching!]*

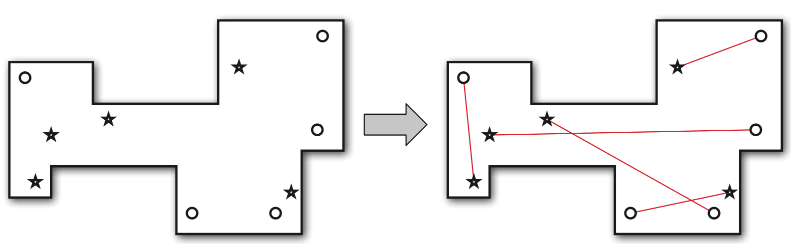
5. The Island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

   Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices f and t represent the stations in Ffarquhar and Tidmouth.

   For example, given the following input graph, your algorithm should return the number 2.



6. The UIUC Computer Science Department is installing a mini-golf course in the basement of the Siebel Center! The playing field is a closed polygon bounded by m horizontal and vertical line segments, meeting at right angles. The course has n *starting points* and n *holes*, in one-to-one correspondence. It is always possible hit the ball along a straight line directly from each starting point to the corresponding hole, without touching the boundary of the playing field. (Players are not allowed to bounce golf balls off the walls; too much glass.) The n starting points and n holes are all at distinct locations.

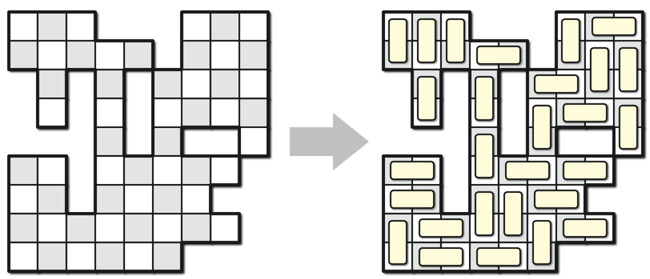


A minigolf course with five starting points (⋆) and five holes (∘), and a legal correspondence between them.

Sadly, the architect's computer crashed just as construction was about to begin. Thanks to the herculean efforts of their sysadmins, they were able to recover the *locations* of the starting points and the holes, but all information about which starting points correspond to which holes was lost!

Describe and analyze an algorithm to compute a one-to-one correspondence between the starting points and the holes that meets the straight-line requirement, or to report that no such correspondence exists. The input consists of the x- and y-coordinates of the m corners of the playing field, the n starting points, and the n holes. Assume you can determine in constant time whether two line segments intersect, given the x- and y-coordinates of their endpoints.

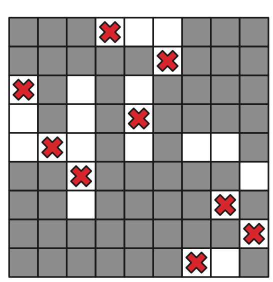
7. Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether one tile the board with dominos—each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.



Your input is a two-dimensional array *Deleted*$[1..n, 1..n]$ of bits, where *Deleted*$[i, j] = $ TRUE if and only if the square in row i and column j has been deleted. Your output is a single bit; you do ***not*** have to compute the actual placement of dominos. For example, for the board shown above, your algorithm should return TRUE.

8. Suppose we are given an $n \times n$ square grid, some of whose squares are colored black and the rest white. Describe and analyze an algorithm to determine whether tokens can be placed on the grid so that

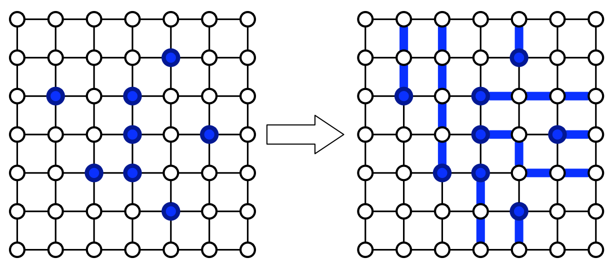
- every token is on a white square;
- every row of the grid contains exactly one token; and
- every column of the grid contains exactly one token.



Your input is a two dimensional array $IsWhite[1..n, 1..n]$ of booleans, indicating which squares are white. Your output is a single boolean. For example, given the grid above as input, your algorithm should return TRUE.

9. An $n \times n$ grid is an undirected graph with n^2 vertices organized into n rows and n columns. We denote the vertex in the ith row and the jth column by (i, j). Every vertex in the grid have exactly four neighbors, except for the *boundary* vertices, which are the vertices (i, j) such that $i = 1$, $i = n$, $j = 1$, or $j = n$.

   Let $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ be distinct vertices, called *terminals*, in the $n \times n$ grid. The ***escape problem*** is to determine whether there are m vertex-disjoint paths in the grid that connect the terminals to any m distinct boundary vertices. Describe and analyze an efficient algorithm to solve the escape problem.



A positive instance of the escape problem, and its solution.

10. The UIUC Faculty Senate has decided to convene a committee to determine whether Chief Illiniwek should become the official ~~mascot~~ *symbol* of the University of Illinois Global Campus.[5] Exactly one faculty member must be chosen from each academic department to serve on this committee. Some faculty members have appointments in multiple departments, but each committee member will represent only one department. For example, if Prof. Blagojevich is affiliated with both the Department of Corruption and the Department of Stupidity, and he is chosen as the Stupidity representative, then someone else must represent Corruption. Finally, University policy requires that any committee on virtual ~~mascots~~ *symbols* must contain the same number of assistant professors, associate professors, and full professors. Fortunately, the number of departments is a multiple of 3.

    Describe an efficient algorithm to select the membership of the Global Illiniwek Committee. Your input is a list of all UIUC faculty members, their ranks (assistant, associate, or full), and their departmental affiliation(s). There are n faculty members and $3k$ departments.

11. You're organizing the First Annual UIUC Computer Science 72-Hour Dance Exchange, to be held all day Friday, Saturday, and Sunday. Several 30-minute sets of music will be played during the event, and a large number of DJs have applied to perform. You need to hire DJs according to the following constraints.

    - Exactly k sets of music must be played each day, and thus $3k$ sets altogether.

[5]Thankfully, the Global Campus has faded into well-deserved obscurity, thanks in part to the 2009 admissions scandal. Imagine MOOCs, but with the same business model and faculty oversight as the University of Phoenix.

- Each set must be played by a single DJ in a consistent music genre (ambient, bubblegum, dubstep, horrorcore, hyphy, trip-hop, Nitzhonot, Kwaito, J-pop, Nashville country, . . .).
- Each genre must be played at most once per day.
- Each candidate DJ has given you a list of genres they are willing to play.
- Each DJ can play at most three sets during the entire event.

Suppose there are n candidate DJs and g different musical genres available. Describe and analyze an efficient algorithm that either assigns a DJ and a genre to each of the $3k$ sets, or correctly reports that no such assignment is possible.

12. The University of Southern North Dakota at Hoople has hired you to write an algorithm to schedule their final exams. Each semester, USNDH offers n different classes. There are r different rooms on campus and t different time slots in which exams can be offered. You are given two arrays $E[1..n]$ and $S[1..r]$, where $E[i]$ is the number of students enrolled in the ith class, and $S[j]$ is the number of seats in the jth room. At most one final exam can be held in each room during each time slot. Class i can hold its final exam in room j only if $E[i] < S[j]$.

    Describe and analyze an efficient algorithm to assign a room and a time slot to each class (or report correctly that no such assignment is possible).

13. Suppose you are running a web site that is visited by the same set of people every day. Each visitor claims membership in one or more *demographic groups*; for example, a visitor might describe himself as male, 40–50 years old, a father, a resident of Illinois, an academic, a blogger, and a fan of Joss Whedon.[6] Your site is supported by advertisers. Each advertiser has told you which demographic groups should see its ads and how many of its ads you must show each day. Altogether, there are n visitors, k demographic groups, and m advertisers.

    Describe an efficient algorithm to determine, given all the data described in the previous paragraph, whether you can show each visitor exactly *one* ad per day, so that every advertiser has its desired number of ads displayed, and every ad is seen by someone in an appropriate demographic group.

14. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

    (a) Describe and analyze an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

    ⋆(b) Suppose we are guaranteed that none of the entries in the input matrix A are integers. Describe and analyze an even faster algorithm that either rounds A or reports correctly

[6] Har har har! Mine is an evil laugh! Now *die!*

that no such rounding is possible. For full credit, your algorithm must run in $O(mn)$ time. *[Hint:* ***Don't*** *use flows.]*

15. *Ad-hoc networks* are made up of low-powered wireless devices. In principle[7], these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be distributed through the area of interest (for example, by dropping them from an airplane); the devices would then automatically configure themselves into a functioning wireless network.

    These devices can communicate only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D.

    We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit its information to some other *backup* device within its communication range. We require each device x to have k potential backup devices, all within distance D of x; we call these k devices the ***backup set*** of x. Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

    So suppose we are given the communication radius D, parameters b and k, and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j. Describe an algorithm that either computes a backup set of size k for each of the n devices, such that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

*16. A *rooted tree* is a directed acyclic graph, in which every vertex has exactly one incoming edge, except for the *root*, which has no incoming edges. Equivalently, a rooted tree consists of a root vertex, which has edges pointing to the roots of zero or more smaller rooted trees. Describe a polynomial-time algorithm to compute, given two rooted trees A and B, the largest common rooted subtree of A and B.

    [Hint: Let $\mathrm{LCS}(u, v)$ *denote the largest common subtree whose root in A is u and whose root in B is v. Your algorithm should compute* $\mathrm{LCS}(u, v)$ *for all vertices u and v using dynamic programming. This would be easy if every vertex had $O(1)$ children, and still straightforward if the children of each node were ordered from left to right and the common subtree had to respect that ordering. But for unordered trees with large degree, you need another trick to combine recursive subproblems efficiently. Don't waste your time trying to reduce the polynomial running time.]*

[7]but not really in practice

"Who are you?" said Lunkwill, rising angrily from his seat. "What do you want?"
"I am Majikthise!" announced the older one.
"And I demand that I am Vroomfondel!" shouted the younger one.
Majikthise turned on Vroomfondel. "It's alright," he explained angrily, "you don't need to demand that."
"Alright!" bawled Vroomfondel banging on an nearby desk. "I am Vroomfondel, and that is not a demand, that is a solid fact! What we demand is solid facts!"
"No we don't!" exclaimed Majikthise in irritation. "That is precisely what we don't *demand!"*
Scarcely pausing for breath, Vroomfondel shouted, "We don't demand solid facts! What we demand is a total absence of solid facts. I demand that I may or may not be Vroomfondel!"

— Douglas Adams, *The Hitchhiker's Guide to the Galaxy* (1979)

★25 Extensions of Maximum Flow

25.1 Maximum Flows with Edge Demands

Now suppose each directed edge e in has both a capacity $c(e)$ and a *demand* $d(e) \le c(e)$, and we want a flow f of maximum value that satisfies $d(e) \le f(e) \le c(e)$ at every edge e. We call a flow that satisfies these constraints a *feasible* flow. In our original setting, where $d(e) = 0$ for every edge e, the zero flow is feasible; however, in this more general setting, even determining whether a feasible flow exists is a nontrivial task.

Perhaps the easiest way to find a feasible flow (or determine that none exists) is to reduce the problem to a standard maximum flow problem, as follows. The input consists of a directed graph $G = (V, E)$, nodes s and t, demand function $d : E \to \mathbb{R}$, and capacity function $c : E \to \mathbb{R}$. Let D denote the sum of all edge demands in G:

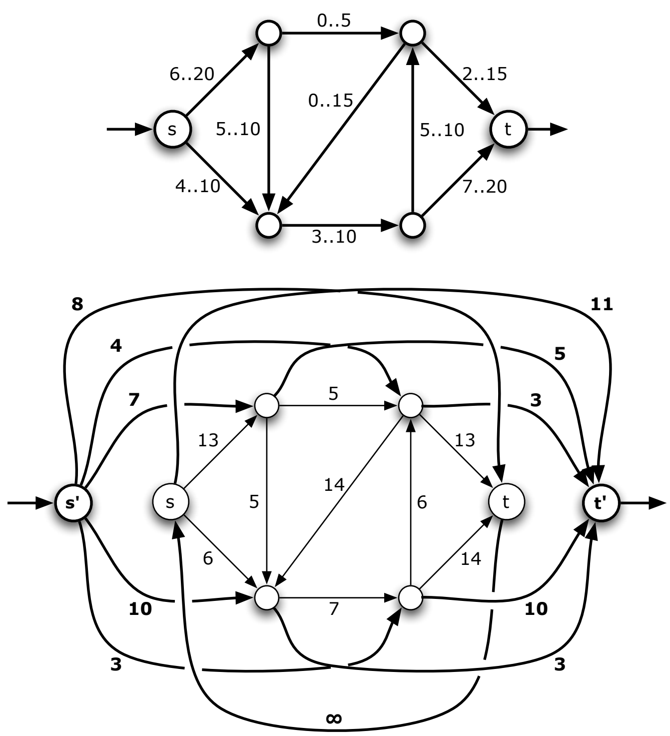
$$D := \sum_{u \to v \in E} d(u \to v).$$

We construct a new graph $G' = (V', E')$ from G by adding new source and target vertices s' and t', adding edges from s' to each vertex in V, adding edges from each vertex in V to t', and finally adding an edge from t to s. We also define a new capacity function $c' : E' \to \mathbb{R}$ as follows:

- For each vertex $v \in V$, we set $c'(s' \to v) = \sum_{u \in V} d(u \to v)$ and $c'(v \to t') = \sum_{w \in V} d(v \to w)$.
- For each edge $u \to v \in E$, we set $c'(u \to v) = c(u \to v) - d(u \to v)$.
- Finally, we set $c'(t \to s) = \infty$.

Intuitively, we construct G' by replacing any edge $u \to v$ in G with three edges: an edge $u \to v$ with capacity $c(u \to v) - d(u \to v)$, an edge $s' \to v$ with capacity $d(u \to v)$, and an edge $u \to t'$ with capacity $d(u \to v)$. If this construction produces multiple edges from s' to the same vertex v (or to t' from the same vertex v), we merge them into a single edge with the same total capacity.

In G', the total capacity out of s' and the total capacity into t' are both equal to D. We call a flow with value exactly D a *saturating* flow, since it saturates all the edges leaving s' or entering t'. If G' has a saturating flow, it must be a maximum flow, so we can find it using any max-flow algorithm.



A flow network G with demands and capacities (written $d\,..\,c$), and the transformed network G'.

Lemma 1. *G has a feasible (s,t)-flow if and only if G' has a saturating (s',t')-flow.*

Proof: Let $f : E \to \mathbb{R}$ be a feasible (s,t)-flow in the original graph G. Consider the following function $f' : E' \to \mathbb{R}$:

$$\begin{aligned} f'(u{\to}v) &= f(u{\to}v) - d(u{\to}v) && \text{for all } u{\to}v \in E \\ f'(s'{\to}v) &= \sum_{u\in V} d(u{\to}v) && \text{for all } v \in V \\ f'(v{\to}t') &= \sum_{w\in V} d(u \to w) && \text{for all } v \in V \\ f'(t{\to}s) &= |f| \end{aligned}$$

We easily verify that f' is a saturating (s',t')-flow in G. The admissibility of f implies that $f(e) \geq d(e)$ for every edge $e \in E$, so $f'(e) \geq 0$ everywhere. Admissibility also implies $f(e) \leq c(e)$ for every edge $e \in E$, so $f'(e) \leq c'(e)$ everywhere. Tedious algebra implies that

$$\sum_{u\in V'} f'(u{\to}v) = \sum_{w\in V'} f(v{\to}w)$$

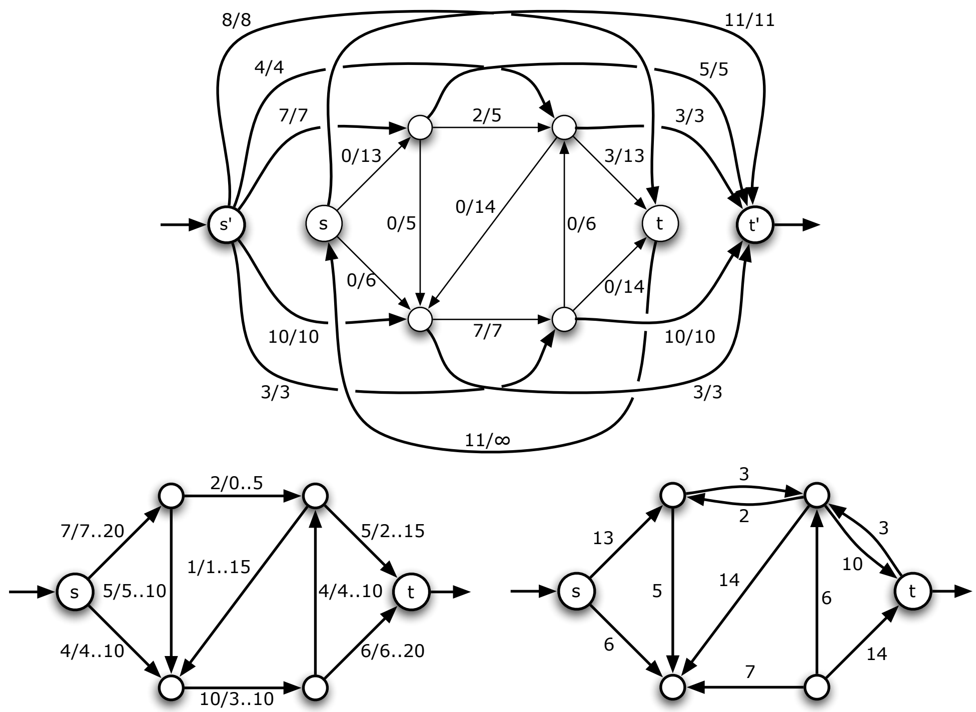
for every vertex $v \in V$ (including s and t). Thus, f' is a legal (s',t')-flow, and every edge out of s' or into t' is clearly saturated. Intuitively, f' diverts $d(u{\to}v)$ units of flow from u directly to the new target t', and injects the same amount of flow into v directly from the new source s'.

The same tedious algebra implies that for any saturating (s',t')-flow $f' : E' \to \mathbb{R}$ for G', the function $f = f'|_E + d$ is a feasible (s,t)-flow in G. □

Thus, we can compute a feasible (s,t)-flow for G, if one exists, by searching for a maximum (s',t')-flow in G' and checking that it is saturating. Once we've found a feasible (s,t)-flow in G, we can transform it into a maximum flow using an augmenting-path algorithm, but with one small change. To ensure that *every* flow we consider is feasible, we must redefine the residual capacity of an edge as follows:

$$c_f(u\to v) = \begin{cases} c(u\to v) - f(u\to v) & \text{if } u\to v \in E, \\ f(v\to u) - d(v\to u) & \text{if } v\to u \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Otherwise, the algorithm is unchanged. If we use the Dinitz/Edmonds-Karp fat-pipe algorithm, we get an overall running time of $\boldsymbol{O(VE^2)}$.



A saturating flow f' in G', the corresponding feasible flow f in G, and the corresponding residual network G_f.

25.2 Node Supplies and Demands

Another useful variant to consider allows flow to be injected or extracted from the flow network at vertices other than s or t. Let $x\colon (V \setminus \{s,t\}) \to \mathbb{R}$ be an *excess* function describing how much flow is to be injected (or extracted if the value is negative) at each vertex. We now want a maximum 'flow' that satisfies the variant balance condition

$$\sum_{u\in V} f(u\to v) - \sum_{w\in V} f(v\to w) = x(v)$$

for every node v except s and t, or prove that no such flow exists. As above, call such a function f a *feasible flow*.

As for flows with edge demands, the only real difficulty in finding a maximum flow under these modified constraints is finding a feasible flow (if one exists). We can reduce this problem to a standard max-flow problem, just as we did for edge demands.

To simplify the transformation, let us assume without loss of generality that the total excess in the network is zero: $\sum_v x(v) = 0$. If the total excess is positive, we add an infinite capacity edge $t \to \tilde{t}$, where $\tilde{t}$ is a new target node, and set $x(t) = -\sum_v x(v)$. Similarly, if the total excess is negative, we add an infinite capacity edge $\tilde{s} \to s$, where $\tilde{s}$ is a new source node, and set $x(s) = -\sum_v x(v)$. In both cases, every feasible flow in the modified graph corresponds to a feasible flow in the original graph.

As before, we modify G to obtain a new graph G' by adding a new source s', a new target t', an infinite-capacity edge $t \to s$ from the old target to the old source, and several edges from s' and to t'. Specifically, for each vertex v, if $x(v) > 0$, we add a new edge $s' \to v$ with capacity $x(v)$, and if $x(v) < 0$, we add an edge $v \to t'$ with capacity $-x(v)$. As before, we call an (s', t')-flow in G' *saturating* if every edge leaving s' or entering t' is saturated; any saturating flow is a maximum flow. It is easy to check that saturating flows in G' are in direct correspondence with feasible flows in G; we leave details as an exercise (hint, hint).

Similar reductions allow us to several other variants of the maximum flow problem using the same path-augmentation techniques. For example, we could associate capacities and demands with the vertices instead of (or in addition to) the edges, as well as a *range* of excesses with every vertex, instead of a single excess value.

25.3 Minimum-Cost Flows

Now imagine that each edge e in the network has both a capacity $c(e)$ and a *cost* $\$(e)$. The cost function describes the cost of sending a unit of flow through the edges; thus, the cost any flow f is defined as follows:

$$\$(f) = \sum_{e \in E} \$(e) \cdot f(e).$$

The ***minimum-cost maximum-flow*** problem is to compute a maximum flow of minimum cost. If the network has only one maximum flow, that's what we want, but if there is more than one maximum flow, we want the maximum flow whose cost is as small as possible. Costs can either be positive, negative, or zero. However, if an edge $u \to v$ and its reversal $v \to u$ both appear in the graph, their costs must sum to zero: $\$(u \to v) = -\$(v \to u)$. Otherwise, we could make an infinite profit by pushing flow back and forth along the edge!

Each augmentation step in the standard Ford-Fulkerson algorithm both increases the value of the flow and changes its cost. If the total cost of the augmenting path is positive, the cost of the flow decreases; conversely, if the total cost of the augmenting path is negative, the cost of the flow decreases. We can also change the cost of the flow without changing its value, by augmenting along a directed *cycle* in the residual graph. Again, augmenting along a negative-cost cycle decreases the cost of the flow, and augmenting along a positive-cost cycle increases the cost of the flow.

It follows immediately that a flow f is a minimum-cost maximum flow in G if and only if the residual graph G_f has no directed paths from s to t *and no negative-cost cycles*.

We can compute a min-cost max-flow using the so-called *cycle cancelling* algorithm first proposed by Morton Klein in 1967. The algorithm has two phases; in the first, we compute an arbitrary maximum flow f, using any method we like. The second phase repeatedly decreases the cost of f, by augmenting f along a negative-cost cycle in the residual graph G_f, until no such cycle exists. As in Ford-Fulkerson, the amount of flow we push around each cycle is equal to the minimum residual capacity of any edge on the cycle.

In each iteration of the second phase, we can use a modification of Shimbel's shortest path algorithm (often called "Bellman-Ford") to find a negative-cost cycle in $O(VE)$ time. To bound the number of iterations in the second phase, we assume that both the capacity and the cost of each edge is an integer, and we define

$$C = \max_{e \in E} c(e) \qquad \text{and} \qquad D = \max_{e \in E} |\$(e)|.$$

The cost of any feasible flow is clearly between $-ECD$ and ECD, and each augmentation step decreases the cost of the flow by a positive integer, and therefore by at least 1. We conclude that the second phase requires at most $2ECD$ iterations, and therefore runs in $\boldsymbol{O(VE^2CD)}$ time. As with the raw Ford-Fulkerson algorithm, this running time is exponential in the complexity of the input, and it may never terminate if the capacities and/or costs are irrational.

Like Ford-Fulkerson, more careful choices of *which* cycle to cancel can lead to more efficient algorithms. Unfortunately, some obvious choices are NP-hard to compute, including the cycle with most negative cost and the negative cycle with the fewest edges. In the late 1980s, Andrew Goldberg and Bob Tarjan developed a min-cost flow algorithm that repeatedly cancels the so-called *minimum-mean cycle*, which is the cycle whose *average cost per edge* is smallest. By combining an algorithm of Karp to compute minimum-mean cycles in $O(EV)$ time, efficient dynamic tree data structures, and other sophisticated techniques that are (unfortunately) beyond the scope of this class, their algorithm achieves a running time of $\boldsymbol{O(E^2V\log^2 V)}$. The fastest min-cost max-flow algorithm currently known,[1] due to James Orlin, reduces the problem to $O(E\log V)$ iterations of Dijkstra's shortest-path algorithm; Orlin's algorithm runs in $\boldsymbol{O(E^2\log V + EV\log^2 V)}$ ***time***.

25.4 Maximum-Weight Matchings

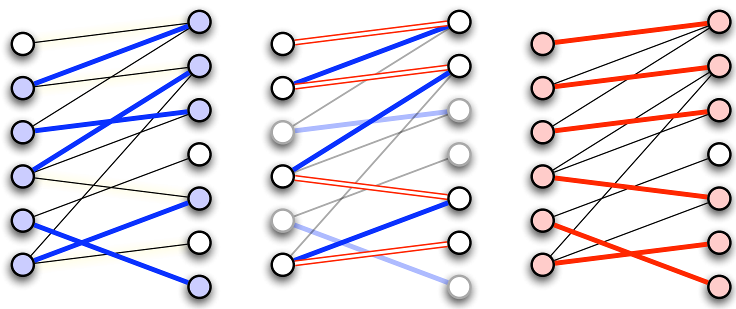
Recall from the previous lecture that we can find a maximum-cardinality matching in any bipartite graph in $O(VE)$ time by reduction to the standard maximum flow problem.

Now suppose the input graph has *weighted* edges, and we want to find the matching with maximum total *weight*. Given a bipartite graph $G = (U \times W, E)$ and a non-negative weight function $w \colon E \to \mathbb{R}$, the goal is to compute a matching M whose total weight $w(M) = \sum_{uw \in M} w(uw)$ is as large as possible. Max-weight matchings can't be found directly using standard max-flow algorithms[2], but we can modify the algorithm for maximum-cardinality matchings described above.

It will be helpful to reinterpret the behavior of our earlier algorithm directly in terms of the original bipartite graph instead of the derived flow network. Our algorithm maintains a matching M, which is initially empty. We say that a vertex is *matched* if it is an endpoint of an edge in M. At each iteration, we find an *alternating path* π that starts and ends at unmatched vertices and alternates between edges in $E \setminus M$ and edges in M. Equivalently, let G_M be the directed graph obtained by orienting every edge in M from W to U, and every edge in $E \setminus M$ from U to W. An alternating path is just a directed path in G_M between two unmatched vertices. Any alternating path has odd length and has exactly one more edge in $E \setminus M$ than in M. The iteration ends by setting $M \leftarrow M \oplus \pi$, thereby increasing the number of edges in M by one. The max-flow/min-cut theorem implies that when there are no more alternating paths, M is a maximum matching.

[1] at least, among algorithms whose running times do not depend on C and D

[2] However, max-flow algorithms can be modified to compute maximum *weighted* flows, where every edge has both a capacity and a weight, and the goal is to maximize $\sum_{u\to v} w(u\to v) f(u\to v)$.

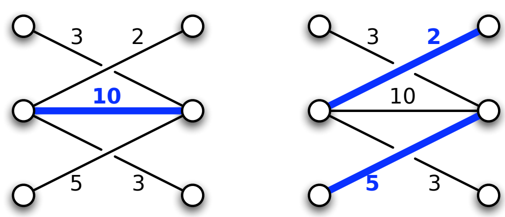


A matching M with 5 edges, an alternating path π, and the augmented matching $M \oplus \pi$ with 6 edges.

If the edges of G are weighted, we only need to make two changes to the algorithm. First, instead of looking for an arbitrary alternating path at each iteration, we look for the alternating path π such that $M \oplus \pi$ has largest weight. Suppose we weight the edges in the residual graph G_M as follows:

$$\begin{aligned} w'(u \to w) &= -w(uw) && \text{for all } uw \notin M \\ w'(w \to u) &= w(uw) && \text{for all } uw \in M \end{aligned}$$

We now have $w(M \oplus \pi) = w(M) - w'(\pi)$. ***Thus, the correct augmenting path π must be the directed path in G_M with minimum total residual weight $w'(\pi)$.*** Second, because the matching with the maximum *weight* may not be the matching with the maximum *cardinality*, we return the heaviest matching considered in *any* iteration of the algorithm.



A maximum-weight matching is not necessarily a maximum-cardinality matching.

Before we determine the running time of the algorithm, we need to check that it actually finds the maximum-weight matching. After all, it's a greedy algorithm, and greedy algorithms don't work unless you prove them into submission! Let M_i denote the maximum-weight matching in G with exactly i edges. In particular, $M_0 = \varnothing$, and the global maximum-weight matching is equal to M_i for some i. (The figure above show M_1 and M_2 for the same graph.) Let G_i denote the directed residual graph for M_i, let w_i denote the residual weight function for M_i as defined above, and let π_i denote the directed path in G_i such that $w_i(\pi_i)$ is minimized. To simplify the proof, I will assume that there is a unique maximum-weight matching M_i of any particular size; this assumption can be enforced by applying a consistent tie-breaking rule. With this assumption in place, the correctness of our algorithm follows inductively from the following lemma.

Lemma 2. *If G contains a matching with $i+1$ edges, then $M_{i+1} = M_i \oplus \pi_i$.*

Proof: I will prove the equivalent statement $M_{i+1} \oplus M_i = \pi_{i-1}$. To simplify notation, call an edge in $M_{i+1} \oplus M_i$ *red* if it is an edge in M_{i+1}, and *blue* if it is an edge in M_i.

The graph $M_{i+1} \oplus M_i$ has maximum degree 2, and therefore consists of pairwise disjoint paths and cycles, each of which alternates between red and blue edges. Since G is bipartite, every cycle must have even length. The number of edges in $M_{i+1} \oplus M_i$ is odd; specifically, $M_{i+1} \oplus M_i$ has $2i + 1 - 2k$ edges, where k is the number of edges that are in both matchings. Thus, $M_{i+1} \oplus M_i$ contains an odd number of paths of odd length, some number of paths of even length, and some number of cycles of even length.

Let γ be a cycle in $M_{i+1} \oplus M_i$. Because γ has an equal number of edges from each matching, $M_i \oplus \gamma$ is another matching with i edges. The total weight of this matching is exactly $w(M_i) - w_i(\gamma)$, which must be less than $w(M_i)$, so $w_i(\gamma)$ must be positive. On the other hand, $M_{i+1} \oplus \gamma$ is a matching with $i+1$ edges whose total weight is $w(M_{i+1}) + w_i(\gamma) < w(M_{i+1})$, so $w_i(\gamma)$ must be negative! We conclude that no such cycle γ exists; $M_{i+1} \oplus M_i$ consists entirely of disjoint *paths*.

Exactly the same reasoning implies that no path in $M_{i+1} \oplus M_i$ has an even number of edges.

Finally, since the number of red edges in $M_{i+1} \oplus M_i$ is one more than the number of blue edges, the number of paths that start with a red edge is exactly one more than the number of paths that start with a blue edge. The same reasoning as above implies that $M_{i+1} \oplus M_i$ does not contain a blue-first path, because we can pair it up with a red-first path.

We conclude that $M_{i+1} \oplus M_i$ consists of a single alternating path π whose first edge is red. Since $w(M_{i+1}) = w(M_i) - w_i(\pi)$, the path π must be the one with minimum weight $w_i(\pi)$. □

We can find the alternating path π_i using a single-source shortest path algorithm. Modify the residual graph G_i by adding zero-weight edges from a new source vertex s to every *unmatched* node in U, and from every *unmatched* node in W to a new target vertex t, exactly as in out unweighted matching algorithm. Then π_i is the shortest path from s to t in this modified graph. Since M_i is the maximum-weight matching with i vertices, G_i has no negative cycles, so this shortest path is well-defined. We can compute the shortest path in G_i in $O(VE)$ time using Shimbel's algorithm, so the overall running time our algorithm is $\boldsymbol{O(V^2E)}$.

The residual graph G_i has negative-weight edges, so we can't speed up the algorithm by replacing Shimbel's algorithm with Dijkstra's. However, we can use a variant of Johnson's all-pairs shortest path algorithm to improve the running time to $\boldsymbol{O(VE + V^2 \log V)}$. Let $d_i(v)$ denote the distance from s to v in the residual graph G_i, using the distance function w_i. Let $\tilde{w}_i$ denote the modified distance function $\tilde{w}_i(u{\to}v) = d_{i-1}(u) + w_i(u{\to}v) - d_{i-1}(v)$. As we argued in the discussion of Johnson's algorithm, shortest paths with respect to w_i are still shortest paths with respect to $\tilde{w}_i$. Moreover, $\tilde{w}_i(u{\to}v) > 0$ for every edge $u{\to}v$ in G_i:

- If $u{\to}v$ is an edge in G_{i-1}, then $w_i(u{\to}v) = w_{i-1}(u{\to}v)$ and $d_{i-1}(v) \le d_{i-1}(u) + w_{i-1}(u{\to}v)$.
- If $u{\to}v$ is not in G_{i-1}, then $w_i(u{\to}v) = -w_{i-1}(v{\to}u)$ and $v{\to}u$ is an edge in the shortest path π_{i-1}, so $d_{i-1}(u) = d_{i-1}(v) + w_{i-1}(v{\to}u)$.

Let $\tilde{d}_i(v)$ denote the shortest path distance from s to v with respect to the distance function $\tilde{w}_i$. Because $\tilde{w}_i$ is positive everywhere, we can quickly compute $\tilde{d}_i(v)$ for all v using Dijkstra's algorithm. This gives us both the shortest alternating path π_i and the distances $d_i(v) = \tilde{d}_i(v) + d_{i-1}(v)$ needed for the next iteration.

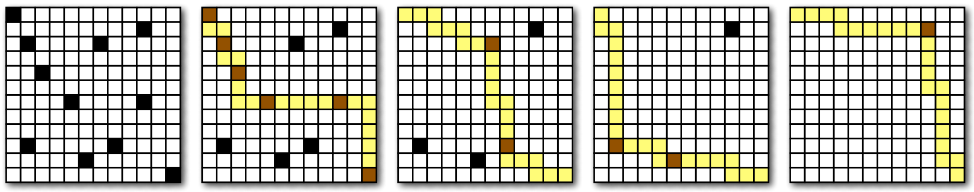
Exercises

1. Suppose we are given a directed graph $G = (V, E)$, two vertices s an t, and a capacity function $c\colon V \to \mathbb{R}^+$. A flow f is *feasible* if the total flow into every vertex v is at most $c(v)$:

$$\sum_u f(u{\to}v) \le c(v) \quad \text{for every vertex } v.$$

Describe and analyze an efficient algorithm to compute a feasible flow of maximum value.

2. Suppose we are given an $n \times n$ grid, some of whose cells are marked; the grid is represented by an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \text{True}$ if and only if cell (i, j) is marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. Our goal is to cover the marked cells with as few monotone paths as possible.



Greedily covering the marked cells in a grid with four monotone paths.

   (a) Describe an algorithm to find a monotone path that covers the largest number of marked cells.

   (b) There is a natural greedy heuristic to find a small cover by monotone paths: If there are any marked cells, find a monotone path π that covers the largest number of marked cells, unmark any cells covered by π those marked cells, and recurse. Show that this algorithm does *not* always compute an optimal solution.

   (c) Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell.

3. Suppose we are given a set of boxes, each specified by their height, width, and depth in centimeters. All three side lengths of every box lie strictly between 10cm and 20cm. As you should expect, one box can be placed inside another if the smaller box can be rotated so that its height, width, and depth are respectively smaller than the height, width, and depth of the larger box. Boxes can be nested recursively. Call a box is *visible* if it is not inside another box.

   Describe and analyze an algorithm to nest the boxes so that the number of visible boxes is as small as possible.

4. Let G be a directed flow network whose edges have costs, but which contains no negative-cost cycles. Prove that one can compute a minimum-cost maximum flow in G using a variant of Ford-Fulkerson that repeatedly augments the (s, t)-path of *minimum total cost* in the current residual graph. What is the running time of this algorithm?

5. An ***(s, t)-series-parallel*** graph is an directed acyclic graph with two designated vertices s (the *source*) and t (the *target* or *sink*) and with one of the following structures:

   - **Base case:** A single directed edge from s to t.
   - **Series:** The union of an (s, u)-series-parallel graph and a (u, t)-series-parallel graph that share a common vertex u but no other vertices or edges.

- **Parallel:** The union of two smaller (s,t)-series-parallel graphs with the same source s and target t, but with no other vertices or edges in common.

(a) Describe an efficient algorithm to compute a maximum flow from s to t in an (s,t)-series-parallel graph with arbitrary edge capacities.

(b) Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s,t)-series-parallel graph whose edges have *unit* capacity and arbitrary costs.

*(c) Describe an efficient algorithm to compute a *minimum-cost* maximum flow from s to t in an (s,t)-series-parallel graph whose edges have *arbitrary* capacities and costs.

The greatest flood has the soonest ebb;
the sorest tempest the most sudden calm;
the hottest love the coldest end; and
from the deepest desire oftentimes ensues the deadliest hate.

— Socrates

Th' extremes of glory and of shame,
Like east and west, become the same.

— Samuel Butler, *Hudibras* Part II, Canto I (c. 1670)

Extremes meet, and there is no better example
than the haughtiness of humility.

— Ralph Waldo Emerson, "Greatness",
in *Letters and Social Aims* (1876)

★26 Linear Programming

The maximum flow/minimum cut problem is a special case of a very general class of problems called *linear programming*. Many other optimization problems fall into this class, including minimum spanning trees and shortest paths, as well as several common problems in scheduling, logistics, and economics. Linear programming was used implicitly by Fourier in the early 1800s, but it was first formalized and applied to problems in economics in the 1930s by Leonid Kantorovich. Kantorivich's work was hidden behind the Iron Curtain (where it was largely ignored) and therefore unknown in the West. Linear programming was rediscovered and applied to shipping problems in the early 1940s by Tjalling Koopmans. The first complete algorithm to solve linear programming problems, called the *simplex method*, was published by George Dantzig in 1947. Koopmans first proposed the name "linear programming" in a discussion with Dantzig in 1948. Kantorovich and Koopmans shared the 1975 Nobel Prize in Economics "for their contributions to the theory of optimum allocation of resources". Dantzig did not; his work was apparently too pure. Koopmans wrote to Kantorovich suggesting that they refuse the prize in protest of Dantzig's exclusion, but Kantorovich saw the prize as a vindication of his use of mathematics in economics, which his Soviet colleagues had written off as "a means for apologists of capitalism".

A linear programming problem asks for a vector $x \in \mathbb{R}^d$ that maximizes (or equivalently, minimizes) a given linear function, among all vectors x that satisfy a given set of linear inequalities. The general form of a linear programming problem is the following:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{d} c_j x_j \\
\text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1 \mathinner{..} p \\
& \sum_{j=1}^{d} a_{ij} x_j = b_i \quad \text{for each } i = p+1 \mathinner{..} p+q \\
& \sum_{j=1}^{d} a_{ij} x_j \ge b_i \quad \text{for each } i = p+q+1 \mathinner{..} n
\end{aligned}
$$

Here, the input consists of a matrix $A = (a_{ij}) \in \mathbb{R}^{n \times d}$, a column vector $b \in \mathbb{R}^n$, and a row vector $c \in \mathbb{R}^d$. Each coordinate of the vector x is called a *variable*. Each of the linear inequalities is called a *constraint*. The function $x \mapsto c \cdot x$ is called the *objective function*. I will always use d to denote the number of variables, also known as the *dimension* of the problem. The number of constraints is usually denoted n.

A linear programming problem is said to be in *canonical form*[1] if it has the following structure:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1 \,..\, n \\ & x_j \ge 0 \quad \text{for each } j = 1 \,..\, d \end{aligned}$$

We can express this canonical form more compactly as follows. For two vectors $x = (x_1, x_2, \dots, x_d)$ and $y = (y_1, y_2, \dots, y_d)$, the expression $x \ge y$ means that $x_i \ge y_i$ for every index i.

$$\boxed{\begin{aligned} \max \quad & c \cdot x \\ \text{s.t. } & Ax \le b \\ & x \ge 0 \end{aligned}}$$

Any linear programming problem can be converted into canonical form as follows:

- For each variable x_j, add the equality constraint $x_j = x_j^+ - x_j^-$ and the inequalities $x_j^+ \ge 0$ and $x_j^- \ge 0$.
- Replace any equality constraint $\sum_j a_{ij} x_j = b_i$ with two inequality constraints $\sum_j a_{ij} x_j \ge b_i$ and $\sum_j a_{ij} x_j \le b_i$.
- Replace any upper bound $\sum_j a_{ij} x_j \ge b_i$ with the equivalent lower bound $\sum_j -a_{ij} x_j \le -b_i$.

This conversion potentially double the number of variables and the number of constraints; fortunately, it is rarely necessary in practice.

Another useful format for linear programming problems is *slack form*[2], in which every inequality is of the form $x_j \ge 0$:

$$\boxed{\begin{aligned} \max \quad & c \cdot x \\ \text{s.t. } & Ax = b \\ & x \ge 0 \end{aligned}}$$

It's fairly easy to convert any linear programming problem into slack form. Slack form is especially useful in executing the simplex algorithm (which we'll see in the next lecture).

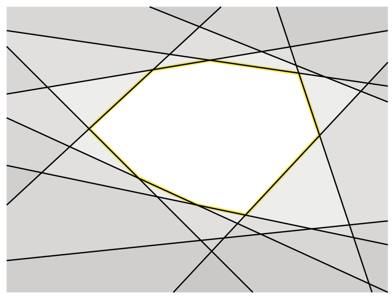
26.1 The Geometry of Linear Programming

A point $x \in \mathbb{R}^d$ is *feasible* with respect to some linear programming problem if it satisfies all the linear constraints. The set of all feasible points is called the *feasible region* for that linear program.

[1]Confusingly, some authors call this *standard form*.
[2]Confusingly, some authors call this *standard form*.

The feasible region has a particularly nice geometric structure that lends some useful intuition to the linear programming algorithms we'll see later.

Any linear equation in d variables defines a *hyperplane* in $\mathbb{R}^d$; think of a line when $d = 2$, or a plane when $d = 3$. This hyperplane divides $\mathbb{R}^d$ into two *halfspaces*; each halfspace is the set of points that satisfy some linear inequality. Thus, the set of feasible points is the intersection of several hyperplanes (one for each equality constraint) and halfspaces (one for each inequality constraint). The intersection of a finite number of hyperplanes and halfspaces is called a *polyhedron*. It's not hard to verify that any halfspace, and therefore any polyhedron, is *convex*—if a polyhedron contains two points x and y, then it contains the entire line segment $\overline{xy}$.



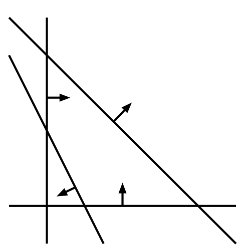
A two-dimensional polyhedron (white) defined by 10 linear inequalities.

By rotating $\mathbb{R}^d$ (or choosing a coordinate frame) so that the objective function points downward, we can express *any* linear programming problem in the following geometric form:

$$\boxed{\text{Find the lowest point in a given polyhedron.}}$$

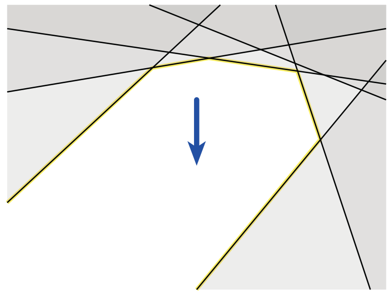
With this geometry in hand, we can easily picture two pathological cases where a given linear programming problem has no solution. The first possibility is that there are no feasible points; in this case the problem is called *infeasible*. For example, the following LP problem is infeasible:

$$\begin{aligned} \text{maximize } & x - y \\ \text{subject to } & 2x + y \le 1 \\ & x + y \ge 2 \\ & x, y \ge 0 \end{aligned}$$



An infeasible linear programming problem; arrows indicate the constraints.

The second possibility is that there are feasible points at which the objective function is arbitrarily large; in this case, we call the problem *unbounded*. The same polyhedron could be unbounded for some objective functions but not others, or it could be unbounded for every objective function.



A two-dimensional polyhedron (white) that is unbounded downward but bounded upward.

26.2 Example 1: Shortest Paths

We can compute the length of the shortest path from s to t in a weighted directed graph by solving the following very simple linear programming problem.

$$\begin{array}{rl} \text{maximize} & d_t \\ \text{subject to} & d_s = 0 \\ & d_v - d_u \le \ell_{u\to v} \quad \text{for every edge } u\to v \end{array}$$

Here, $\ell_{u\to v}$ is the length of the edge $u\to v$. Each variable d_v represents a tentative shortest-path distance from s to v. The constraints mirror the requirement that every edge in the graph must be relaxed. These relaxation constraints imply that in any feasible solution, d_v is *at most* the shortest path distance from s to v. Thus, somewhat counterintuitively, we are correctly *maximizing* the objective function to compute the *shortest* path! In the optimal solution, the objective function d_t is the actual shortest-path distance from s to t, but for any vertex v that is not on the shortest path from s to t, d_v may be an underestimate of the true distance from s to v. However, we can obtain the true distances from s to every other vertex by modifying the objective function:

$$\begin{array}{rl} \text{maximize} & \sum_v d_v \\ \text{subject to} & d_s = 0 \\ & d_v - d_u \le \ell_{u\to v} \quad \text{for every edge } u\to v \end{array}$$

There is another formulation of shortest paths as an LP minimization problem using an indicator variable $x_{u\to v}$ for each edge $u\to v$.

$$\begin{array}{rl} \text{minimize} & \sum_{u\to v} \ell_{u\to v} \cdot x_{u\to v} \\ \text{subject to} & \sum_u x_{u\to s} - \sum_w x_{s\to w} = 1 \\ & \sum_u x_{u\to t} - \sum_w x_{t\to w} = -1 \\ & \sum_u x_{u\to v} - \sum_w x_{v\to w} = 0 \quad \text{for every vertex } v \ne s, t \\ & x_{u\to v} \ge 0 \quad \text{for every edge } u\to v \end{array}$$

Intuitively, $x_{u\to v} = 1$ means $u\to v$ lies on the shortest path from s to t, and $x_{u\to v} = 0$ means $u\to v$ does not lie on this shortest path. The constraints merely state that the path should start at s, end at t, and either pass through or avoid every other vertex v. Any path from s to t—in particular, the shortest path—clearly implies a feasible point for this linear program.

However, there are other feasible solutions, possibly even *optimal* solutions, with non-integral values that do not represent paths. Nevertheless, there is always an optimal solution in which every x_e is either 0 or 1 and the edges e with $x_e = 1$ comprise the shortest path. (This fact is by no means obvious, but a proof is beyond the scope of these notes.) Moreover, in any optimal solution, even if not every x_e is an integer, the objective function gives the shortest path distance!

26.3 Example 2: Maximum Flows and Minimum Cuts

Recall that the input to the maximum (s,t)-flow problem consists of a weighted directed graph $G = (V, E)$, two special vertices s and t, and a function assigning a non-negative *capacity* c_e to each edge e. Our task is to choose the flow f_e across each edge e, as follows:

$$\begin{array}{lrl}
\text{maximize} & \displaystyle\sum_w f_{s\to w} - \sum_u f_{u\to s} & \\
\text{subject to} & \displaystyle\sum_w f_{v\to w} - \sum_u f_{u\to v} = 0 & \text{for every vertex } v \neq s,t \\
& f_{u\to v} \leq c_{u\to v} & \text{for every edge } u\to v \\
& f_{u\to v} \geq 0 & \text{for every edge } u\to v
\end{array}$$

Similarly, the minimum cut problem can be formulated using 'indicator' variables similarly to the shortest path problem. We have a variable S_v for each vertex v, indicating whether $v \in S$ or $v \in T$, and a variable $X_{u\to v}$ for each edge $u\to v$, indicating whether $u \in S$ and $v \in T$, where (S, T) is some (s,t)-cut.[3]

$$\begin{array}{lrl}
\text{minimize} & \displaystyle\sum_{u\to v} c_{u\to v} \cdot X_{u\to v} & \\
\text{subject to} & X_{u\to v} + S_v - S_u \geq 0 & \text{for every edge } u\to v \\
& X_{u\to v} \geq 0 & \text{for every edge } u\to v \\
& S_s = 1 & \\
& S_t = 0 &
\end{array}$$

Like the minimization LP for shortest paths, there can be optimal solutions that assign fractional values to the variables. Nevertheless, the minimum value for the objective function is the cost of the minimum cut, and there is an optimal solution for which every variable is either 0 or 1, representing an actual minimum cut. No, this is not obvious; in particular, my claim is not a proof!

26.4 Linear Programming Duality

Each of these pairs of linear programming problems is related by a transformation called *duality*. For any linear programming problem, there is a corresponding dual linear program that can be obtained by a mechanical translation, essentially by swapping the constraints and the variables.

[3]These two linear programs are not quite *syntactic* duals; I've added two redundant variables S_s and S_t to the min-cut program to increase readability.

The translation is simplest when the LP is in canonical form:

$$\textbf{Primal }(\Pi)\quad \boxed{\begin{array}{rl}\max & c\cdot x\\ \text{s.t.} & Ax\le b\\ & x\ge 0\end{array}} \iff \textbf{Dual }(\amalg)\quad \boxed{\begin{array}{rl}\min & y\cdot b\\ \text{s.t.} & yA\ge c\\ & y\ge 0\end{array}}$$

We can also write the dual linear program in exactly the same canonical form as the primal, by swapping the coefficient vector c and the objective vector b, negating both vectors, and replacing the constraint matrix A with its negative transpose.[4]

$$\textbf{Primal }(\Pi)\quad \boxed{\begin{array}{rl}\max & c\cdot x\\ \text{s.t.} & Ax\le b\\ & x\ge 0\end{array}} \iff \textbf{Dual }(\amalg)\quad \boxed{\begin{array}{rl}\max & -b^\top\cdot y^\top\\ \text{s.t.} & -A^\top y^\top\le -c\\ & y^\top\ge 0\end{array}}$$

Written in this form, it should be immediately clear that duality is an *involution*: The dual of the dual linear program $\amalg$ is identical to the primal linear program Π. The choice of which LP to call the 'primal' and which to call the 'dual' is totally arbitrary.[5]

The Fundamental Theorem of Linear Programming. *A linear program Π has an optimal solution x^* if and only if the dual linear program $\amalg$ has an optimal solution y^* such that $c\cdot x^* = y^*Ax^* = y^*\cdot b$.*

The weak form of this theorem is trivial to prove.

Weak Duality Theorem. *If x is a feasible solution for a canonical linear program Π and y is a feasible solution for its dual $\amalg$, then $c\cdot x \le yAx \le y\cdot b$.*

Proof: Because x is feasible for Π, we have $Ax \le b$. Since y is positive, we can multiply both sides of the inequality to obtain $yAx \le y\cdot b$. Conversely, y is feasible for $\amalg$ and x is positive, so $yAx \ge c\cdot x$. □

It immediately follows that if $c\cdot x = y\cdot b$, then x and y are optimal solutions to their respective linear programs. This is in fact a fairly common way to prove that we have the optimal value for a linear program.

[4]For the notational purists: In these formulations, x and b are column vectors, and y and c are row vectors. This is a somewhat nonstandard choice. Yes, that means the dot in $c\cdot x$ is redundant. Sue me.

[5]For historical reasons, maximization LPs tend to be called 'primal' and minimization LPs tend to be called 'dual'. This is a pointless religious tradition, nothing more. Duality is a relationship between LP problems, not a type of LP problem.

26.5 Duality Example

Before I prove the stronger duality theorem, let me first provide some intuition about where this duality thing comes from in the first place.[6] Consider the following linear programming problem:

$$\begin{array}{ll} \text{maximize} & 4x_1 + x_2 + 3x_3 \\ \text{subject to} & x_1 + 4x_2 \qquad \leq 2 \\ & 3x_1 - x_2 + x_3 \leq 4 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Let σ^* denote the optimum objective value for this LP. The feasible solution $x = (1,0,0)$ gives us a lower bound $\sigma^* \geq 4$. A different feasible solution $x = (0,0,3)$ gives us a better lower bound $\sigma^* \geq 9$. We could play this game all day, finding different feasible solutions and getting ever larger lower bounds. How do we know when we're done? Is there a way to prove an *upper* bound on σ^*?

In fact, there is. Let's multiply each of the constraints in our LP by a new non-negative scalar value y_i:

$$\begin{array}{ll} \text{maximize} & 4x_1 + x_2 + 3x_3 \\ \text{subject to} & y_1(x_1 + 4x_2 \qquad) \leq 2y_1 \\ & y_2(3x_1 - x_2 + x_3) \leq 4y_2 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Because each y_i is non-negative, we do not reverse any of the inequalities. Any feasible solution (x_1, x_2, x_3) must satisfy both of these inequalities, so it must also satisfy their sum:

$$(y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \leq 2y_1 + 4y_2.$$

Now suppose that each y_i is larger than the ith coefficient of the objective function:

$$y_1 + 3y_2 \geq 4, \qquad 4y_1 - y_2 \geq 1, \qquad y_2 \geq 3.$$

This assumption lets us derive an upper bound on the objective value of *any* feasible solution:

$$4x_1 + x_2 + 3x_3 \;\leq\; (y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \;\leq\; 2y_1 + 4y_2. \tag{$*$}$$

In particular, by plugging in the optimal solution (x_1^*, x_2^*, x_3^*) for the original LP, we obtain the following upper bound on σ^*:

$$\sigma^* \;=\; 4x_1^* + x_2^* + 3x_3^* \;\leq\; 2y_1 + 4y_2.$$

Now it's natural to ask how tight we can make this upper bound. How small can we make the expression $2y_1 + 4y_2$ without violating any of the inequalities we used to prove the upper bound? This is just another linear programming problem.

$$\begin{array}{ll} \text{minimize} & 2y_1 + 4y_2 \\ \text{subject to} & y_1 + 3y_2 \geq 4 \\ & 4y_1 - y_2 \geq 1 \\ & y_2 \geq 3 \\ & y_1, y_2 \geq 0 \end{array}$$

[6]This example is taken from Robert Vanderbei's excellent textbook *Linear Programming: Foundations and Extensions* [Springer, 2001], but the idea appears earlier in Jens Clausen's 1997 paper 'Teaching Duality in Linear Programming: The Multiplier Approach'.

In fact, this is precisely the dual of our original linear program! Moreover, inequality $(*)$ is just an instantiation of the Weak Duality Theorem.

26.6 Strong Duality

The Fundamental Theorem can be rephrased in the following form:

Strong Duality Theorem. *If x^* is an optimal solution for a canonical linear program Π, then there is an optimal solution y^* for its dual $\amalg$, such that $c \cdot x^* = y^*Ax^* = y^* \cdot b$.*

Proof (sketch): I'll prove the theorem only for ***non-degenerate*** linear programs, in which (a) the optimal solution (if one exists) is a unique vertex of the feasible region, and (b) at most d constraint hyperplanes pass through any point. These non-degeneracy assumptions are relatively easy to enforce in practice and can be removed from the proof at the expense of some technical detail. I will also prove the theorem only for the case $n \geq d$; the argument for under-constrained LPs is similar (if not simpler).

To develop some intuition, let's first consider the *very* special case where $x^* = (0, 0, \ldots, 0)$. Let e_i denote the ith standard basis vector, whose ith coordinate is 1 and all other coordinates are 0. Because $x_i^* = 0$ for all i, our non-degeneracy assumption implies the *strict* inequality $a_i \cdot x^* < b_i$ for all i. Thus, any sufficiently small ball around the origin does not intersect any other constraint hyperplane $a_i \cdot x = b_i$. Thus, for all i, and for any sufficiently small $\delta > 0$, the vector δe_i is feasible. Because x^* is the unique optimum, we must have $\delta c_i = c \cdot (\delta e_i) < c \cdot x^* = 0$. We conclude that $c_i < 0$ for all i.

Now let $y = (0, 0, \ldots, 0)$ as well. We immediately observe that $yA \geq c$ and $y \geq 0$; in other words, y is a *feasible* solution for the dual linear program $\amalg$. But $y \cdot b = 0 = c \cdot x^*$, so the weak duality theorem implies that y is an *optimal* solution to $\amalg$, and the proof is complete for this very special case.

Now let us consider the more general case. Let x^* be the optimal solution for the linear program Π; our non-degeneracy assumption implies that this solution is unique, and that exactly d of the n linear constraints are satisfied with equality. Without loss of generality (by permuting the constraints and possibly changing coordinates), we can assume that these are the first d constraints. Thus, we have

$$\begin{aligned} a_i \cdot x^* &= b_i && \text{for all } i \leq d, \\ a_i \cdot x^* &< b_i && \text{for all } i \geq d+1, \end{aligned}$$

where a_i denotes the ith row of A. Let $A_\bullet$ denote the $d \times d$ matrix containing the first d rows of A. Our non-degeneracy assumption implies that $A_\bullet$ has full rank, and thus has a well-defined inverse $V = A_\bullet^{-1}$.

Now define a vector $y \in \mathbb{R}^n$ by setting

$$\begin{aligned} y_j &:= c \cdot v^j && \text{for all } j \leq d, \\ y_j &:= 0 && \text{for all } j \geq d+1, \end{aligned}$$

where v^j denotes the jth column of $V = A_\bullet^{-1}$. Note that $a_i \cdot v^j = 0$ if $i \neq j$, and $a_i \cdot v^j = 1$ if $i = j$.

To simplify notation, let $y_\bullet = (y_1, y_2, \ldots, y_d)$ and let $b_\bullet = (b_1, b_2, \ldots, b_d) = A_\bullet x^*$. Because $y_i = 0$ for all $i \geq d+1$, we immediately have

$$y \cdot b = y_\bullet \cdot b_\bullet = cV b_\bullet = cA_\bullet^{-1} b_\bullet = c \cdot x^*$$

and

$$yA = y_\bullet A_\bullet = cVA_\bullet = cA_\bullet^{-1}A_\bullet = c.$$

The point x^* lies on exactly d constraint hyperplanes; moreover, any sufficiently small ball around x^* intersects *only* those d constraint hyperplanes. Consider the point $\tilde{x} = x^* - \varepsilon v^j$, for some index $1 \le j \le d$ and some sufficiently small $\varepsilon > 0$. We have $a_i \cdot \tilde{x} = a_i \cdot x^* - \varepsilon(a_i \cdot v^j) = b_i$ for all $i \neq j$, and $a_j \cdot \tilde{x} = a_j \cdot x^* - \varepsilon(a_j \cdot v^j) = b_j - \varepsilon < b_j$. Thus, $\tilde{x}$ is a feasible point for Π. Because x^* is the *unique* optimum for Π, we must have $c \cdot \tilde{x} = c \cdot x^* - \varepsilon(c \cdot v^j) < c \cdot x^*$. We conclude that $y_j = c \cdot v^j > 0$ for all j.

We have shown that $yA \ge c$ and $y \ge 0$, so y is a *feasible* solution for the dual linear program ⨿. We have also shown that $y \cdot b = c \cdot x^*$, so by the Weak Duality Theorem, y is also an *optimal* solution for ⨿, and the proof is complete! □

We can also give a useful geometric interpretation to the vector $y_\bullet \in \mathbb{R}^d$. Each linear equation $a_i \cdot x = b_i$ defines a hyperplane in $\mathbb{R}^d$ with normal vector a_i. The normal vectors $a_1, \dots, a_d$ are linearly independent (by non-degeneracy) and therefore describe a coordinate frame for the vector space $\mathbb{R}^d$. The definition of $y_\bullet$ implies that $c = y_\bullet A_\bullet = \sum_{i=1}^d y_i a_i$. In other words, ***$y_\bullet$ lists the coefficients of the objective vector c in the coordinate frame $a_1, \dots, a_d$.***

26.7 Complementary Slackness

Complementary Slackness Theorem. *Let x^* be an optimal solution to a canonical linear program Π, and let y^* be an optimal solution to its dual ⨿. Then for every index i, we have $y_i^* > 0$ if and only if $a_i \cdot x^* = b_i$. Symmetrically, for every index j, we have $x_j^* > 0$ if and only if $y^* \cdot a^j = c_j$.*

To be written

Exercises

1. (a) Describe how to transform any linear program written in general form into an equivalent linear program written in slack form.

$$\boxed{\begin{array}{lll} \text{maximize} & \displaystyle\sum_{j=1}^d c_j x_j & \\ \text{subject to} & \displaystyle\sum_{j=1}^d a_{ij} x_j \le b_i & \text{for each } i = 1 \,..\, p \\ & \displaystyle\sum_{j=1}^d a_{ij} x_j = b_i & \text{for each } i = p+1 \,..\, p+q \\ & \displaystyle\sum_{j=1}^d a_{ij} x_j \ge b_i & \text{for each } i = p+q+1 \,..\, n \end{array}} \Longrightarrow \boxed{\begin{array}{rl} \max & c \cdot x \\ \text{s.t.} & Ax = b \\ & x \ge 0 \end{array}}$$

(b) Describe precisely how to dualize a linear program written in slack form.

(c) Describe precisely how to dualize a linear program written in general form:

In all cases, keep the number of variables in the resulting linear program as small as possible.

2. A matrix $A = (a_{ij})$ is *skew-symmetric* if and only if $a_{ji} = -a_{ij}$ for all indices $i \neq j$; in particular, every skew-symmetric matrix is square. A canonical linear program $\max\{c \cdot x \mid Ax \leq b; x \geq 0\}$ is *self-dual* if the matrix A is skew-symmetric and the objective vector c is *equal* to the constraint vector b.

   (a) Prove that any self-dual linear program Π is syntactically equivalent to its dual program ∐.

   (b) Show that any linear program Π with d variables and n constraints can be transformed into a self-dual linear program with $n+d$ variables and $n+d$ constraints. The optimal solution to the self-dual program should include both the optimal solution for Π (in d of the variables) and the optimal solution for the dual program ∐ (in the other n variables).

3. (a) Give a linear-programming formulation of the *maximum-cardinality bipartite matching* problem. The input is a bipartite graph $G = (U \cup V; E)$, where $E \subseteq U \times V$; the output is the largest matching in G. Your linear program should have one variable for each edge.

   (b) Now dualize the linear program from part (a). What do the dual variables represent? What does the objective function represent? What problem is this!?

4. Give a linear-programming formulation of the *minimum-cost feasible circulation problem*. Here you are given a flow network whose edges have both capacities and costs, and your goal is to find a feasible circulation (flow with value 0) whose cost is as small as possible.

5. An *integer program* is a linear program with the additional constraint that the variables must take only integer values.

   (a) Prove that deciding whether an integer program has a feasible solution is NP-complete.

   (b) Prove that finding the optimal feasible solution to an integer program is NP-hard.

   [Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]

*6. *Helly's theorem* states that for any collection of convex bodies in $\mathbb{R}^d$, if every $d+1$ of them intersect, then there is a point lying in the intersection of all of them. Prove Helly's theorem for the special case where the convex bodies are halfspaces. Equivalently, show that if a system of linear inequalities $Ax \leq b$ does not have a solution, then we can select $d+1$ of the inequalities such that the resulting subsystem also does not have a solution. *[Hint: Construct a dual LP from the system by choosing a 0 cost vector.]*

7. Given points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ in the plane, the *linear regression problem* asks for real numbers a and b such that the line $y = ax + b$ fits the points as closely as possible, according to some criterion. The most common fit criterion is minimizing the L_2 *error*,

defined as follows:[7]

$$\varepsilon_2(a,b) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

But there are several other fit criteria, some of which can be optimized via linear programming.

(a) The L_1 *error* (or *total absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_1(a,b) = \sum_{i=1}^{n} |y_i - ax_i - b|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_1 error.

(b) The L_∞ *error* (or *maximum absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_\infty(a,b) = \max_{i=1}^{n} |y_i - ax_i - b|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_∞ error.

[7]This measure is also known as *sum of squared residuals*, and the algorithm to compute the best fit is normally called *(ordinary/linear) least squares fitting*.

Simplicibus itaque verbis gaudet Mathematica Veritas, cum etiam per se simplex sit Veritatis oratio. [And thus Mathematical Truth prefers simple words, because the language of Truth is itself simple.]

— Tycho Brahe (quoting Seneca (quoting Euripides))
Epistolarum astronomicarum liber primus (1596)

When a jar is broken, the space that was inside
Merges into the space outside.
In the same way, my mind has merged in God;
To me, there appears no duality.

— Sankara, *Viveka-Chudamani* (c. 700), translator unknown

*27 Linear Programming Algorithms

In this lecture, we'll see a few algorithms for actually solving linear programming problems. The most famous of these, the *simplex method*, was proposed by George Dantzig in 1947. Although most variants of the simplex algorithm performs well in practice, no deterministic simplex variant is known to run in sub-exponential time in the worst case.[1] However, if the dimension of the problem is considered a constant, there are several linear programming algorithms that run in *linear* time. I'll describe a particularly simple randomized algorithm due to Raimund Seidel.

My approach to describing these algorithms will rely much more heavily on geometric intuition than the usual linear-algebraic formalism. This works better for me, but your mileage may vary. For a more traditional description of the simplex algorithm, see Robert Vanderbei's excellent textbook *Linear Programming: Foundations and Extensions* [Springer, 2001], which can be freely downloaded (but not legally printed) from the author's website.

27.1 Bases, Feasibility, and Local Optimality

Consider the canonical linear program $\max\{c \cdot x \mid Ax \le b, x \ge 0\}$, where A is an $n \times d$ constraint matrix, b is an n-dimensional coefficient vector, and c is a d-dimensional objective vector. We will interpret this linear program geometrically as looking for the lowest point in a convex polyhedron in $\mathbb{R}^d$, described as the intersection of $n+d$ halfspaces. As in the last lecture, we will consider only *non-degenerate* linear programs: Every subset of d constraint hyperplanes intersects in a single point; at most d constraint hyperplanes pass through any point; and objective vector is linearly independent from any $d-1$ constraint vectors.

A ***basis*** is a subset of d constraints, which by our non-degeneracy assumption must be linearly independent. The ***location*** of a basis is the unique point x that satisfies all d constraints with equality; geometrically, x is the unique intersection point of the d hyperplanes. The ***value*** of a basis is $c \cdot x$, where x is the location of the basis. There are precisely $\binom{n+d}{d}$ bases. Geometrically, the set of constraint hyperplanes defines a decomposition of $\mathbb{R}^d$ into convex polyhedra; this cell decomposition is called the ***arrangement*** of the hyperplanes. Every subset of d hyperplanes (that is, every basis) defines a *vertex* of this arrangement (the location of the basis). I will use the words 'vertex' and 'basis' interchangeably.

[1]However, there are *randomized* variants of the simplex algorithm that run in subexponential *expected* time, most notably the RandomFacet algorithm analyzed by Gil Kalai in 1992, and independently by Jií Matoušek, Micha Sharir, and Emo Welzl in 1996. No randomized variant is known to run in polynomial time. In particular, in 2010, Oliver Friedmann, Thomas Dueholm Hansen, and Uri Zwick proved that the worst-case expected running time of RandomFacet is superpolynomial.

A basis is ***feasible*** if its location x satisfies all the linear constraints, or geometrically, if the point x is a vertex of the polyhedron. If there are no feasible bases, the linear program is ***infeasible***.

A basis is ***locally optimal*** if its location x is the optimal solution to the linear program with the same objective function and *only* the constraints in the basis. Geometrically, a basis is locally optimal if its location x is the lowest point in the intersection of those d halfspaces. A careful reading of the proof of the Strong Duality Theorem reveals that local optimality is the dual equivalent of feasibility; a basis is locally feasible for a linear program Π if and only if the same basis is feasible for the dual linear program $\amalg$. For this reason, locally optimal bases are sometimes also called ***dual feasible***. If there are no locally optimal bases, the linear program is ***unbounded***.[2]

Two bases are ***neighbors*** if they have $d-1$ constraints in common. Equivalently, in geometric terms, two vertices are neighbors if they lie on a *line* determined by some $d-1$ constraint hyperplanes. Every basis is a neighbor of exactly dn other bases; to change a basis into one of its neighbors, there are d choices for which constraint to remove and n choices for which constraint to add. The graph of vertices and edges on the boundary of the feasible polyhedron is a subgraph of the basis graph.

The Weak Duality Theorem implies that the value of every feasible basis is less than or equal to the value of every locally optimal basis; equivalently, every feasible vertex is higher than every locally optimal vertex. The Strong Duality Theorem implies that (under our non-degeneracy assumption), if a linear program has an optimal solution, it is the *unique* vertex that is both feasible and locally optimal. Moreover, the optimal solution is both the lowest feasible vertex and the highest locally optimal vertex.

27.2 The Primal Simplex Algorithm: Falling Marbles

From a geometric standpoint, Dantzig's simplex algorithm is very simple. The input is a set H of halfspaces; we want the lowest vertex in the intersection of these halfspaces.

```
SIMPLEX1(H):
  if ∩H = ∅
      return INFEASIBLE
  x ← any feasible vertex
  while x is not locally optimal
      ⟨⟨pivot downward, maintaining feasibility⟩⟩
      if every feasible neighbor of x is higher than x
          return UNBOUNDED
      else
          x ← any feasible neighbor of x that is lower than x
  return x
```

Let's ignore the first three lines for the moment. The algorithm maintains a feasible vertex x. At each so-called ***pivot*** operation, the algorithm moves to a *lower* vertex, so the algorithm never visits the same vertex more than once. Thus, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the feasible vertex x is locally optimal, and therefore the optimum vertex, or the feasible vertex x is not locally optimal but has no lower feasible neighbor, in which case the feasible region must be unbounded.

[2]For non-degenerate linear programs, the feasible region is unbounded in the objective direction if and only if no basis is locally optimal. However, there are degenerate linear programs with no locally optimal basis that are infeasible.

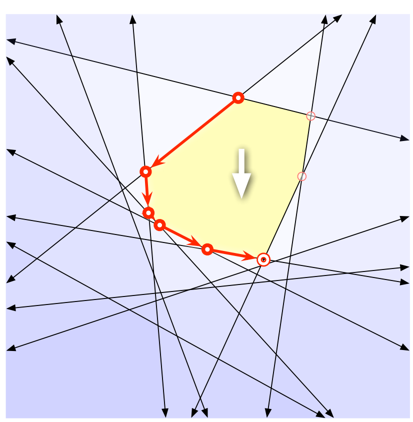
Notice that we have not specified *which* neighbor to choose at each pivot. Many different pivoting rules have been proposed, but for almost every known pivot rule, there is an input polyhedron that requires an exponential number of pivots under that rule. *No* pivoting rule is known that guarantees a polynomial number of pivots in the worst case, or even in expectation.[3]

27.3 The Dual Simplex Algorithm: Rising Bubbles

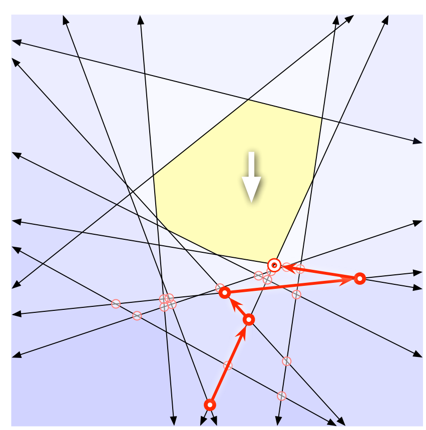
We can also geometrically interpret the execution of the simplex algorithm on the dual linear program ∐. Again, the input is a set H of halfspaces, and we want the lowest vertex in the intersection of these halfspaces. By the Strong Duality Theorem, this is the same as the *highest locally-optimal* vertex in the hyperplane arrangement.

```
SIMPLEX2(H):
  if there is no locally optimal vertex
      return UNBOUNDED
  x ← any locally optimal vertex
  while x is not feasbile
      ⟨⟨pivot upward, maintaining local optimality⟩⟩
      if every locally optimal neighbor of x is lower than x
          return INFEASIBLE
      else
          x ← any locally-optimal neighbor of x that is higher than x
  return x
```

Let's ignore the first three lines for the moment. The algorithm maintains a locally optimal vertex x. At each pivot operation, it moves to a *higher* vertex, so the algorithm never visits the same vertex more than once. Thus, the algorithm must halt after at most $\binom{n+d}{d}$ pivots. When the algorithm halts, either the locally optimal vertex x is feasible, and therefore the optimum vertex, or the locally optimal vertex x is not feasible but has no higher locally optimal neighbor, in which case the problem must be infeasible.



The primal simplex (falling marble) algorithm in action.



The dual simplex (rising bubble) algorithm in action.

From the standpoint of linear algebra, there is absolutely no difference between running SIMPLEX1 on any linear program Π and running SIMPLEX2 on the dual linear program ∐. The

[3] In 1957, Hirsch conjectured that for *any* linear programming instance with d variables and $n+d$ constraints, starting at any feasible basis, there is a sequence of ***at most n*** pivots that leads to the optimal basis. This long-standing conjecture was finally disproved in 2010 by Fransisco Santos, who described an counterexample with 43 variables, 86 facets, and diameter 44.

actual *code* is identical. The only difference between the two algorithms is how we interpret the linear algebra geometrically.

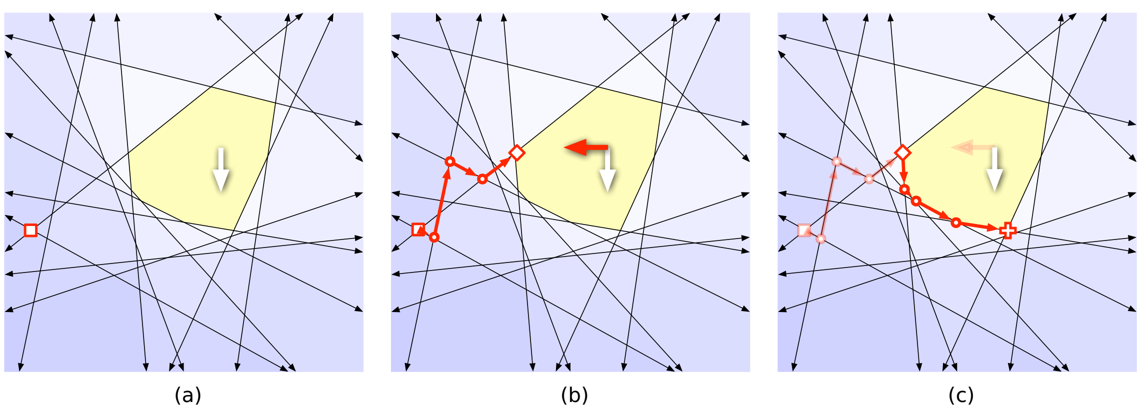
27.4 Computing the Initial Basis

To complete our description of the simplex algorithm, we need to describe how to find the initial vertex x. Our algorithm relies on the following simple observations.

First, the feasibility of a vertex does not depend at all on the choice of objective vector; a vertex is either feasible for every objective function or for none. No matter how we rotate the polyhedron, every feasible vertex stays feasible. Conversely (or by duality, equivalently), the local optimality of a vertex does not depend on the exact location of the d hyperplanes, but only on their normal directions and the objective function. No matter how we translate the hyperplanes, every locally optimal vertex stays locally optimal. In terms of the original matrix formulation, feasibility depends on A and b but not c, and local optimality depends on A and c but not b.

The second important observation is that *every* basis is locally optimal for *some* objective function. Specifically, it suffices to choose any vector that has a positive inner product with each of the normal vectors of the d chosen hyperplanes. Equivalently, we can make *any* basis feasible by translating the hyperplanes appropriately. Specifically, it suffices to translate the chosen d hyperplanes so that they pass through the origin, and then translate all the other halfspaces so that they strictly contain the origin.

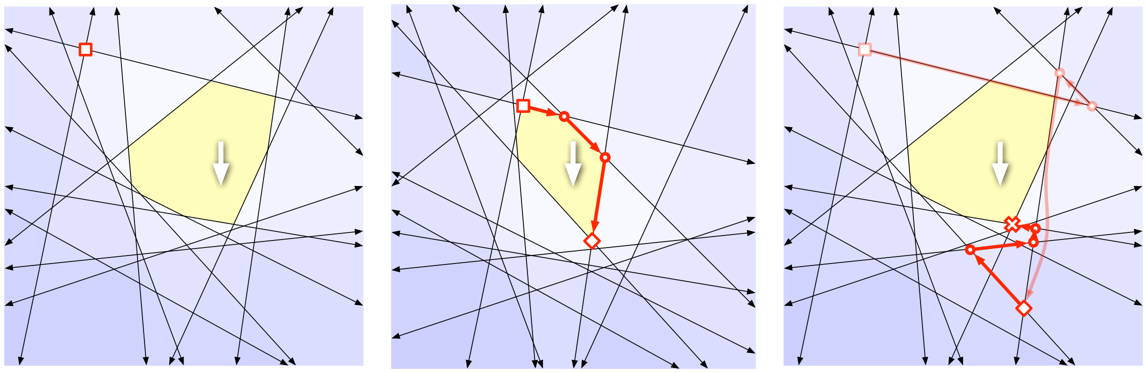
Our strategy for finding our initial feasible vertex is to choose *any* vertex, choose a new objective function that makes that vertex locally optimal, and then find the optimal vertex for *that* objective function by running the (dual) simplex algorithm. This vertex must be feasible, even after we restore the original objective function!



(a) Choose any basis. (b) Rotate objective to make it locally optimal, and pivot 'upward' to find a feasible basis. (c) Pivot downward to the optimum basis for the original objective.

Equivalently, to find an initial locally optimal vertex, we choose *any* vertex, translate the hyperplanes so that that vertex becomes feasible, and then find the optimal vertex for those translated constraints using the (primal) simplex algorithm. This vertex must be locally optimal, even after we restore the hyperplanes to their original locations!

Here are more complete descriptions of the simplex algorithm with this initialization rule, in both primal and dual forms. As usual, the input is a set H of halfspaces, and the algorithms either return the lowest vertex in the intersection of these halfspaces or report that no such vertex exists.



(a) Choose any basis. (b) Translate constraints to make it feasible, and pivot downward to find a locally optimal basis. (c) Pivot upward to the optimum basis for the original constraints.

```
SIMPLEX1(H):
  x ← any vertex
  H̃ ← any rotation of H that makes x locally optimal

  while x is not feasible
      if every locally optimal neighbor of x is lower (wrt H̃) than x
          return INFEASIBLE
      else
          x ← any locally optimal neighbor of x that is higher (wrt H̃) than x

  while x is not locally optimal
      if every feasible neighbor of x is higher than x
          return UNBOUNDED
      else
          x ← any feasible neighbor of x that is lower than x
  return x
```

```
SIMPLEX2(H):
  x ← any vertex
  H̃ ← any translation of H that makes x feasible

  while x is not locally optimal
      if every feasible neighbor of x is higher (wrt H̃) than x
          return UNBOUNDED
      else
          x ← any feasible neighbor of x that is lower (wrt H̃) than x

  while x is not feasible
      if every locally optimal neighbor of x is lower than x
          return INFEASIBLE
      else
          x ← any locally-optimal neighbor of x that is higher than x
  return x
```

27.5 Linear Expected Time for Fixed Dimensions

In most geometric applications of linear programming, the number of variables is a small constant, but the number of constraints may still be very large.

The input to the following algorithm is a set H of n halfspaces and a set B of b hyperplanes. (B stands for *basis*.) The algorithm returns the lowest point in the intersection of the halfspaces in H and the hyperplanes B. At the top level of recursion, B is empty. I will implicitly assume that the linear program is both feasible and bounded. (If necessary, we can guarantee boundedness by adding a single halfspace to H, and we can guarantee feasibility by adding a dimension.) A point x ***violates*** a constraint h if it is not contained in the corresponding halfspace.

```
SEIDELLP(H, B):
  if |B| = d
      return ⋂B
  if |H ∪ B| = d
      return ⋂(H ∪ B)
  h ← random element of H
  x ← SEIDELLP(H \ h, B)        (∗)
  if x violates h
      return SEIDELLP(H \ h, B ∪ ∂h)
  else
      return x
```

The point x recursively computed in line $(*)$ is the optimal solution if and only if the random halfspace h is *not* one of the d halfspaces that define the optimal solution. In other words, the probability of calling SEIDELLP$(H, B \cup h)$ is exactly $(d-b)/n$. Thus, we have the following recurrence for the expected number of recursive calls for this algorithm:

$$T(n,b) = \begin{cases} 1 & \text{if } b = d \text{ or } n + b = d \\ T(n-1,b) + \dfrac{d-b}{n} \cdot T(n-1,b+1) & \text{otherwise} \end{cases}$$

The recurrence is somewhat simpler if we write $\delta = d - b$:

$$T(n,\delta) = \begin{cases} 1 & \text{if } \delta = 0 \text{ or } n = \delta \\ T(n-1,\delta) + \dfrac{\delta}{n} \cdot T(n-1,\delta-1) & \text{otherwise} \end{cases}$$

It's easy to prove by induction that $T(n,\delta) = O(\delta!\,n)$:

$$\begin{aligned} T(n,\delta) &= T(n-1,\delta) + \frac{\delta}{n} \cdot T(n-1,\delta-1) \\ &\le \delta!\,(n-1) + \frac{\delta}{n}(\delta-1)! \cdot (n-1) && \text{[induction hypothesis]} \\ &= \delta!\,(n-1) + \delta!\,\frac{n-1}{n} \\ &\le \delta!\,n \end{aligned}$$

At the top level of recursion, we perform one violation test in $O(d)$ time. In each of the base cases, we spend $O(d^3)$ time computing the intersection point of d hyperplanes, and in the first base case, we spend $O(dn)$ additional time testing for violations. More careful analysis implies that the algorithm runs in $\boldsymbol{O(d! \cdot n)}$ ***expected time***.

Exercises

1. Fix a non-degenerate linear program in canonical form with d variables and $n+d$ constraints.

(a) Prove that every *feasible* basis has exactly d *feasible* neighbors.

(b) Prove that every *locally optimal* basis has exactly n *locally optimal* neighbors.

2. Suppose you have a subroutine that can solve linear programs in polynomial time, but only if they are both feasible and bounded. Describe an algorithm that solves *arbitrary* linear programs in polynomial time. Your algorithm should return an optimal solution if one exists; if no optimum exists, your algorithm should report that the input instance is UNBOUNDED or INFEASIBLE, whichever is appropriate. *[Hint: Add one variable and one constraint.]*

3. (a) Give an example of a non-empty polyhedron $Ax \leq b$ that is unbounded for every objective vector c.

   (b) Give an example of an infeasible linear program whose dual is also infeasible.

   In both cases, your linear program will be degenerate.

4. Describe and analyze an algorithm that solves the following problem in $O(n)$ time: Given n red points and n blue points in the plane, either find a line that separates every red point from every blue point, or prove that no such line exists.

5. The single-source shortest path problem can be formulated as a linear programming problem, with one variable d_v for each vertex $v \neq s$ in the input graph, as follows:

$$
\begin{array}{lrll}
\text{maximize} & \displaystyle\sum_v d_v & & \\
\text{subject to} & d_v \leq \ell_{s\to v} & & \text{for every edge } s \to v \\
& d_v - d_u \leq \ell_{u\to v} & & \text{for every edge } u \to v \text{ with } u \neq s \\
& d_v \geq 0 & & \text{for every vertex } v \neq s
\end{array}
$$

   This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of distances. Assume that the edge weights $\ell_{u\to v}$ are all non-negative and that there is a unique shortest path between any two vertices in the graph.

   (a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?

   (b) Show that in the optimal basis, every variable d_v is equal to the shortest-path distance from s to v.

   (c) Describe the primal simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?

   (d) Describe the dual simplex algorithm for the shortest-path linear program directly in terms of vertex distances. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?

   (e) Is Dijkstra's algorithm an instance of the simplex method? Justify your answer.

(f) Is Shimbel's algorithm an instance of the simplex method? Justify your answer.

6. The maximum (s,t)-flow problem can be formulated as a linear programming problem, with one variable $f_{u\to v}$ for each edge $u \to v$ in the input graph:

$$\begin{array}{lll}
\text{maximize} & \displaystyle\sum_w f_{s\to w} - \sum_u f_{u\to s} & \\
\text{subject to} & \displaystyle\sum_w f_{v\to w} - \sum_u f_{u\to v} = 0 & \text{for every vertex } v \neq s,t \\
& f_{u\to v} \leq c_{u\to v} & \text{for every edge } u \to v \\
& f_{u\to v} \geq 0 & \text{for every edge } u \to v
\end{array}$$

This problem asks you to describe the behavior of the simplex algorithm on this linear program in terms of flows.

(a) What is a basis for this linear program? What is a feasible basis? What is a locally optimal basis?

(b) Show that the optimal basis represents a maximum flow.

(c) Describe the primal simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?

(d) Describe the dual simplex algorithm for the flow linear program directly in terms of flows. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?

(e) Is the Ford-Fulkerson augmenting path algorithm an instance of the simplex method? Justify your answer. *[Hint: There is a one-line argument.]*

7. (a) Formulate the minimum spanning tree problem as an instance of linear programming. Try to minimize the number of variables and constraints.

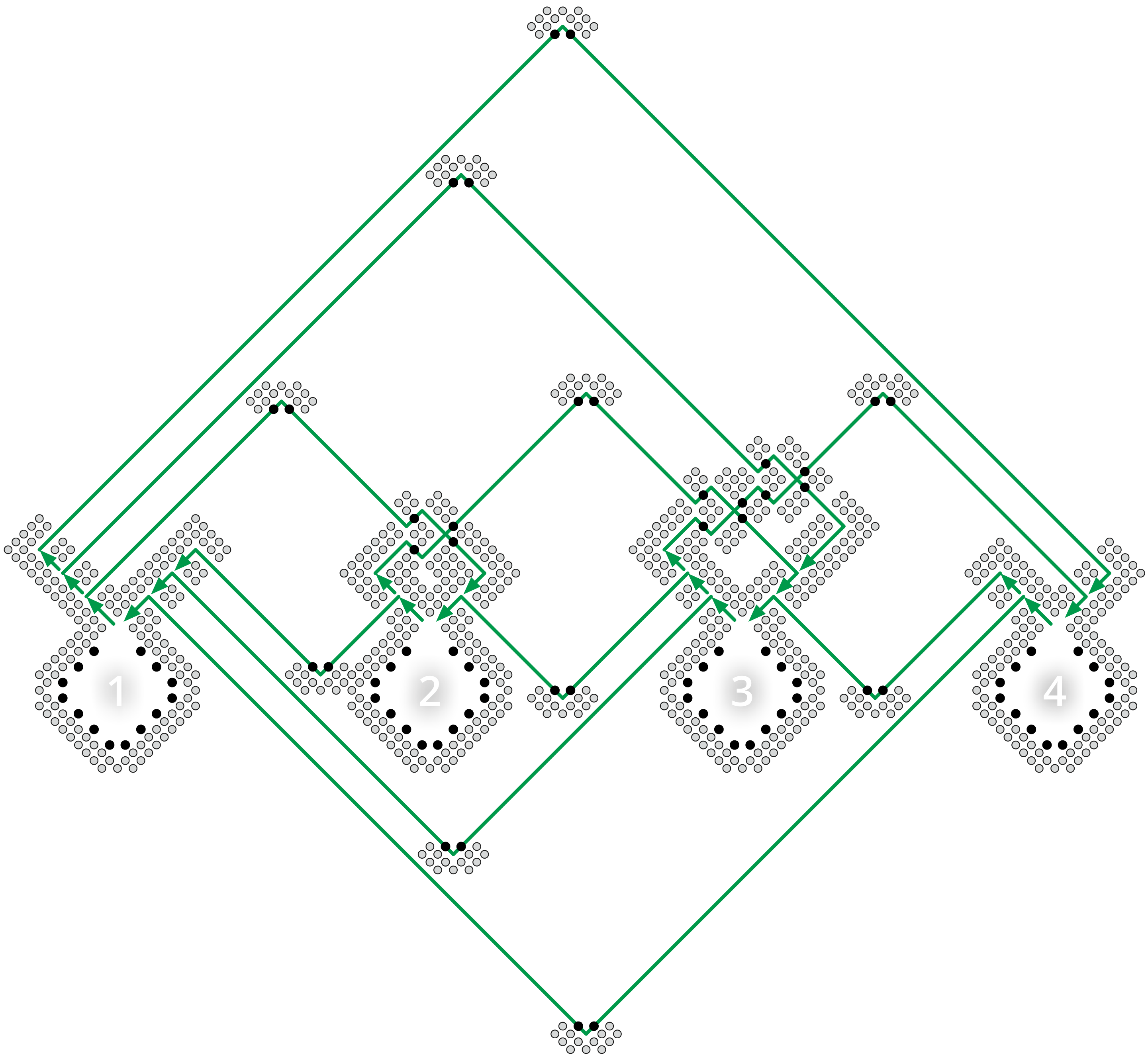
(b) In your MST linear program, what is a basis? What is a feasible basis? What is a locally optimal basis?

(c) Describe the primal simplex algorithm for your MST linear program directly in terms of the input graph. In particular, what does it mean to pivot from a feasible basis to a neighboring feasible basis, and how can we execute such a pivot quickly?

(d) Describe the dual simplex algorithm for your MST linear program directly in terms of the input graph. In particular, what does it mean to pivot from a locally optimal basis to a neighboring locally optimal basis, and how can we execute such a pivot quickly?

(e) Which of the classical MST algorithms (Borvka, Jarník, Kruskal, reverse greedy), if any, are instances of the simplex method? Justify your answer.

Hardness



It was a Game called Yes and No, where Scrooge's nephew had to think of something, and the rest must find out what; he only answering to their questions yes or no, as the case was. The brisk fire of questioning to which he was exposed, elicited from him that he was thinking of an animal, a live animal, rather a disagreeable animal, a savage animal, an animal that growled and grunted sometimes, and talked sometimes, and lived in London, and walked about the streets, and wasn't made a show of, and wasn't led by anybody, and didn't live in a menagerie, and was never killed in a market, and was not a horse, or an ass, or a cow, or a bull, or a tiger, or a dog, or a pig, or a cat, or a bear. At every fresh question that was put to him, this nephew burst into a fresh roar of laughter; and was so inexpressibly tickled, that he was obliged to get up off the sofa and stamp. At last the plump sister, falling into a similar state, cried out :

"I have found it out! I know what it is, Fred! I know what it is!"

"What is it?" cried Fred.

"It's your Uncle Scro-o-o-o-oge!"

Which it certainly was. Admiration was the universal sentiment, though some objected that the reply to "Is it a bear?" ought to have been "Yes;" inasmuch as an answer in the negative was sufficient to have diverted their thoughts from Mr Scrooge, supposing they had ever had any tendency that way.

— Charles Dickens, *A Christmas Carol* (1843)

28 Lower Bounds

28.1 Huh? Whuzzat?

So far in this class we've been developing algorithms and data structures to solve certain problems as quickly as possible. Starting with this lecture, we'll turn the tables, by proving that certain problems *cannot* be solved as quickly as we might like them to be.

Let $T_A(X)$ denote the running time of algorithm A given input X. For most of the semester, we've been concerned with the the worst-case running time of A as a function of the input size:

$$T_A(n) := \max_{|X|=n} T_A(X).$$

The worst-case complexity of a *problem* Π is the worst-case running time of the *fastest* algorithm for solving it:

$$T_\Pi(n) := \min_{A \text{ solves } \Pi} T_A(n) \;=\; \min_{A \text{ solves } \Pi} \max_{|X|=n} T_A(X).$$

Any algorithm A that solves Π immediately implies an *upper bound* on the complexity of Π; the inequality $\boldsymbol{T_\Pi(n) \le T_A(n)}$ follows directly from the definition of T_Π. Just as obviously, faster algorithms give us better (smaller) upper bounds. In other words, whenever we give a running time for an algorithm, what we're really doing—and what most computer scientists devote their entire careers doing[1]—is bragging about how *easy* some problem is.

Now, instead of bragging about how easy problems are, we will argue that certain problems are *hard*, by proving *lower bounds* on their complexity. This is considerably harder than proving

[1]This sometimes leads to long sequences of results that sound like an obscure version of "Name that Tune":

Lennes: "I can triangulate that polygon in $O(n^2)$ time."
Shamos: "I can triangulate that polygon in $O(n \log n)$ time."
Tarjan: "I can triangulate that polygon in $O(n \log\log n)$ time."
Seidel: "I can triangulate that polygon in $O(n \log^* n)$ time." [Audience gasps.]
Chazelle: "I can triangulate that polygon in $O(n)$ time." [Audience gasps and applauds.]
"Triangulate that polygon!"

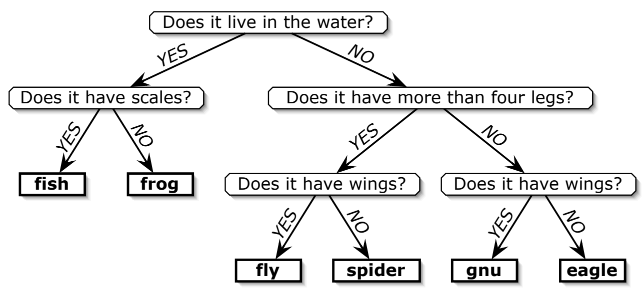
an upper bound, because it's no longer enough to examine a single algorithm. To prove an inequality of the form $T_\Pi(n) = \Omega(f(n))$, we must prove that ***every*** algorithm that solves Π has a worst-case running time $\Omega(f(n))$, or equivalently, that ***no*** algorithm runs in $o(f(n))$ time.

28.2 Decision Trees

Unfortunately, there is no formal definition of the phrase 'all algorithms'![2] So when we derive lower bounds, we first have to specify *precisely* what kinds of algorithms we will consider and *precisely* how to measure their running time. This specification is called a ***model of computation***.

One rather powerful model of computation—and the only model we'll talk about in this lecture—is the ***decision tree*** model. A decision tree is, as the name suggests, a tree. Each internal node in the tree is labeled by a *query*, which is just a question about the input. The edges out of a node correspond to the possible answers to that node's query. Each leaf of the tree is labeled with an *output*. To compute with a decision tree, we start at the root and follow a path down to a leaf. At each internal node, the answer to the query tells us which node to visit next. When we reach a leaf, we output its label.

For example, the guessing game where one person thinks of an animal and the other person tries to figure it out with a series of yes/no questions can be modeled as a decision tree. Each internal node is labeled with a question and has two edges labeled 'yes' and 'no'. Each leaf is labeled with an animal.

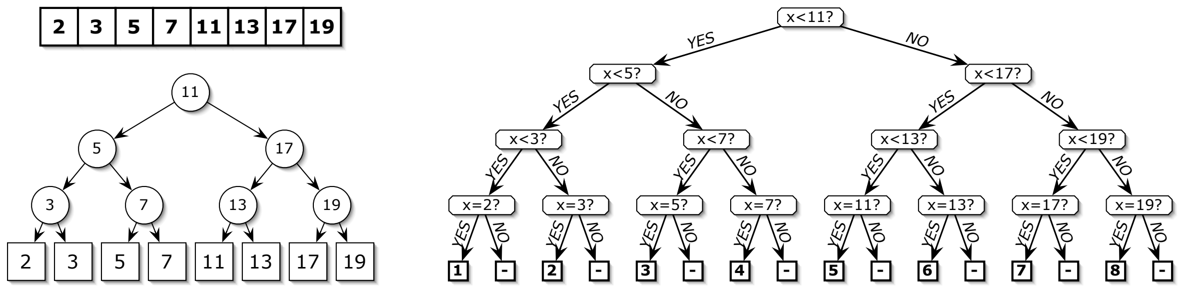


A decision tree to choose one of six animals.

Here's another simple and familiar example, called the ***dictionary problem***. Let A be a fixed array with n numbers. Suppose we want to determine, given a number x, the position of x in the array A, if any. One solution to the dictionary problem is to sort A (remembering every element's original position) and then use binary search. The (implicit) binary search tree can be used almost directly as a decision tree. Each internal node in the *search* tree stores a key k; the corresponding node in the *decision* tree stores the question 'Is $x < k$?'. Each leaf in the *search* tree stores some value $A[i]$; the corresponding node in the *decision* tree asks 'Is $x = A[i]$?' and has two leaf children, one labeled 'i' and the other 'none'.

We *define* the running time of a decision tree algorithm for a given input to be the number of queries in the path from the root to the leaf. For example, in the 'Guess the animal' tree above,

[2]Complexity-theory ~~snobs~~ purists sometimes argue that 'all algorithms' is just a synonym for 'all Turing machines'. This is utter nonsense; Turing machines are just another model of computation. Turing machines *might* be a reasonable abstraction of *physically realizable* computation—that's the Church-Turing thesis—but it has a few problems. First, computation is an abstract mathematical process, not a physical process. Algorithms that use physically unrealistic components (like exact real numbers, or unbounded memory) are still mathematically well-defined and still provide useful intuition about real-world computation. Moreover, Turing machines don't accurately reflect the complexity of physically realizable algorithms, because (for example) they can't do arithmetic or access arbitrary memory locations in constant time. At best, they estimate algorithmic complexity up to polynomial factors (although even that is unknown).



Left: A binary search tree for the first eight primes.
Right: The corresponding binary decision tree for the dictionary problem (- = 'none').

$T(\text{frog}) = 2$. Thus, the worst-case running time of the algorithm is just the depth of the tree. This definition ignores other kinds of operations that the algorithm might perform that have nothing to do with the queries. (Even the most efficient binary search problem requires more than one machine instruction per comparison!) But the number of decisions is certainly a *lower bound* on the actual running time, which is good enough to prove a lower bound on the complexity of a problem.

Both of the examples describe *binary* decision trees, where every query has only two answers. We may sometimes want to consider decision trees with higher degree. For example, we might use queries like 'Is x greater than, equal to, or less than y?' or 'Are these three points in clockwise order, colinear, or in counterclockwise order?' A *k-ary* decision tree is one where every query has (at most) k different answers. **From now on, I will only consider k-ary decision trees where k is a constant.**

28.3 Information Theory

Most lower bounds for decision trees are based on the following simple observation: ***The answers to the queries must give you enough information to specify any possible output.*** If a problem has N different outputs, then obviously any decision tree must have at least N leaves. (It's possible for several leaves to specify the same output.) Thus, if every query has at most k possible answers, then the depth of the decision tree must be at least $\lceil \log_k N \rceil = \Omega(\log N)$.

Let's apply this to the dictionary problem for a set S of n numbers. Since there are $n+1$ possible outputs, any decision tree must have at least $n+1$ leaves, and thus any decision tree must have depth at least $\lceil \log_k(n+1) \rceil = \Omega(\log n)$. So the complexity of the dictionary problem, in the decision-tree model of computation, is $\Omega(\log n)$. This matches the upper bound $O(\log n)$ that comes from a perfectly-balanced binary search tree. That means that the standard binary search algorithm, which runs in $O(\log n)$ time, is *optimal*—there is no faster algorithm in this model of computation.

28.4 But wait a second...

We can solve the membership problem in $O(1)$ expected time using hashing. Isn't this inconsistent with the $\Omega(\log n)$ lower bound?

No, it isn't. The reason is that hashing involves a query with more than a constant number of outcomes, specifically 'What is the hash value of x?' In fact, if we don't restrict the degree of the decision tree, we can get constant running time even without hashing, by using the obviously unreasonable query 'For which index i (if any) is $A[i] = x$?'. No, I am *not* cheating — remember that the decision tree model allows us to ask *any* question about the input!

This example illustrates a common theme in proving lower bounds: *choosing the right model of computation is absolutely crucial.* If you choose a model that is too powerful, the problem you're studying may have a completely trivial algorithm. On the other hand, if you consider more restrictive models, the problem may not be solvable at all, in which case any lower bound will be meaningless! (In this class, we'll just tell you the right model of computation to use.)

28.5 Sorting

Now let's consider the classical *sorting* problem — Given an array of n numbers, arrange them in increasing order. Unfortunately, decision trees don't have any way of describing moving data around, so we have to rephrase the question slightly:

> Given a sequence $\langle x_1, x_2, \ldots, x_n\rangle$ of n distinct numbers, find the permutation π such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$.

Now a k-ary decision-tree lower bound is immediate. Since there are $n!$ possible permutations π, any decision tree for sorting must have at least $n!$ leaves, and so must have depth $\Omega(\log(n!))$. To simplify the lower bound, we apply *Stirling's approximation*

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1+\Theta\left(\frac{1}{n}\right)\right) > \left(\frac{n}{e}\right)^n.$$

This gives us the lower bound

$$\left\lceil \log_k(n!)\right\rceil > \left\lceil \log_k\left(\frac{n}{e}\right)^n\right\rceil = \left\lceil n\log_k n - n\log_k e\right\rceil = \Omega(n\log n).$$

This matches the $O(n\log n)$ upper bound that we get from mergesort, heapsort, or quicksort, so those algorithms are optimal. The decision-tree complexity of sorting is $\Theta(n\log n)$.

Well... we're not quite done. In order to say that those algorithms are optimal, we have to demonstrate that they fit into our model of computation. A few minutes of thought will convince you that they can be described as a special type of decision tree called a *comparison* tree, where every query is of the form 'Is x_i bigger or smaller than x_j?' These algorithms treat any two input sequences exactly the same way as long as the same comparisons produce exactly the same results. This is a feature of any comparison tree. In other words, *the actual input values don't matter, only their order*. Comparison trees describe almost all well-known sorting algorithms: bubble sort, selection sort, insertion sort, shell sort, quicksort, heapsort, mergesort, and so forth—but *not* radix sort or bucket sort.

28.6 Finding the Maximum and Adversaries

Finally let's consider the ***maximum problem***: Given an array X of n numbers, find its largest entry. Unfortunately, there's no hope of proving a lower bound in this formulation, since there are an infinite number of possible answers, so let's rephrase it slightly.

> Given a sequence $\langle x_1, x_2, \ldots, x_n\rangle$ of n distinct numbers, find the index m such that x_m is the largest element in the sequence.

We can get an upper bound of $n-1$ comparisons in several different ways. The easiest is probably to start at one end of the sequence and do a linear scan, maintaining a current maximum. Intuitively, this seems like the best we can do, but the information-theoretic bound is only $\lceil \log_2 n \rceil$. And in fact, this bound is tight! We can locate the maximum element by asking only $\lceil \log_2 n \rceil$ 'unreasonable' questions like "Is the index of the maximum element odd?" No, this is *not* cheating—the decision tree model allows *arbitrary* questions.

To prove a non-trivial lower bound for this problem, we must do two things. First, we need to consider a more reasonable model of computation, by restricting the kinds of questions the algorithm is allowed to ask. We will consider the ***comparison tree model***, where every query must have the form "Is $x_i > x_j$?". Since most algorithms[3] for finding the maximum rely on comparisons to make control-flow decisions, this does not seem like an unreasonable restriction.

Second, we will use something called an ***adversary argument***. The idea is that an all-powerful malicious adversary *pretends* to choose an input for the algorithm. When the algorithm asks a question about the input, the adversary answers in whatever way will make the algorithm do the most work. If the algorithm does not ask enough queries before terminating, then there will be several different inputs, each consistent with the adversary's answers, that should result in different outputs. In this case, whatever the algorithm outputs, the adversary can 'reveal' an input that is consistent with its answers, but contradicts the algorithm's output, and then claim that that was the input that he was using all along.

For the maximum problem, the adversary originally pretends that $x_i = i$ for all i, and answers all comparison queries accordingly. Whenever the adversary reveals that $x_i < x_j$, he *marks* x_i as an item that the algorithm knows (or should know) is not the maximum element. At most one element x_i is marked after each comparison. Note that x_n is never marked. If the algorithm does less than $n-1$ comparisons before it terminates, the adversary must have at least one other unmarked element $x_k \neq x_n$. In this case, the adversary can change the value of x_k from k to $n+1$, making x_k the largest element, without being inconsistent with any of the comparisons that the algorithm has performed. In other words, the algorithm cannot tell that the adversary has cheated. However, x_n is the maximum element in the original input, and x_k is the largest element in the modified input, so the algorithm cannot possibly give the correct answer for both cases. Thus, in order to be correct, any algorithm must perform at least $n-1$ comparisons.

The adversary argument we described has two very important properties. First, no algorithm can distinguish between a malicious adversary and an honest user who actually chooses an input in advance and answers all queries truthfully. But much more importantly, **the adversary makes *absolutely no* assumptions about the order in which the algorithm performs comparisons.** The adversary forces *any* comparison-based algorithm[4] to either perform $n-1$ comparisons, or to give the wrong answer for at least one input sequence.

Exercises

0. Simon bar Kokhba thinks of an integer between 1 and 1,000,000 (or so he claims). You are trying to determine his number by asking as few yes/no questions as possible. How many yes/no questions are required to determine Simon's number in the worst case? Give both an upper bound (supported by an algorithm) and a lower bound.

1. Consider the following *multi-dictionary* problem. Let $A[1..n]$ be a fixed array of distinct integers. Given an array $X[1..k]$, we want to find the position (if any) of each integer

[3]but not all—see Exercise 4

[4]In fact, the $n-1$ lower bound for finding the maximum holds in a more powerful model called *algebraic* decision trees, which are binary trees where every query is a comparison between two polynomial functions of the input values, such as 'Is $x_1^2 - 3x_2x_3 + x_4^{17}$ bigger or smaller than $5 + x_1x_3^5x_5^2 - 2x_7^{42}$?'

$X[i]$ in the array A. In other words, we want to compute an array $I[1..k]$ where for each i, either $I[i] = 0$ (so zero means 'none') or $A[I[i]] = X[i]$. Determine the *exact* complexity of this problem, as a function of n and k, in the binary decision tree model.

2. We say that an array $A[1..n]$ is *k-sorted* if it can be divided into k blocks, each of size n/k, such that the elements in each block are larger than the elements in earlier blocks, and smaller than elements in later blocks. The elements within each block need not be sorted.

   For example, the following array is 4-sorted:

1	2	4	3	7	6	8	5	10	11	9	12	15	13	16	14

   (a) Describe an algorithm that k-sorts an arbitrary array in $O(n \log k)$ time.

   (b) Prove that any comparison-based k-sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst case.

   (c) Describe an algorithm that completely sorts an already k-sorted array in $O(n \log(n/k))$ time.

   (d) Prove that any comparison-based algorithm to completely sort a k-sorted array requires $\Omega(n \log(n/k))$ comparisons in the worst case.

   In all cases, you can assume that n/k is an integer.

3. Recall the nuts-and-bolts problem from the lecture on randomized algorithms. We are given n bolts and n nuts of different sizes, where each bolt exactly matches one nut. Our goal is to find the matching nut for each bolt. The nuts and bolts are too similar to compare directly; however, we can test whether any nut is too big, too small, or the same size as any bolt.

   (a) Prove that in the worst case, $\Omega(n \log n)$ nut-bolt tests are required to correctly match up the nuts and bolts.

   (b) Now suppose we would be happy to find *most* of the matching pairs. Prove that in the worst case, $\Omega(n \log n)$ nut-bolt tests are required even to find $n/2$ arbitrary matching nut-bolt pairs.

   ⋆(c) Prove that in the worst case, $\Omega(n + k \log n)$ nut-bolt tests are required to find k arbitrary matching pairs. *[Hint: Use an adversary argument for the $\Omega(n)$ term.]*

   ⋆(d) Describe a randomized algorithm that finds k matching nut-bolt pairs in $O(n + k \log n)$ expected time.

⋆4. Suppose you want to determine the largest number in an n-element set $X = \{x_1, x_2, \ldots, x_n\}$, where each element x_i is an integer between 1 and $2^m - 1$. Describe an algorithm that solves this problem in $O(n + m)$ steps, where at each step, your algorithm compares one of the elements x_i with a *constant*. In particular, your algorithm must never actually compare two elements of X! *[Hint: Construct and maintain a nested set of 'pinning intervals' for the numbers that you have not yet removed from consideration, where each interval but the largest is either the upper half or lower half of the next larger block.]*

An adversary means opposition and competition,
but not having an adversary means grief and loneliness.

— Zhuangzi (Chuang-tsu) c. 300 BC

It is possible that the operator could be hit by an asteroid and your $20 could fall off his cardboard box and land on the ground, and while you were picking it up, $5 could blow into your hand. You therefore could win $5 by a simple twist of fate.

— Penn Jillette, explaining how to win at Three-Card Monte (1999)

29 Adversary Arguments

29.1 Three-Card Monte

Until Times Square was turned into a glitzy sanitized tourist trap, you could often find dealers stealing tourists' money using a game called "Three Card Monte" or "Spot the Lady". The dealer show the tourist three cards, say the Queen of Hearts, the two of spades, and three of clubs. The dealer shuffles the cards face down on a table (usually slowly enough that the tourist can follow the Queen), and then asks the tourist to bet on which card is the Queen. In principle, the tourist's odds of winning are at least one in three, more if the tourist was carefully watching the movement of the cards.

In practice, however, the tourist *never* wins, because the dealer cheats. The dealer actually holds at least *four* cards; before he even starts shuffling the cards, the dealer palms the queen or sticks it up his sleeve. No matter what card the tourist bets on, the dealer turns over a black card (which might be the two of clubs, but most tourists won't notice that wasn't one o the original cards). If the tourist gives up, the dealer slides the queen under one of the cards and turns it over, showing the tourist 'where the queen was all along'. If the dealer is really good, the tourist won't see the dealer changing the cards and will think maybe the queen *was* there all along and he just wasn't smart enough to figure that out. As long as the dealer doesn't reveal all the black cards at once, the tourist has no way to prove that the dealer cheated![1]

29.2 n-Card Monte

Now let's consider a similar game, but with an algorithm acting as the tourist and with bits instead of cards. Suppose we have an array of n bits and we want to determine if any of them is a 1. Obviously we can figure this out by just looking at every bit, but can we do better? Is there maybe some complicated tricky algorithm to answer the question "Any ones?" without looking at every bit? Well, of course not, but how do we prove it?

The simplest proof technique is called an *adversary* argument. The idea is that an all-powerful malicious adversary (the dealer) *pretends* to choose an input for the algorithm (the tourist). When the algorithm wants looks at a bit (a card), the adversary sets that bit to whatever value will make the algorithm do the most work. If the algorithm does not look at enough bits before terminating, then there will be several different inputs, each consistent with the bits already seen,

[1]Even if the dealer is a sloppy magician, he'll cheat anyway. The dealer is almost always surrounded by shills; these are the "tourists" who look like they're actually winning, who turn over cards when the dealer "isn't looking", who casually mention how easy the game is to win, and so on. The shills physically protect the dealer from any angry tourists who notice the dealer cheating, and shake down any tourists who refuse to pay after making a bet. Really, you ***cannot*** win this game, ***ever***.

the should result in different outputs. Whatever the algorithm outputs, the adversary can 'reveal' an input that is has all the examined bits but contradicts the algorithm's output, and then claim that that was the input that he was using all along. Since the only information the algorithm has is the set of bits it examined, the algorithm cannot distinguish between a malicious adversary and an honest user who actually chooses an input in advance and answers all queries truthfully.

For the n-card monte problem, the adversary originally pretends that the input array is all zeros—whenever the algorithm looks at a bit, it sees a 0. Now suppose the algorithms stops before looking at all three bits. If the algorithm says 'No, there's no 1,' the adversary changes one of the unexamined bits to a 1 and shows the algorithm that it's wrong. If the algorithm says 'Yes, there's a 1,' the adversary reveals the array of zeros and again proves the algorithm wrong. Either way, the algorithm cannot tell that the adversary has cheated.

One absolutely crucial feature of this argument is that *the adversary makes absolutely* ***no*** *assumptions about the algorithm*. The adversary strategy can't depend on some predetermined order of examining bits, and it doesn't care about anything the algorithm might or might not do when it's not looking at bits. *Any* algorithm that doesn't examine every bit falls victim to the adversary.

29.3 Finding Patterns in Bit Strings

Let's make the problem a little more complicated. Suppose we're given an array of n bits and we want to know if it contains the substring 01, a zero followed immediately by a one. Can we answer this question without looking at every bit?

It turns out that if n is odd, we *don't* have to look at all the bits. First we look the bits in every even position: $B[2], B[4], \ldots, B[n-1]$. If we see $B[i] = 0$ and $B[j] = 1$ for any $i < j$, then we know the pattern 01 is in there somewhere—starting at the last 0 before $B[j]$—so we can stop without looking at any more bits. If we see only 1s followed by 0s, we don't have to look at the bit between the last 0 and the first 1. If every even bit is a 0, we don't have to look at $B[1]$, and if every even bit is a 1, we don't have to look at $B[n]$. In the worst case, our algorithm looks at only $n-1$ of the n bits.

But what if n is even? In that case, we can use the following adversary strategy to show that any algorithm *does* have to look at every bit. The adversary will attempt to produce an 'input' string B *without* the substring 01; all such strings have the form $11\ldots100\ldots0$. The adversary maintains two indices ℓ and r and pretends that the prefix $B[1..\ell]$ contains only 1s and the suffix $B[r..n]$ contains only 0s. Initially $\ell = 0$ and $r = n+1$.

$$\mathbf{11111}\underset{\substack{\uparrow \\ \boldsymbol{\ell}}}{\mathbf{1}}\square\square\square\square\square\square\underset{\substack{\uparrow \\ \boldsymbol{r}}}{\mathbf{0}}\mathbf{000}$$

What the adversary is thinking; □ represents an unknown bit.

The adversary maintains the invariant that $r - \ell$, the length of the undecided portion of the 'input' string, is even. When the algorithm looks at a bit between ℓ and r, the adversary chooses whichever value preserves the parity of the intermediate chunk of the array, and then moves either ℓ or r. Specifically, here's what the adversary does when the algorithm examines bit $B[i]$. (Note that I'm specifying the adversary strategy as an algorithm!)

```
HIDE01(i):
  if i ≤ ℓ
      B[i] ← 1
  else if i ≥ r
      B[i] ← 0
  else if i − ℓ is even
      B[i] ← 0
      r ← i
  else
      B[i] ← 1
      ℓ ← i
```

It's fairly easy to prove that this strategy forces the algorithm to examine every bit. If the algorithm doesn't look at every bit to the right of r, the adversary could replace some unexamined bit with a 1. Similarly, if the algorithm doesn't look at every bit to the left of ℓ, the adversary could replace some unexamined bit with a zero. Finally, if there are any unexamined bits between ℓ and r, there must be at least two such bits (since $r - \ell$ is always even) and the adversary can put a 01 in the gap.

In general, we say that a bit pattern is *evasive* if we have to look at every bit to decide if a string of n bits contains the pattern. So the pattern 1 is evasive for all n, and the pattern 01 is evasive if and only if n is even. It turns out that the *only* patterns that are evasive for *all* values of n are the one-bit patterns 0 and 1.

29.4 Evasive Graph Properties

Another class of problems for which adversary arguments give good lower bounds is graph problems where the graph is represented by an adjacency matrix, rather than an adjacency list. Recall that the adjacency matrix of an undirected n-vertex graph $G = (V, E)$ is an $n \times n$ matrix A, where $A[i, j] = \big[(i, j) \in E\big]$. We are interested in deciding whether an undirected graph has or does not have a certain *property*. For example, is the input graph connected? Acyclic? Planar? Complete? A tree? We call a graph property *evasive* if we have to look look at all $\binom{n}{2}$ entries in the adjacency matrix to decide whether a graph has that property.

An obvious example of an evasive graph property is *emptiness*: Does the graph have any edges at all? We can show that emptiness is evasive using the following simple adversary strategy. The adversary maintains *two* graphs E and G. E is just the empty graph with n vertices. Initially G is the complete graph on n vertices. Whenever the algorithm asks about an edge, the adversary removes that edge from G (unless it's already gone) and answers 'no'. If the algorithm terminates without examining every edge, then G is not empty. Since both G and E are consistent with all the adversary's answers, the algorithm must give the wrong answer for one of the two graphs.

29.5 Connectedness Is Evasive

Now let me give a more complicated example, *connectedness*. Once again, the adversary maintains two graphs, Y and M ('yes' and 'maybe'). Y contains all the edges that the algorithm knows are definitely in the input graph. M contains all the edges that the algorithm thinks *might* be in the input graph, or in other words, all the edges of Y plus all the unexamined edges. Initially, Y is empty and M is complete.

Here's the strategy that adversary follows when the adversary asks whether the input graph contains the edge e. I'll assume that whenever an algorithm examines an edge, it's in M but not in Y; in other words, algorithms never ask about the same edge more than once.

```
HIDECONNECTEDNESS(e):
  if M \ {e} is connected
      remove (i, j) from M
      return 0
  else
      add e to Y
      return 1
```

Notice that the graphs Y and M are both consistent with the adversary's answers at all times. The adversary strategy maintains a few other simple invariants.

- **Y is a subgraph of M.** This is obvious.
- **M is connected.** This is also obvious.
- **If M has a cycle, none of its edges are in Y.** If M has a cycle, then deleting any edge in that cycle leaves M connected.
- **Y is acyclic.** This follows directly from the previous invariant.
- **If $Y \neq M$, then Y is disconnected.** The only connected acyclic graph is a tree. Suppose Y is a tree and some edge e is in M but not in Y. Then there is a cycle in M that contains e, all of whose other edges are in Y. This violated our third invariant.

We can also think about the adversary strategy in terms of minimum spanning trees. Recall the anti-Kruskal algorithm for computing the *maximum* spanning tree of a graph: Consider the edges one at a time in increasing order of length. If removing an edge would disconnect the graph, declare it part of the spanning tree (by adding it to Y); otherwise, throw it away (by removing it from M). If the algorithm examines all $\binom{n}{2}$ possible edges, then Y and M are both equal to the maximum spanning tree of the complete n-vertex graph, where the weight of an edge is the time when the algorithm asked about it.

Now, if an algorithm terminates before examining all $\binom{n}{2}$ edges, then there is at least one edge in M that is not in Y. Since the algorithm cannot distinguish between M and Y, even though M is connected and Y is not, the algorithm cannot possibly give the correct output for both graphs. Thus, in order to be correct, any algorithm must examine every edge—*Connectedness is evasive!*

29.6 An Evasive Conjecture

A graph property is *nontrivial* is there is at least one graph with the property and at least one graph without the property. (The only trivial properties are 'Yes' and 'No'.) A graph property is *monotone* if it is closed under taking subgraphs — if G has the property, then any subgraph of G has the property. For example, emptiness, planarity, acyclicity, and *non*-connectedness are monotone. The properties of being a tree and of having a vertex of degree 3 are not monotone.

Conjecture 1 (Aanderraa, Karp, and Rosenberg). *Every nontrivial monotone property of n-vertex graphs is evasive.*

The Aanderraa-Karp-Rosenberg conjecture has been proven when $n = p^e$ for some prime p and positive integer exponent e—the proof uses some interesting results from algebraic topology[2]—but it is still open for other values of n.

[2]Let Δ be a contractible simplicial complex whose automorphism group Aut(Δ) is vertex-transitive, and let Γ be a vertex-transitive subgroup of Aut(Δ). If there are normal subgroups $\Gamma_1 \lhd \Gamma_2 \lhd \Gamma$ such that $|\Gamma_1| = p^\alpha$ for some prime p and integer α, $|\Gamma/\Gamma_2| = q^\beta$ for some prime q and integer β, and Γ_2/Γ_1 is cyclic, then Δ is a simplex.
No, this will not be on the final exam.

There are non-trivial non-evasive graph properties, but all known examples are non-monotone. One such property—'scorpionhood'—is described in an exercise at the end of this lecture note.

29.7 Finding the Minimum and Maximum

Last time, we saw an adversary argument that finding the largest element of an unsorted set of n numbers requires at least $n-1$ comparisons. Let's consider the complexity of finding the largest *and* smallest elements. More formally:

> Given a sequence $X = \langle x_1, x_2, \dots, x_n \rangle$ of n distinct numbers, find indices i and j such that $x_i = \min X$ and $x_j = \max X$.

How many comparisons do we need to solve this problem? An upper bound of $2n-3$ is easy: find the minimum in $n-1$ comparisons, and then find the maximum of everything else in $n-2$ comparisons. Similarly, a lower bound of $n-1$ is easy, since any algorithm that finds the min and the max certainly finds the max.

We can improve both the upper and the lower bound to $\lceil 3n/2 \rceil - 2$. The upper bound is established by the following algorithm. Compare all $\lfloor n/2 \rfloor$ consecutive pairs of elements x_{2i-1} and x_{2i}, and put the smaller element into a set S and the larger element into a set L. if n is odd, put x_n into both L and S. Then find the smallest element of S and the largest element of L. The total number of comparisons is at most

$$\underbrace{\left\lfloor \frac{n}{2} \right\rfloor}_{\text{build } S \text{ and } L} + \underbrace{\left\lceil \frac{n}{2} \right\rceil - 1}_{\text{compute } \min S} + \underbrace{\left\lceil \frac{n}{2} \right\rceil - 1}_{\text{compute } \max L} = \left\lceil \frac{3n}{2} \right\rceil - 2.$$

For the lower bound, we use an adversary argument. The adversary marks each element $+$ if it *might* be the maximum element, and $-$ if it *might* be the minimum element. Initially, the adversary puts both marks $+$ and $-$ on every element. If the algorithm compares two double-marked elements, then the adversary declares one smaller, removes the $+$ mark from the smaller element, and removes the $-$ mark from the larger one. In every other case, the adversary can answer so that at most one mark needs to be removed. For example, if the algorithm compares a double marked element against one labeled $-$, the adversary says the one labeled $-$ is smaller and removes the $-$ mark from the other. If the algorithm compares to $+$'s, the adversary must unmark one of the two.

Initially, there are $2n$ marks. At the end, in order to be correct, exactly one item must be marked $+$ and exactly one other must be marked $-$, since the adversary can make any $+$ the maximum and any $-$ the minimum. Thus, the algorithm must force the adversary to remove $2n-2$ marks. At most $\lfloor n/2 \rfloor$ comparisons remove two marks; every other comparison removes at most one mark. Thus, the adversary strategy forces any algorithm to perform at least $2n-2-\lfloor n/2 \rfloor = \lceil 3n/2 \rceil - 2$ comparisons.

29.8 Finding the Median

Finally, let's consider the *median* problem: Given an unsorted array X of n numbers, find its $n/2$th largest entry. (I'll assume that n is even to eliminate pesky floors and ceilings.) More formally:

> Given a sequence $\langle x_1, x_2, \dots, x_n \rangle$ of n distinct numbers, find the index m such that x_m is the $n/2$th largest element in the sequence.

To prove a lower bound for this problem, we can use a combination of information theory and two adversary arguments. We use one adversary argument to prove the following simple lemma:

Lemma 1. *Any comparison tree that correctly finds the median element also identifies the elements smaller than the median and larger than the median.*

Proof: Suppose we reach a leaf of a decision tree that chooses the median element x_m, and there is still some element x_i that isn't known to be larger or smaller than x_m. In other words, we cannot decide based on the comparisons that we've already performed whether $x_i < x_m$ or $x_i > x_m$. Then in particular no element is known to lie between x_i and x_m. This means that there must be an input that is consistent with the comparisons we've performed, in which x_i and x_m are adjacent in sorted order. But then we can swap x_i and x_m, without changing the result of any comparison, and obtain a different consistent input in which x_i is the median, not x_m. Our decision tree gives the wrong answer for this 'swapped' input. □

This lemma lets us rephrase the median-finding problem yet again.

> Given a sequence $X = \langle x_1, x_2, \dots, x_n \rangle$ of n distinct numbers, find the indices of its $n/2-1$ largest elements L and its $n/2$th largest element x_m.

Now suppose a 'little birdie' tells us the set L of elements larger than the median. This information fixes the outcomes of certain comparisons—any item in L is bigger than any element not in L—so we can 'prune' those comparisons from the comparison tree. The pruned tree finds the largest element of $X \setminus L$ (the median of X), and thus must have depth at least $n/2-1$. In fact, the adversary argument in the last lecture implies that *every* leaf in the pruned tree must have depth at least $n/2-1$, so the pruned tree has at least $2^{n/2-1}$ leaves.

There are $\binom{n}{n/2-1} \approx 2^n/\sqrt{n/2}$ possible choices for the set L. Every leaf in the original comparison tree is also a leaf in exactly one of the $\binom{n}{n/2-1}$ pruned trees, so the original comparison tree must have at least $\binom{n}{n/2-1}2^{n/2-1} \approx 2^{3n/2}/\sqrt{n/2}$ leaves. Thus, any comparison tree that finds the median must have depth at least

$$\left\lceil \frac{n}{2} - 1 + \lg\binom{n}{n/2-1} \right\rceil = \frac{3n}{2} - O(\log n).$$

A more complicated adversary argument (also involving pruning the comparison tree with little birdies) improves this lower bound to $2n - o(n)$.

A similar argument implies that at least $n-k+\left\lceil \lg\binom{n}{k-1} \right\rceil = \Omega((n-k)+k\log(n/k))$ comparisons are required to find the kth largest element in an n-element set. This bound is tight up to constant factors for all $k \le n/2$; there is an algorithm that uses at most $O(n + k\log(n/k))$ comparisons. Moreover, this lower bound is *exactly* tight when $k = 1$ or $k = 2$. In fact, these are the *only* values of $k \le n/2$ for which the exact complexity of the selection problem is known. Even the case $k = 3$ is still open!

Exercises

1. (a) Let X be a set containing an odd number of n-bit strings. Prove that any algorithm that decides whether a given n-bit string is an element of X *must* examine every bit of the input string in the worst case.

   (b) Give a one-line proof that the bit pattern 01 is evasive for all even n.

(c) Prove that the bit pattern 11 is evasive if and only if $n \bmod 3 = 1$.

*(d) Prove that the bit pattern 111 is evasive if and only if $n \bmod 4 = 0$ or 3.

2. Suppose we are given the adjacency matrix of a *directed* graph G with n vertices. Describe an algorithm that determines whether G has a *sink* by probing only $O(n)$ bits in the input matrix. A sink is a vertex that has an incoming edge from every other vertex, but no outgoing edges.

*3. A *scorpion* is an undirected graph with three special vertices: the *sting*, the *tail*, and the *body*. The sting is connected only to the tail; the tail is connected only to the sting and the body; and the body is connected to every vertex except the sting. The rest of the vertices (the head, eyes, legs, antennae, teeth, gills, flippers, wheels, etc.) can be connected arbitrarily. Describe an algorithm that determines whether a given n-vertex graph is a scorpion by probing only $O(n)$ entries in the adjacency matrix.

4. Prove using an adversary argument that acyclicity is an evasive graph property. *[Hint: Kruskal.]*

5. Prove that finding the second largest element in an n-element array requires *exactly* $n - 2 + \lceil \lg n \rceil$ comparisons in the worst case. Prove the upper bound by describing and analyzing an algorithm; prove the lower bound using an adversary argument.

6. Let T be a perfect ternary tree where every leaf has depth ℓ. Suppose each of the 3^ℓ leaves of T is labeled with a bit, either 0 or 1, and each internal node is labeled with a bit that agrees with the *majority* of its children.

   (a) Prove that any deterministic algorithm that determines the label of the root must examine all 3^ℓ leaf bits in the worst case.

   (b) Describe and analyze a *randomized* algorithm that determines the root label, such that the expected number of leaves examined is $o(3^\ell)$. (You may want to review the notes on randomized algorithms.)

*7. UIUC has just finished constructing the new Reingold Building, the tallest dormitory on campus. In order to determine how much insurance to buy, the university administration needs to determine the highest safe floor in the building. A floor is consdered *safe* if ~~a drunk student~~ ***an egg*** can fall from a window on that floor and land without breaking; if the egg breaks, the floor is considered *unsafe*. Any floor that is higher than an unsafe floor is also considered unsafe. The only way to determine whether a floor is safe is to drop an egg from a window on that floor.

   You would like to find the lowest unsafe floor L by performing as few tests as possible; unfortunately, you have only a very limited supply of eggs.

   (a) Prove that if you have only one egg, you can find the lowest unsafe floor with L tests. *[Hint: Yes, this is trivial.]*

(b) Prove that if you have only one egg, you must perform at least L tests in the worst case. In other words, prove that your algorithm from part (a) is optimal. *[Hint: Use an adversary argument.]*

(c) Describe an algorithm to find the lowest unsafe floor using *two* eggs and only $O(\sqrt{L})$ tests. *[Hint: Ideally, each egg should be dropped the same number of times. How many floors can you test with n drops?]*

(d) Prove that if you start with two eggs, you must perform at least $\Omega(\sqrt{L})$ tests in the worst case. In other words, prove that your algorithm from part (c) is optimal.

*(e) Describe an algorithm to find the lowest unsafe floor using k eggs, using as few tests as possible, and prove your algorithm is optimal for all values of k.

*[I]n his short and broken treatise he provides an eternal example—not of laws, or even of method, for **there is no method except to be very intelligent**, but of intelligence itself swiftly operating the analysis of sensation to the point of principle and definition.*

— T. S. Eliot on Aristotle, "The Perfect Critic", *The Sacred Wood* (1921)

The nice thing about standards is that you have so many to choose from; furthermore, if you do not like any of them, you can just wait for next year's model.

— Andrew S. Tannenbaum, *Computer Networks* (1981)
Also attributed to Grace Murray Hopper and others

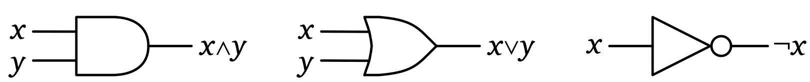
*If a problem has no solution, it may not be a problem, but a fact —
not to be solved, but to be coped with over time.*

— Shimon Peres, as quoted by David Rumsfeld, *Rumsfeld's Rules* (2001)

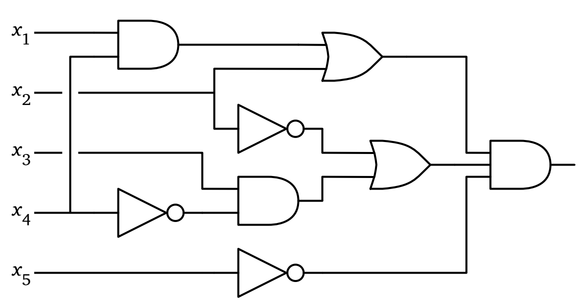
30 NP-Hard Problems

30.1 A Game You Can't Win

A salesman in a red suit who looks suspiciously like Tom Waits presents you with a black steel box with n binary switches on the front and a light bulb on the top. The salesman tells you that the state of the light bulb is controlled by a complex *boolean circuit*—a collection of AND, OR, and NOT gates connected by wires, with one input wire for each switch and a single output wire for the light bulb. He then asks you the following question: Is there a way to set the switches so that the light bulb turns on? If you can answer this question correctly, he will give you the box and a ~~million~~ ~~billion~~ trillion dollars; if you answer incorrectly, or if you die without answering at all, he will take your soul.



An AND gate, an OR gate, and a NOT gate.



A boolean circuit. inputs enter from the left, and the output leaves to the right.

As far as you can tell, the Adversary hasn't connected the switches to the light bulb at all, so no matter how you set the switches, the light bulb will stay off. If you declare that it *is* possible to turn on the light, the Adversary will open the box and reveal that there is no circuit at all. But if you declare that it is *not* possible to turn on the light, before testing all 2^n settings, the

Adversary will magically create a circuit inside the box that turns on the light *if and only if* the switches are in one of the settings you haven't tested, and then flip the switches to that setting, turning on the light. (You can't detect the Adversary's cheating, because you can't see inside the box until the end.) The only way to *provably* answer the Adversary's question correctly is to try all 2^n possible settings. You quickly realize that this will take *far* longer than you expect to live, so you gracefully decline the Adversary's offer.

The Adversary smiles and says, "Ah, yes, of course, you have no reason to trust me. But perhaps I can set your mind at ease." He hands you a large roll of parchment—which you hope was made from sheep skin—with a circuit diagram drawn (or perhaps tattooed) on it. "Here are the complete plans for the circuit inside the box. Feel free to poke around inside the box to make sure the plans are correct. Or build your own circuit from these plans. Or write a computer program to simulate the circuit. Whatever you like. If you discover that the plans don't match the actual circuit in the box, you win the trillion bucks." A few spot checks convince you that the plans have no obvious flaws; subtle cheating appears to be impossible.

But you should still decline the Adversary's generous offer. The problem that the Adversary is posing is called ***circuit satisfiability*** or ***CircuitSat***: Given a boolean circuit, is there is a set of inputs that makes the circuit output True, or conversely, whether the circuit *always* outputs False. For any particular input setting, we can calculate the output of the circuit in polynomial (actually, *linear*) time using depth-first-search. But nobody knows how to solve CircuitSat faster than just trying all 2^n possible inputs to the circuit, but this requires exponential time. On the other hand, nobody has actually *proved* that this is the best we can do; maybe there's a clever algorithm that just hasn't been discovered yet!

30.2 P versus NP

A minimal requirement for an algorithm to be considered "efficient" is that its running time is polynomial: $O(n^c)$ for some constant c, where n is the size of the input.[1] Researchers recognized early on that not all problems can be solved this quickly, but had a hard time figuring out exactly which ones could and which ones couldn't. There are several so-called ***NP-hard*** problems, which most people believe *cannot* be solved in polynomial time, even though nobody can prove a super-polynomial lower bound.

A *decision problem* is a problem whose output is a single boolean value: Yes or No. Let me define three classes of decision problems:

- ***P*** is the set of decision problems that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly.
- ***NP*** is the set of decision problems with the following property: If the answer is Yes, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of decision problems where we can verify a Yes answer quickly if we have the solution in front of us.
- ***co-NP*** is essentially the opposite of NP. If the answer to a problem in co-NP is No, then there is a proof of this fact that can be checked in polynomial time.

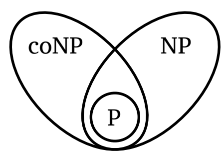
[1]This notion of efficiency was independently formalized by Alan Cobham (The intrinsic computational difficulty of functions. *Logic, Methodology, and Philosophy of Science (Proc. Int. Congress)*, 24–30, 1965), Jack Edmonds (Paths, trees, and flowers. *Canadian Journal of Mathematics* 17:449–467, 1965), and Michael Rabin (Mathematical theory of automata. *Proceedings of the 19th ACM Symposium in Applied Mathematics*, 153–175, 1966), although similar notions were considered more than a decade earlier by Kurt Gödel and John von Neumann.

For example, the circuit satisfiability problem is in NP. If the answer is YES, then any set of m input values that produces TRUE output is a proof of this fact; we can check the proof by evaluating the circuit in polynomial time. It is widely believed that circuit satisfiability is *not* in P or in co-NP, but nobody actually knows.

Every decision problem in P is also in NP. If a problem is in P, we can verify YES answers in polynomial time recomputing the answer from scratch! Similarly, every problem in P is also in co-NP.

Perhaps the single most important unanswered question in theoretical computer science—if not all of computer science—if not all of ***science***—is whether the complexity classes P and NP are actually different. Intuitively, it seems obvious to most people that P $\neq$ NP; the homeworks and exams in this class and others have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are obvious in retrospect. It's completely obvious; *of course* solving problems from scratch is harder than just checking that a solution is correct. But nobody knows how to prove it! The Clay Mathematics Institute lists P versus NP as the first of its seven Millennium Prize Problems, offering a \$1,000,000 reward for its solution. And yes, in fact, several people *have* lost their souls attempting to solve this problem.

A more subtle but still open question is whether the complexity classes NP and co-NP are different. Even if we can verify every YES answer quickly, there's no reason to believe we can also verify NO answers quickly. For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable. It is generally believed that NP $\neq$ co-NP, but nobody knows how to prove it.



What we *think* the world looks like.

30.3 NP-hard, NP-easy, and NP-complete

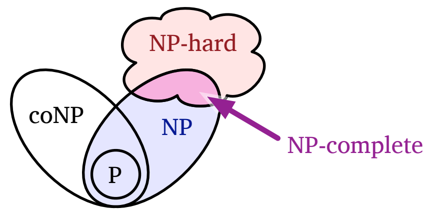
A problem Π is ***NP-hard*** if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for *every problem in NP*. In other words:

$$\boxed{\textbf{Π is NP-hard} \iff \textbf{If Π can be solved in polynomial time, then P=NP}}$$

Intuitively, if we could solve one particular NP-hard problem quickly, then we could quickly solve *any* problem whose solution is easy to understand, using the solution to that one special problem as a subroutine. NP-hard problems are at least as hard as any problem in NP.

Calling a problem NP-hard is like saying 'If I own a dog, then it can speak fluent English.' You probably don't know whether or not I own a dog, but I bet you'repretty sure that I don't own a *talking* dog. Nobody has a mathematical *proof* that dogs can't speak English—the fact that no one has ever heard a dog speak English is evidence, as are the hundreds of examinations of dogs that lacked the proper mouth shape and brainpower, but mere evidence is not a mathematical proof. Nevertheless, no sane person would believe me if I said I owned a dog that spoke fluent English. So the statement 'If I own a dog, then it can speak fluent English' has a natural corollary: No one in their right mind should believe that I own a dog! Likewise, if a problem is NP-hard, no one in their right mind should believe it can be solved in polynomial time.

Finally, a problem is ***NP-complete*** if it is both NP-hard and an element of NP (or 'NP-easy'). NP-complete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would imply a polynomial-time algorithm for *every* NP-complete problem. Literally *thousands* of problems have been shown to be NP-complete, so a polynomial-time algorithm for one (and therefore all) of them seems incredibly unlikely.



More of what we *think* the world looks like.

It is not immediately clear that *any* decision problems are NP-hard or NP-complete. NP-hardness is already a lot to demand of a problem; insisting that the problem also have a nondeterministic polynomial-time algorithm seems almost completely unreasonable. The following remarkable theorem was first published by Steve Cook in 1971 and independently by Leonid Levin in 1973.[2] I won't even sketch the proof, since I've been (deliberately) vague about the definitions.

The Cook-Levin Theorem. *Circuit satisfiability is NP-complete.*

*30.4 Formal Definitions (*HC SVNT DRACONES*)

Formally, the complexity classes P, NP, and co-NP are defined in terms of *languages* and *Turing machines*. A language is just a set of strings over some finite alphabet Σ; without loss of generality, we can assume that $\Sigma = \{0, 1\}$. P is the set of languages that can be decided in ***P***olynomial time by a deterministic single-tape Turing machine. Similarly, NP is the set of all languages that can be decided in polynomial time by a nondeterministic Turing machine; NP is an abbreviation for ***N***ondeterministic ***P***olynomial-time.

Polynomial time is a sufficient crude requirement that the precise form of Turing machine (number of heads, number of tracks, and so one) is unimportant. In fact, careful application and analysis of the techniques described in the Turing machine notes imply that any algorithm that runs on a random-access machine[3] in $T(n)$ time can be simulated by a single-tape, single-track, single-head Turing machine that runs in $O(T(n)^3)$ time. This simulation result allows us to argue formally about computational complexity in terms of standard high-level programming

[2]Levin first reported his results at seminars in Moscow in 1971, while still a PhD student. News of Cook's result did not reach the Soviet Union until at least 1973, after Levin's announcement of his results had been published; in accordance with Stigler's Law, this result is often called 'Cook's Theorem'. Levin was denied his PhD at Moscow University for political reasons; he emigrated to the US in 1978 and earned a PhD at MIT a year later. Cook was denied tenure at Berkeley in 1970, just one year before publishing his seminal paper; he (but not Levin) later won the Turing award for his proof.

[3]Random-access machines are a model of computation that more faithfully models physical computers. A random-access machine has unbounded random-access memory, modeled as an array $M[0..\infty]$ where each address $M[i]$ holds a single w-bit integer, for some fixed integer w, and can read to or write from any memory addresses in constant time. RAM algorithms are formally written in assembly-like language, using instructions like ***ADD*** i, j, k (meaning "$M[i] \leftarrow M[j] + M[k]$"), ***INDIR*** i, j (meaning "$M[i] \leftarrow M[M[j]]$"), and ***IFZGOTO*** i, ℓ (meaning "if $M[i] = 0$, go to line ℓ"). In practice, RAM algorithms can be faithfully described using higher-level pseudocode, as long as we're careful about arithmetic precision.

constructs like for-loops and recursion, instead of describing everything directly in terms of Turing machines.

A problem Π is formally NP-hard if and only if, for every language $\Pi' \in \text{NP}$, there is a polynomial-time ***Turing reduction*** from Π' to Π. A Turing reduction just means a reduction that can be executed on a Turing machine; that is, a Turing machine M that can solve Π' using another Turing machine M' for Π as a black-box subroutine. Turing reductions are also called *oracle reductions*; polynomial-time Turing reductions are also called *Cook reductions*.

Researchers in complexity theory prefer to define NP-hardness in terms of polynomial-time ***many-one reductions***, which are also called *Karp reductions*. A *many-one* reduction from one language $L' \subseteq \Sigma^*$ to another language $L \subseteq \Sigma^*$ is an function $f : \Sigma^* \to \Sigma^*$ such that $x \in L'$ if and only if $f(x) \in L$. Then we can define a *language* L to be NP-hard if and only if, for any language $L' \in \text{NP}$, there is a many-one reduction from L' to L that can be computed in polynomial time.

Every Karp reduction "is" a Cook reduction, but not vice versa. Specifically, any Karp reduction from one decision problem Π to another decision Π' is equivalent to transforming the input to Π into the input for Π', invoking an oracle (that is, a subroutine) for Π', and then returning the answer verbatim. However, as far as we know, not every Cook reduction can be simulated by a Karp reduction.

Complexity theorists prefer Karp reductions primarily because NP is closed under Karp reductions, but is *not* closed under Cook reductions (unless NP=co-NP, which is considered unlikely). There are natural problems that are (1) NP-hard with respect to Cook reductions, but (2) NP-hard with respect to Karp reductions only if P=NP. One trivial example is of such a problem is UnSat: Given a boolean formula, is it *always false*? On the other hand, many-one reductions apply *only* to decision problems (or more formally, to languages); formally, no optimization or construction problem is Karp-NP-hard.

To make things even more confusing, both Cook and Karp originally defined NP-hardness in terms of ***logarithmic-space*** reductions. Every logarithmic-space reduction is a polynomial-time reduction, but (as far as we know) not vice versa. It is an open question whether relaxing the set of allowed (Cook or Karp) reductions from logarithmic-space to polynomial-time changes the set of NP-hard problems.

Fortunately, none of these subtleties raise their ugly heads in practice—in particular, every algorithmic reduction described in these notes can be formalized as a logarithmic-space many-one reduction—so you can wake up now.

30.5 Reductions and SAT

To prove that any problem other than Circuit satisfiability is NP-hard, we use a *reduction argument*. Reducing problem A to another problem B means describing an algorithm to solve problem A under the assumption that an algorithm for problem B already exists. You're already used to doing reductions, only you probably call it something else, like writing subroutines or utility functions, or modular programming. To prove something is NP-hard, we describe a similar transformation between problems, but not in the direction that most people expect.

You should tattoo the following rule of onto the back of your hand, right next to your Mom's birthday and the *actual* rules of Monopoly.[4]

[4]If a player lands on an available property and declines (or is unable) to buy it, that property is immediately auctioned off to the highest bidder; the player who originally declined the property may bid, and bids may be arbitrarily higher or lower than the list price. Players in Jail can still buy and sell property, buy and sell houses and hotels, and collect rent. The game has 32 houses and 12 hotels; once they're gone, they're gone. In particular, if all houses are already on the board, you cannot downgrade a hotel to four houses; you must sell all three hotels in the

To prove that problem *A* is NP-hard, reduce a known NP-hard problem to *A*.

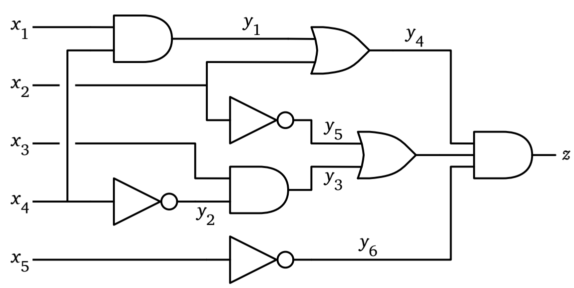
In other words, to prove that your problem is hard, you need to describe an algorithm to solve a *different* problem, which you already know is hard, using a mythical algorithm for *your* problem as a subroutine. The essential logic is a proof by contradiction. Your reduction shows implies that if your problem were easy, then the other problem would be easy, too. Equivalently, since you know the other problem is hard, your problem must also be hard.

For example, consider the *formula satisfiability* problem, usually just called ***SAT***. The input to SAT is a boolean *formula* like

$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

and the question is whether it is possible to assign boolean values to the variables $a, b, c, \ldots$ so that the formula evaluates to TRUE.

To show that SAT is NP-hard, we need to give a reduction from a known NP-hard problem. The only problem we know is NP-hard so far is circuit satisfiability, so let's start there. Given a boolean circuit, we can transform it into a boolean formula by creating new output variables for each gate, and then just writing down the list of gates separated by ANDS. For example, we can transform the example circuit into a formula as follows:



$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

A boolean circuit with gate variables added, and an equivalent boolean formula.

Now the original circuit is satisfiable if and only if the resulting formula is satisfiable. Given a satisfying input to the circuit, we can get a satisfying assignment for the formula by computing the output of every gate. Given a satisfying assignment for the formula, we can get a satisfying input the the circuit by just ignoring the internal gate variables y_i and the output variable z.

We can transform any boolean circuit into a formula in linear time using depth-first search, and the size of the resulting formula is only a constant factor larger than the size of the circuit. Thus, we have a polynomial-time reduction from circuit satisfiability to SAT:

★★★ Redraw reduction cartoons so that the *boxes* represent algorithms, not the arrows.

group. Players can sell/exchange undeveloped properties, but not buildings or cash. A player landing on Free Parking does not win anything. A player landing on Go gets \$200, no more. Railroads are not magic transporters. Finally, Jeff *always* gets the car.

$$\begin{tikzcd}
\boxed{\text{boolean circuit}} \arrow[r, "O(n)"] & \boxed{\text{boolean formula}} \arrow[d, Rightarrow, "\text{SAT}"] \\
\boxed{\text{TRUE or FALSE}} & \boxed{\text{TRUE or FALSE}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

$$T_{\text{CSAT}}(n) \le O(n) + T_{\text{SAT}}(O(n)) \quad \Longrightarrow \quad T_{\text{SAT}}(n) \ge T_{\text{CSAT}}(\Omega(n)) - O(n)$$

The reduction implies that if we had a polynomial-time algorithm for SAT, then we'd have a polynomial-time algorithm for circuit satisfiability, which would imply that P=NP. So SAT is NP-hard.

To prove that a boolean formula is satisfiable, we only have to specify an assignment to the variables that makes the formula TRUE. We can check the proof in linear time just by reading the formula from left to right, evaluating as we go. So SAT is also in NP, and thus is actually NP-complete.

30.6 3SAT (from SAT)

A special case of SAT that is particularly useful in proving NP-hardness results is called *3SAT*.

A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (AND) of several *clauses*, each of which is the disjunction (OR) of several *literals*, each of which is either a variable or its negation. For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

A *3CNF* formula is a CNF formula with exactly three literals per clause; the previous example is not a 3CNF formula, since its first clause has four literals and its last clause has only two. 3SAT is just SAT restricted to 3CNF formulas: Given a 3CNF formula, is there an assignment to the variables that makes the formula evaluate to TRUE?

We could prove that 3SAT is NP-hard by a reduction from the more general SAT problem, but it's easier just to start over from scratch, with a boolean circuit. We perform the reduction in several stages.

1. *Make sure every AND and OR gate has only two inputs.* If any gate has $k > 2$ inputs, replace it with a binary tree of $k-1$ two-input gates.

2. *Write down the circuit as a formula, with one clause per gate.* This is just the previous reduction.

3. *Change every gate clause into a CNF formula.* There are only three types of clauses, one for each type of gate:

$$\begin{aligned}
a = b \wedge c &\quad\longmapsto\quad (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\
a = b \vee c &\quad\longmapsto\quad (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\
a = \bar{b} &\quad\longmapsto\quad (a \vee b) \wedge (\bar{a} \vee \bar{b})
\end{aligned}$$

4. *Make sure every clause has exactly three literals.* Introduce new variables into each one- and two-literal clause, and expand it into two clauses as follows:

$$\begin{aligned}
a &\quad\longmapsto\quad (a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y}) \\
a \vee b &\quad\longmapsto\quad (a \vee b \vee x) \wedge (a \vee b \vee \bar{x})
\end{aligned}$$

For example, if we start with the same example circuit we used earlier, we obtain the following 3CNF formula. Although this may look a lot more ugly and complicated than the original circuit at first glance, it's actually only a constant factor larger—every binary gate in the original circuit has been transformed into at most five clauses. Even if the formula size were a large *polynomial* function (like n^{573}) of the circuit size, we would still have a valid reduction.

$$\begin{aligned}
&(y_1 \vee \overline{x_1} \vee \overline{x_4}) \wedge (\overline{y_1} \vee x_1 \vee z_1) \wedge (\overline{y_1} \vee x_1 \vee \overline{z_1}) \wedge (\overline{y_1} \vee x_4 \vee z_2) \wedge (\overline{y_1} \vee x_4 \vee \overline{z_2})\\
\wedge\, &(y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \overline{z_3}) \wedge (\overline{y_2} \vee \overline{x_4} \vee z_4) \wedge (\overline{y_2} \vee \overline{x_4} \vee \overline{z_4})\\
\wedge\, &(y_3 \vee \overline{x_3} \vee \overline{y_2}) \wedge (\overline{y_3} \vee x_3 \vee z_5) \wedge (\overline{y_3} \vee x_3 \vee \overline{z_5}) \wedge (\overline{y_3} \vee y_2 \vee z_6) \wedge (\overline{y_3} \vee y_2 \vee \overline{z_6})\\
\wedge\, &(\overline{y_4} \vee y_1 \vee x_2) \wedge (y_4 \vee \overline{x_2} \vee z_7) \wedge (y_4 \vee \overline{x_2} \vee \overline{z_7}) \wedge (y_4 \vee \overline{y_1} \vee z_8) \wedge (y_4 \vee \overline{y_1} \vee \overline{z_8})\\
\wedge\, &(y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \overline{z_9}) \wedge (\overline{y_5} \vee \overline{x_2} \vee z_{10}) \wedge (\overline{y_5} \vee \overline{x_2} \vee \overline{z_{10}})\\
\wedge\, &(y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \overline{z_{11}}) \wedge (\overline{y_6} \vee \overline{x_5} \vee z_{12}) \wedge (\overline{y_6} \vee \overline{x_5} \vee \overline{z_{12}})\\
\wedge\, &(\overline{y_7} \vee y_3 \vee y_5) \wedge (y_7 \vee \overline{y_3} \vee z_{13}) \wedge (y_7 \vee \overline{y_3} \vee \overline{z_{13}}) \wedge (y_7 \vee \overline{y_5} \vee z_{14}) \wedge (y_7 \vee \overline{y_5} \vee \overline{z_{14}})\\
\wedge\, &(y_8 \vee \overline{y_4} \vee \overline{y_7}) \wedge (\overline{y_8} \vee y_4 \vee z_{15}) \wedge (\overline{y_8} \vee y_4 \vee \overline{z_{15}}) \wedge (\overline{y_8} \vee y_7 \vee z_{16}) \wedge (\overline{y_8} \vee y_7 \vee \overline{z_{16}})\\
\wedge\, &(y_9 \vee \overline{y_8} \vee \overline{y_6}) \wedge (\overline{y_9} \vee y_8 \vee z_{17}) \wedge (\overline{y_9} \vee y_8 \vee \overline{z_{17}}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}})\\
\wedge\, &(y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \overline{z_{19}} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \overline{z_{20}}) \wedge (y_9 \vee \overline{z_{19}} \vee \overline{z_{20}})
\end{aligned}$$

This process transforms the circuit into an equivalent 3CNF formula; the output formula is satisfiable if and only if the input circuit is satisfiable. As with the more general SAT problem, the formula is only a constant factor larger than any reasonable description of the original circuit, and the reduction can be carried out in polynomial time. Thus, we have a polynomial-time reduction from circuit satisfiability to 3SAT:

$$\begin{tikzcd}
\boxed{\text{boolean circuit}} \arrow[r, "O(n)"] & \boxed{\text{3CNF formula}} \arrow[d, Rightarrow, "\text{3SAT}"] \\
\boxed{\textsc{True or False}} & \boxed{\textsc{True or False}} \arrow[l, "\text{trivial}"']
\end{tikzcd}$$

$$T_{\text{CSAT}}(n) \le O(n) + T_{\text{3SAT}}(O(n)) \quad\Longrightarrow\quad T_{\text{3SAT}}(n) \ge T_{\text{CSAT}}(\Omega(n)) - O(n)$$

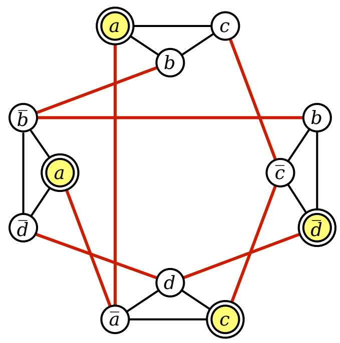
We conclude 3SAT is NP-hard. And because 3SAT is a special case of SAT, it is also in NP. Therefore, 3SAT is NP-complete.

30.7 Maximum Independent Set (from 3SAT)

For the next few problems we consider, the input is a simple, unweighted graph, and the problem asks for the size of the largest or smallest subgraph satisfying some structural property.

Let G be an arbitrary graph. An ***independent set*** in G is a subset of the vertices of G with no edges between them. The *maximum independent set* problem, or simply ***MaxIndSet***, asks for the size of the largest independent set in a given graph.

I'll prove that MaxIndSet is NP-hard (but not NP-complete, since it isn't a decision problem) using a reduction from 3SAT. I'll describe a reduction from a 3CNF formula into a graph that has an independent set of a certain size if and only if the formula is satisfiable. The graph has one node for each instance of each literal in the formula. Two nodes are connected by an edge if (1) they correspond to literals in the same clause, or (2) they correspond to a variable and its inverse. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ is transformed into the following graph.



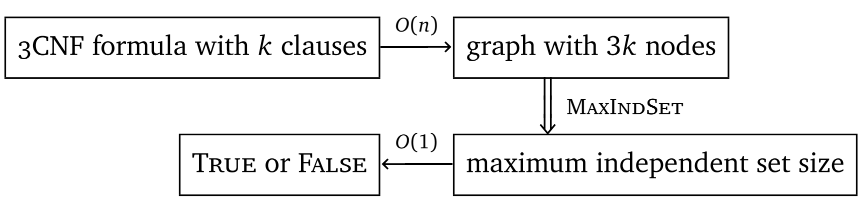
A graph derived from a 3CNF formula, and an independent set of size 4.
Black edges join literals from the same clause; red (heavier) edges join contradictory literals.

Now suppose the original formula had k clauses. Then I claim that the formula is satisfiable if and only if the graph has an independent set of size k.

1. **independent set $\Longrightarrow$ satisfying assignment:** If the graph has an independent set of k vertices, then each vertex must come from a different clause. To obtain a satisfying assignment, we assign the value TRUE to each literal in the independent set. Since contradictory literals are connected by edges, this assignment is consistent. There may be variables that have no literal in the independent set; we can set these to any value we like. The resulting assignment satisfies the original 3CNF formula.

2. **satisfying assignment $\Longrightarrow$ independent set:** If we have a satisfying assignment, then we can choose one literal in each clause that is TRUE. Those literals form an independent set in the graph.

Thus, the reduction is correct. Since the reduction from 3CNF formula to graph takes polynomial time, we conclude that MAXINDSET is NP-hard. Here's a diagram of the reduction:

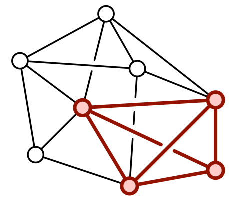


$$T_{3SAT}(n) \leq O(n) + T_{\text{MAXINDSET}}(O(n)) \quad \Longrightarrow \quad T_{\text{MAXINDSET}}(n) \geq T_{3SAT}(\Omega(n)) - O(n)$$

30.8 Clique (from Independent Set)

A *clique* is another name for a complete graph, that is, a graph where every pair of vertices is connected by an edge. The *maximum clique size* problem, or simply MAXCLIQUE, is to compute, given a graph, the number of nodes in its largest complete subgraph.

There is an easy proof that MAXCLIQUE is NP-hard, using a reduction from MAXINDSET. Any graph G has an *edge-complement* $\overline{G}$ with the same vertices, but with exactly the opposite set of edges—(u, v) is an edge in $\overline{G}$ if and only if it is *not* an edge in G. A set of vertices is independent in G if and only if the same vertices define a clique in $\overline{G}$. Thus, we can compute the largest independent in a graph simply by computing the largest clique in the complement of the graph.



A graph with maximum clique size 4.

$$
\begin{tikzcd}
\boxed{\text{graph } G} \arrow[r, "O(n)"] & \boxed{\text{complement graph } \overline{G}} \arrow[d, Rightarrow, "\textsc{MaxClique}"] \\
\boxed{\text{largest independent set}} & \boxed{\text{largest clique}} \arrow[l, "\text{trivial}"']
\end{tikzcd}
$$

30.9 Vertex Cover (from Independent Set)

A *vertex cover* of a graph is a set of vertices that touches every edge in the graph. The MINVERTEXCOVER problem is to find the smallest vertex cover in a given graph.

Again, the proof of NP-hardness is simple, and relies on just one fact: If I is an independent set in a graph $G = (V, E)$, then $V \setminus I$ is a vertex cover. Thus, to find the *largest* independent set, we just need to find the vertices that aren't in the *smallest* vertex cover of the same graph.

$$
\begin{tikzcd}
\boxed{\text{graph } G = (V, E)} \arrow[r, "\text{trivial}"] & \boxed{\text{graph } G = (V, E)} \arrow[d, Rightarrow, "\textsc{MinVertexCover}"] \\
\boxed{\text{largest independent set } V \setminus S} & \boxed{\text{smallest vertex cover } S} \arrow[l, "O(n)"']
\end{tikzcd}
$$

30.10 Graph Coloring (from 3SAT)

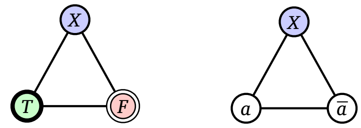
A *k-coloring* of a graph is a map $C : V \to \{1, 2, \ldots, k\}$ that assigns one of k 'colors' to each vertex, so that every edge has two different colors at its endpoints. The graph coloring problem is to find the smallest possible number of colors in a legal coloring. To show that this problem is NP-hard, it's enough to consider the special case 3COLORABLE: Given a graph, does it have a 3-coloring?

To prove that 3COLORABLE is NP-hard, we use a reduction from 3SAT. Given a 3CNF formula Φ, we produce a graph G_Φ as follows. The graph consists of a *truth* gadget, one *variable* gadget for each variable in the formula, and one *clause* gadget for each clause in the formula.

- The truth gadget is just a triangle with three vertices T, F, and X, which intuitively stand for TRUE, FALSE, and OTHER. Since these vertices are all connected, they must have different colors in any 3-coloring. For the sake of convenience, we will *name* those colors TRUE, FALSE, and OTHER. Thus, when we say that a node is colored TRUE, all we mean is that it must be colored the same as the node T.

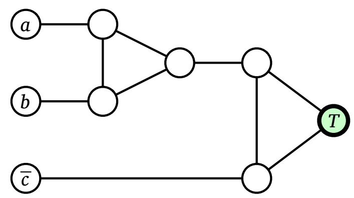
- The variable gadget for a variable a is also a triangle joining two new nodes labeled a and $\overline{a}$ to node X in the truth gadget. Node a must be colored either TRUE or FALSE, and so node $\overline{a}$ must be colored either FALSE or TRUE, respectively.

- Finally, each clause gadget joins three literal nodes to node T in the truth gadget using five new unlabeled nodes and ten edges; see the figure below. A straightforward case analysis



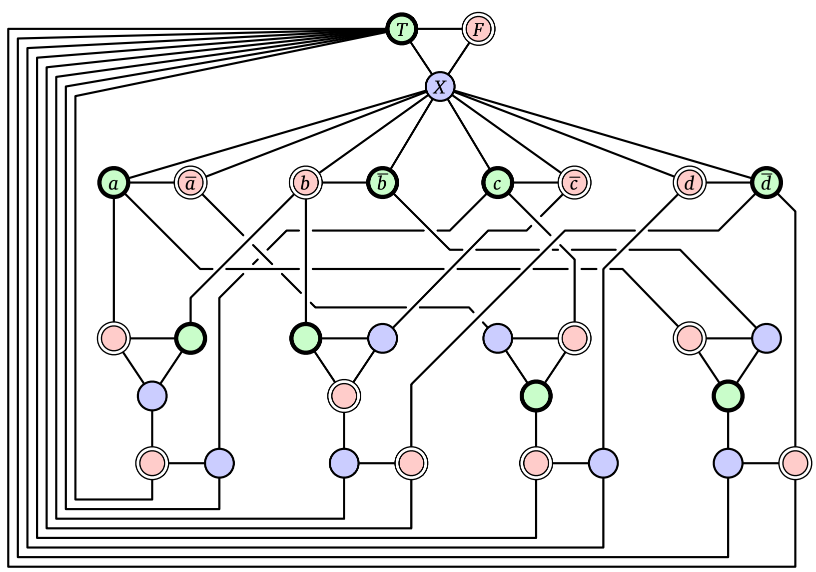
The truth gadget and a variable gadget for a.

implies that if all three literal nodes in the clause gadget are colored FALSE, then some edge in the gadget must be monochromatic. Since the variable gadgets force each literal node to be colored either TRUE or FALSE, in any valid 3-coloring, at least one of the three literal nodes is colored TRUE. On the other hand, for any coloring of the literal nodes where at least one literal node is colored TRUE, there is a valid 3-coloring of the clause gadget.



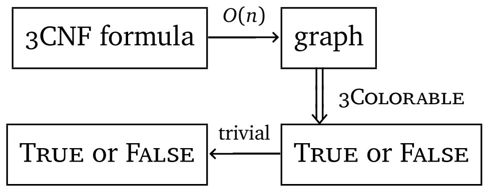
A clause gadget for $(a \vee b \vee \bar{c})$.

The final graph G_Φ contains exactly *one* node T, exactly *one* node F, and exactly *two* nodes a and $\bar{a}$ for each variable. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ that I used to illustrate the MAXCLIQUE reduction would be transformed into the graph shown on the next page. The 3-coloring is one of several that correspond to the satisfying assignment $a = c =$ TRUE, $b = d =$ FALSE.



A 3-colorable graph derived from the satisfiable 3CNF formula
$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$

Now the proof of correctness is just brute force case analysis. If the graph is 3-colorable, then we can extract a satisfying assignment from any 3-coloring—at least one of the three literal nodes in every clause gadget is colored TRUE. Conversely, if the formula is satisfiable, then we can color the graph according to any satisfying assignment.



We can easily verify that a graph has been correctly 3-colored in linear time: just compare the endpoints of every edge. Thus, 3Coloring is in NP, and therefore NP-complete. Moreover, since 3Coloring is a special case of the more general graph coloring problem—What is the minimum number of colors?—the more problem is also NP-hard, but *not* NP-complete, because it's not a decision problem.

30.11 Hamiltonian Cycle (from Vertex Cover)

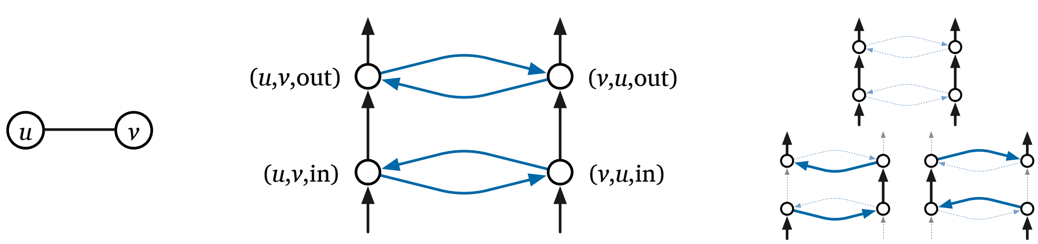
A ***Hamiltonian cycle*** in a graph is a cycle that visits every vertex exactly once. This is very different from an *Eulerian cycle*, which is actually a closed *walk* that traverses every *edge* exactly once. Eulerian cycles are easy to find and construct in linear time using a variant of depth-first search.

To prove that finding a Hamiltonian cycle in a directed graph is NP-hard, we describe a reduction from the vertex cover problem. Given an undirected graph G and an integer k, we need to transform it into another graph H, such that H has a Hamiltonian cycle if and only if G has a vertex cover of size k. As usual, our transformation uses several gadgets.

- For each undirected edge uv in G, the directed graph H contains an *edge gadget* consisting of four vertices $(u, v, \text{in}), (u, v, \text{out}), (v, u, \text{in}), (v, u, \text{out})$ and six directed edges

$$\begin{array}{ccc} (u,v,\text{in})\to(u,v,\text{out}) & (u,v,\text{in})\to(v,u,\text{in}) & (v,u,\text{in})\to(u,v,\text{in}) \\ (v,u,\text{in})\to(v,u,\text{out}) & (u,v,\text{out})\to(v,u,\text{out}) & (v,u,\text{out})\to(u,v,\text{out}) \end{array}$$

  as shown on the next page. Each "in" vertex has an additional incoming edge, and each "out" vertex has an additional outgoing edge. A Hamiltonian cycle must pass through an edge gadget in one of three ways—either straight through on both sides, or with a detour from one side to the other and back. Eventually, these options will correspond to both u and v, only u, or only v belonging to some vertex cover.

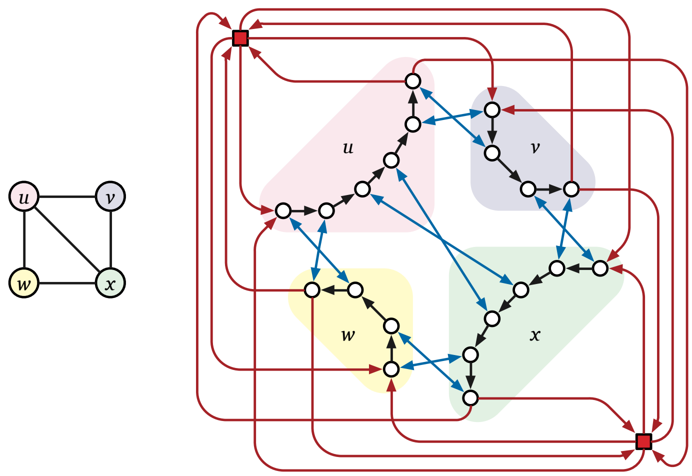


An edge gadget for uv and its only possible intersections with a Hamiltonian cycle.

- For each vertex u in G, all the edge gadgets for incident edges uv are connected in H into a single directed path, which we call a *vertex chain*. Specifically, suppose vertex u has d neighbors $v_1, v_2, \ldots, v_d$. Then H has $d-1$ additional edges $(u, v_i, \text{out})\to(u, v_{i+1}, \text{in})$ for each i.

- Finally, H also contains k *cover vertices*, simply numbered 1 through k. Each cover vertex has a directed edge to the first vertex in each vertex chain, and a directed edge from the last vertex in each vertex chain.

An example of our complete transformation is shown below.

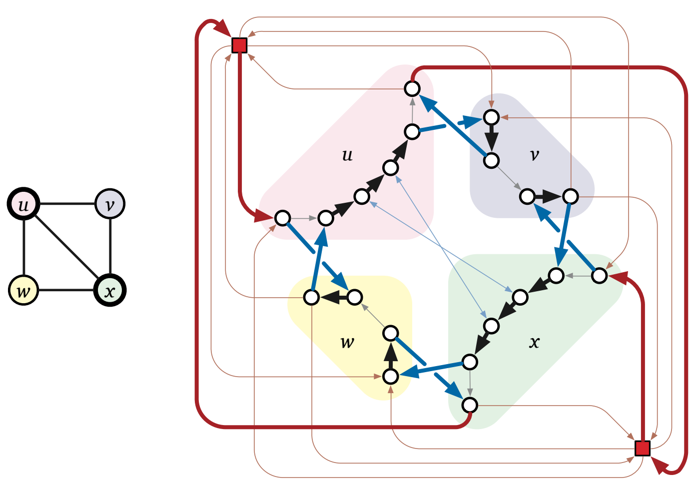


The original graph G and the transformed graph H, where $k = 2$.

Now suppose $C = \{u_1, u_2, \ldots, u_k\}$ is a vertex cover of G. Then H contains a Hamiltonian cycle, constructed as follows. Start at cover vertex 1, through traverse the vertex chain for vu_1, then visit cover vertex 2, then traverse the vertex chain for vu_2, and so forth, eventually returning to cover vertex 1. As we traverse the vertex chain for any vertex u_i, we have a choice for how to proceed when we reach any node (u_i, v, in).

- If $v \in C$, follow the edge $(u_i, v, \text{in}) \to (u_i, v, \text{out})$.
- If $v \notin C$, detour through the path $(u_i, v, \text{in}) \to (v, u_i, \text{in}) \to (v, u_i, \text{out}) \to (u_i, v, \text{out})$.

Thus, for each edge uv of G, the Hamiltonian cycle visits (u, v, in) and (u, v, out) as part of u's vertex chain if $u \in C$ and as part of v's vertex chain otherwise.



A vertex cover $\{u, x\}$ in G and the corresponding Hamiltonian cycle in H.

Now suppose H contains a Hamiltonian cycle C. This cycle must contain an edge from each cover vertex to the start of some vertex chain. Our case analysis of edge gadgets inductively implies that after C enters the vertex chain for some vertex u, it must traverse the entire vertex chain. Specifically, at each vertex (u, v, in), the cycle must contain either the single edge $(u, v, \text{in}) \rightarrow (u, v, \text{out})$ or the detour path $(u, v, \text{in}) \rightarrow (v, u, \text{in}) \rightarrow (v, u, \text{out}) \rightarrow (u, v, \text{out})$, followed by an edge to the next edge gadget in u's vertex chain, or to a cover vertex if this is the last such edge gadget. In particular, if C contains the detour edge $(u, v, \text{in}) \rightarrow (v, u, \text{in})$, it does not contain edges between any cover vertex and v's vertex chain. It follows that C traverses exactly k vertex chains. Moreover, these vertex chains describe a vertex cover of the original graph G, because C visits the vertex (u, v, in) for every edge uv in G.

We conclude that G contains a vertex cover of size k if and only if H contains a Hamiltonian cycle.

The transformation from G to H takes at most $O(n^2)$ time; we conclude that the Hamiltonian cycle problem is NP-hard. Moreover, since we can easily verify a Hamiltonian cycle in linear time, the Hamiltonian cycle problem is in NP, and therefore is NP-complete.

$$\begin{tikzcd} \boxed{\text{undirected graph } G, \text{ integer } k} \arrow[r, "O(n^2)"] & \boxed{\text{directed graph } H} \arrow[d, Rightarrow, "\textsc{HamCycle}"] \\ \boxed{\textsc{True or False}} & \boxed{\textsc{True or False}} \arrow[l, "\text{trivial}"'] \end{tikzcd}$$

A closely related problem to Hamiltonian cycles is the famous *traveling salesman problem*—Given a *weighted* graph G, find the shortest cycle that visits every vertex. Finding the shortest cycle is obviously harder than determining if a cycle exists at all, so the traveling salesman problem is also NP-hard.

Finally, we can prove prove that finding Hamiltonian cycles in *undirected* graphs is NP-hard using a simple reduction from the same problem in *directed* graphs. I'll leave the details of this reduction as an entertaining exercise.

30.12 Subset Sum (from Vertex Cover)

The next problem that we prove NP-hard is the SubsetSum problem considered in the very first lecture on recursion: Given a set X of positive integers and an integer t, determine whether X has a subset whose elements sum to t.

To prove this problem is NP-hard, we once again reduce from VertexCover. Given a graph G and an integer k, we compute a set X of integer and an integer t, such that X has a subset that sums to t if and only if G has an vertex cover of size k. Our transformation uses just two 'gadgets', which are *integers* representing vertices and edges in G.

Number the *edges* of G arbitrarily from 0 to $m-1$. Our set X contains the integer $b_i := 4^i$ for each edge i, and the integer

$$a_v := 4^m + \sum_{i \in \Delta(v)} 4^i$$

for each vertex v, where $\Delta(v)$ is the set of edges that have v has an endpoint. Alternately, we can think of each integer in X as an $(m+1)$-digit number written in base 4. The mth digit is 1 if the integer represents a vertex, and 0 otherwise; and for each $i < m$, the ith digit is 1 if the integer represents edge i or one of its endpoints, and 0 otherwise. Finally, we set the target sum

$$t := k \cdot 4^m + \sum_{i=0}^{m-1} 2 \cdot 4^i.$$

Now let's prove that the reduction is correct. First, suppose there is a vertex cover of size k in the original graph G. Consider the subset $X_C \subseteq X$ that includes a_v for every vertex v in the vertex cover, and b_i for every edge i that has *exactly one* vertex in the cover. The sum of these integers, written in base 4, has a 2 in each of the first m digits; in the most significant digit, we are summing exactly k 1's. Thus, the sum of the elements of X_C is exactly t.

On the other hand, suppose there is a subset $X' \subseteq X$ that sums to t. Specifically, we must have

$$\sum_{v \in V'} a_v + \sum_{i \in E'} b_i = t$$

for some subsets $V' \subseteq V$ and $E' \subseteq E$. Again, if we sum these base-4 numbers, there are no carries in the first m digits, because for each i there are only three numbers in X whose ith digit is 1. Each edge number b_i contributes only one 1 to the ith digit of the sum, but the ith digit of t is 2. Thus, for each edge in G, at least one of its endpoints must be in V'. In other words, V is a vertex cover. On the other hand, only vertex numbers are larger than 4^m, and $\lfloor t/4^m \rfloor = k$, so V' has at most k elements. (In fact, it's not hard to see that V' has *exactly* k elements.)

For example, given the four-vertex graph used on the previous page to illustrate the reduction to Hamiltonian cycle, our set X might contain the following base-4 integers:

$$\begin{array}{rlrl}
a_u := 111000_4 = 1344 & \quad & b_{uv} := 010000_4 = 256 \\
a_v := 110110_4 = 1300 & & b_{uw} := 001000_4 = 64 \\
a_w := 101101_4 = 1105 & & b_{vw} := 000100_4 = 16 \\
a_x := 100011_4 = 1029 & & b_{vx} := 000010_4 = 4 \\
 & & b_{wx} := 000001_4 = 1
\end{array}$$

If we are looking for a vertex cover of size 2, our target sum would be $t := 222222_4 = 2730$. Indeed, the vertex cover $\{v, w\}$ corresponds to the subset $\{a_v, a_w, b_{uv}, b_{uw}, b_{vx}, b_{wx}\}$, whose sum is $1300 + 1105 + 256 + 64 + 4 + 1 = 2730$.

The reduction can clearly be performed in polynomial time. Since VertexCover is NP-hard, it follows that SubsetSum is NP-hard.

There is one subtle point that needs to be emphasized here. Way back at the beginning of the semester, we developed a dynamic programming algorithm to solve SubsetSum in time $O(nt)$. Isn't this a polynomial-time algorithm? idn't we just prove that P=NP? Hey, where's our million dollars? Alas, life is not so simple. True, the running time is polynomial in n and t, but in order to qualify as a true polynomial-time algorithm, the running time must be a polynomial function *of the size of the input*. The *values* of the elements of X and the target sum t could be exponentially larger than the number of input bits. Indeed, the reduction we just described produces a value of t that is exponentially larger than the size of our original input graph, which would force our dynamic programming algorithm to run in exponential time.

Algorithms like this are said to run in ***pseudo-polynomial time***, and any NP-hard problem with such an algorithm is called ***weakly NP-hard***. Equivalently, a weakly NP-hard problem is one that can be solved in polynomial time when all input numbers are represented in *unary* (as a sum of 1s), but becomes NP-hard when all input numbers are represented in *binary*. If a problem is NP-hard even when all the input numbers are represented in unary, we say that the problem is ***strongly NP-hard***.

30.13 A Frivolous Example

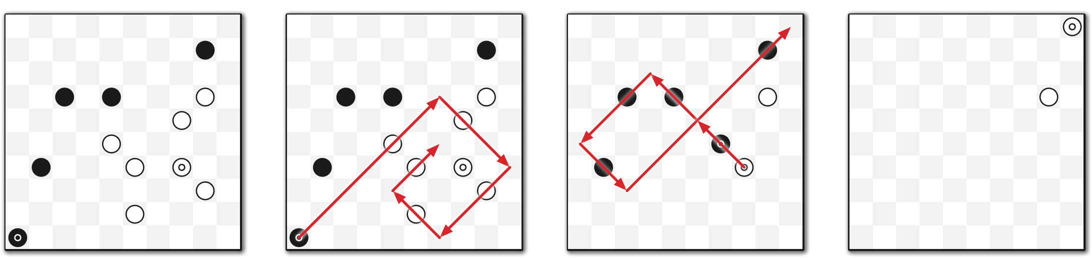
Draughts is a family of board games that have been played for thousands of years. Most Americans are familiar with the version called *checkers* or *English draughts*, but the most common variant worldwide, known as ***international draughts*** or ***Polish draughts***, originated in the Netherlands in the 16th century. For a complete set of rules, the reader should consult Wikipedia; here a few important differences from the Anglo-American game:

- **Flying kings:** As in checkers, a piece that ends a move in the row closest to the opponent becomes a *king* and gains the ability to move backward. Unlike in checkers, however, a king in international draughts can move any distance along a diagonal line in a single turn, as long as the intermediate squares are empty or contain exactly one opposing piece (which is captured).

- **Forced maximum capture:** In each turn, the moving player must capture as many opposing pieces as possible. This is distinct from the forced-capture rule in checkers, which requires only that each player must capture if possible, and that a capturing move ends only when the moving piece cannot capture further. In other words, checkers requires capturing a *maximal* set of opposing pieces on each turn; whereas, international draughts requires a *maximum* capture.

- **Capture subtleties:** As in checkers, captured pieces are removed from the board only at the end of the turn. Any piece can be captured at most once. Thus, when an opposing piece is jumped, that piece remains on the board *but cannot be jumped again* until the end of the turn.

For example, in the first position shown below, each circle represents a piece, and doubled circles represent kings. Black must make the indicated move, capturing five white pieces, because it is not possible to capture more than five pieces, and there is no other move that captures five. Black cannot extend his capture further northeast, because the captured White pieces are still on the board.



Two forced(!) moves in international draughts.

The actual game, which is played on a 10×10 board with 20 pieces of each color, is computationally trivial; we can precompute the optimal move for both players in every possible board configuration and hard-code the results into a lookup table of constant size. Sure, it's a *big* constant, but it's still just a constant!

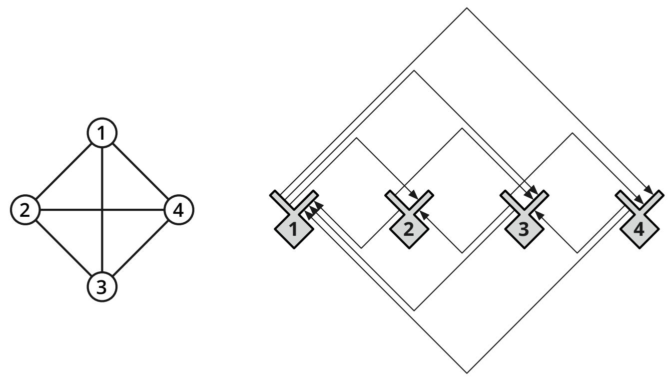
But consider the natural generalization of international draughts to an $n \times n$ board. In this setting, ***finding a legal move is actually NP-hard!*** The following reduction from the Hamiltonian cycle problem in directed graphs was discovered by Bob Hearn in 2010.[5] In most two-player

[5]Posted on Theoretical Computer Science Stack Exchange: http://cstheory.stackexchange.com/a/1999/111.

games, finding the *best* move is NP-hard (or worse); this is the only example I know of a game where just *following the rules* is an intractable problem!

Given a graph G with n vertices, we construct a board configuration for international draughts, such that White can capture a certain number of black pieces in a single move if and only if G has a Hamiltonian cycle. We treat G as a directed graph, with two arcs $u \rightarrow v$ and $v \rightarrow u$ in place of each undirected edge uv. Number the vertices arbitrarily from 1 to n. The final draughts configuration has several gadgets.

- The vertices of G are represented by rabbit-shaped *vertex gadgets*, which are evenly spaced along a horizontal line. Each arc $i \rightarrow j$ is represented by a path of two diagonal line segments from the "right ear" of vertex gadget i to the "left ear" of vertex gadget j. The path for arc $i \rightarrow j$ is located above the vertex gadgets if $i < j$, and below the vertex gadgets if $i > j$.



A high level view of the reduction from Hamiltonian cycle to international draughts.

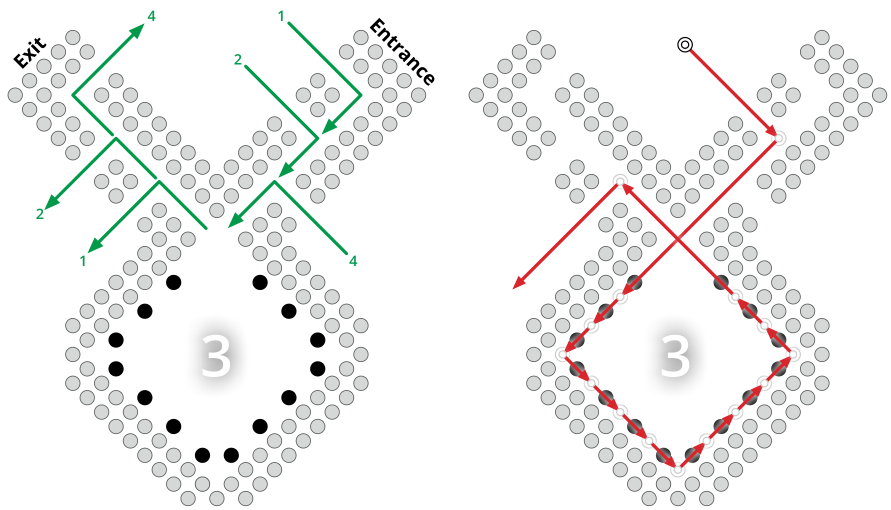
- The bulk of each vertex gadget is a diamond-shaped region called a *vault*. The walls of the vault are composed of two solid layers of black pieces, which cannot be captured; these pieces are drawn as gray circles in the figures. There are N capturable black pieces inside each vault, for some large integer N to be determined later. A white king can enter the vault through the "right ear", capture every internal piece, and then exit through the "left ear". Both ears are hallways, again with walls two pieces thick, with gaps where the arc paths end to allow the white king to enter and leave. The lengths of the "ears" can be adjusted easily to align with the other gadgets.

- For each arc $i \rightarrow j$, we have a *corner gadget*, which allows a white king leaving vertex gadget i to be redirected to vertex gadget j.

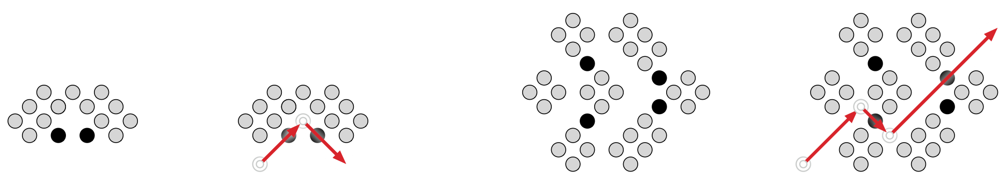
- Finally, wherever two arc paths cross, we have a *crossing gadget*; these gadgets allow the white king to traverse either arc path, but forbid switching from one arc path to the other.

A single white king starts at the bottom corner of one of the vaults. In any legal move, this king must alternate between traversing entire arc paths and clearing vaults. The king can traverse the various gadgets backward, entering each vault through the exit and vice versa. But the reversal of a Hamiltonian cycle in G is another Hamiltonian cycle in G, so walking backward is fine.

If there is a Hamiltonian cycle in G, the white king can capture at least nN black pieces by visiting each of the other vaults and returning to the starting vault. On the other hand, if there is no Hamiltonian cycle in G, the white king can can capture at most half of the pieces in the



Left: A vertex gadget. Right: A white king emptying the vault.
Gray circles are black pieces that cannot be captured.


A corner gadget and a crossing gadget.

starting vault, and thus can capture at most $(n-1/2)N + O(n^3)$ enemy pieces altogether. The $O(n^3)$ term accounts for the corner and crossing gadgets; each edge passes through one corner gadget and at most $n^2/2$ crossing gadgets.

To complete the reduction, we set $N = n^4$. Summing up, we obtain an $O(n^5) \times O(n^5)$ board configuration, with $O(n^5)$ black pieces and one white king. We can clearly construct this board configuration in polynomial time. A complete example of the construction appears on the next page.

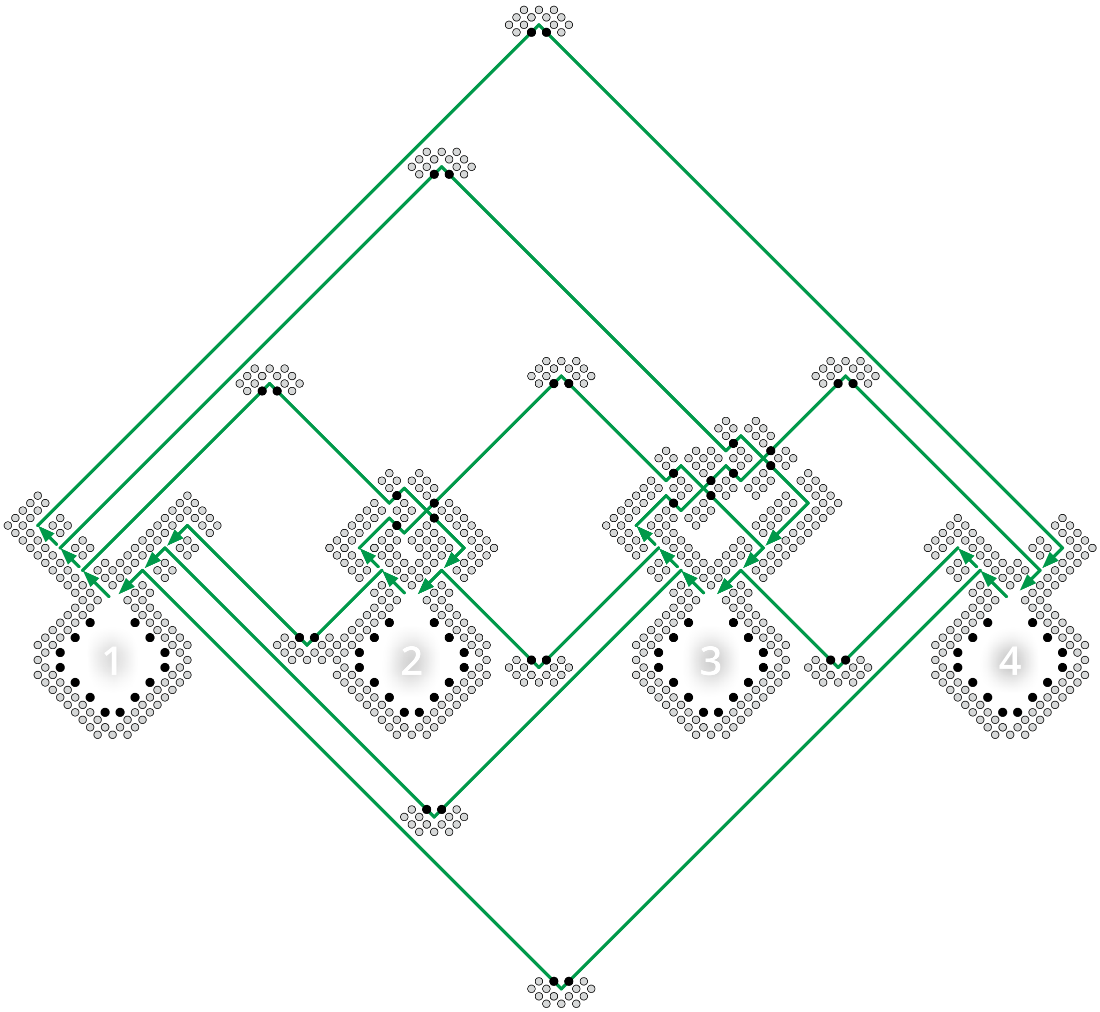
It is still open whether the following related question is NP-hard: Given an $n \times n$ board configuration for international draughts, can (and therefore *must*) White capture *all* the black pieces in a single turn?

30.14 Other Useful NP-hard Problems

Literally thousands of different problems have been proved to be NP-hard. I want to close this note by listing a few NP-hard problems that are useful in deriving reductions. I won't describe the NP-hardness proofs for these problems in detail, but you can find most of them in Garey and Johnson's classic *Scary Black Book of NP-Completeness*.[6]

- PLANARCIRCUITSAT: Given a boolean circuit that can be embedded in the plane so that no two wires cross, is there an input that makes the circuit output TRUE? This problem can be proved NP-hard by reduction from the general circuit satisfiability problem, by replacing each crossing with a small series of gates.

[6]Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.



The final draughts configuration for the example graph. (The green arrows are not

- NotAllEqual3SAT: Given a 3CNF formula, is there an assignment of values to the variables so that every clause contains at least one True literal *and* at least one False literal? This problem can be proved NP-hard by reduction from the usual 3SAT.

- Planar3SAT: Given a 3CNF boolean formula, consider a bipartite graph whose vertices are the clauses and variables, where an edge indicates that a variable (or its negation) appears in a clause. If this graph is planar, the 3CNF formula is also called planar. The Planar3SAT problem asks, given a planar 3CNF formula, whether it has a satisfying assignment. This problem can be proved NP-hard by reduction from PlanarCircuitSAT.[7]

- Exact3DimensionalMatching or X3M: Given a set S and a collection of three-element subsets of S, called *triples*, is there a sub-collection of disjoint triples that exactly cover S? This problem can be proved NP-hard by a reduction from 3SAT.

- Partition: Given a set S of n integers, are there subsets A and B such that $A \cup B = S$, $A \cap B = \varnothing$, and
$$\sum_{a \in A} a = \sum_{b \in B} b?$$

[7]Surprisingly, PlanarNotAllEqual3SAT is solvable in polynomial time!

This problem can be proved NP-hard by a simple reduction from SubsetSum. Like SubsetSum, the Partition problem is only weakly NP-hard.

- 3Partition: Given a set S of $3n$ integers, can it be partitioned into n disjoint three-element subsets, such that every subset has exactly the same sum? Despite the similar names, this problem is *very* different from Partition; sorry, I didn't make up the names. This problem can be proved NP-hard by reduction from X3M. Unlike Partition, the 3Partition problem is *strongly* NP-hard, that is, it remains NP-hard even if the input numbers are less than some polynomial in n.
- SetCover: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the smallest sub-collection of S_i's that contains all the elements of $\bigcup_i S_i$. This problem is a generalization of both VertexCover and X3M.
- HittingSet: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the minimum number of elements of $\bigcup_i S_i$ that hit every set in $\mathcal{S}$. This problem is also a generalization of VertexCover.
- HamiltonianPath: Given an graph G, is there a path in G that visits every vertex exactly once? This problem can be proved NP-hard either by modifying the reductions from 3Sat or VertexCover to HamiltonianCycle, or by a direct reduction from HamiltonianCycle.
- LongestPath: Given a non-negatively weighted graph G and two vertices u and v, what is the longest simple path from u to v in the graph? A path is *simple* if it visits each vertex at most once. This problem is a generalization of the HamiltonianPath problem. Of course, the corresponding *shortest* path problem is in P.
- SteinerTree: Given a weighted, undirected graph G with some of the vertices marked, what is the minimum-weight subtree of G that contains every marked vertex? If *every* vertex is marked, the minimum Steiner tree is just the minimum spanning tree; if exactly two vertices are marked, the minimum Steiner tree is just the shortest path between them. This problem can be proved NP-hard by reduction from VertexCover.

In addition to these dry but useful problems, most interesting puzzles and solitaire games have been shown to be NP-hard, or to have NP-hard generalizations. (Arguably, if a game or puzzle isn't at least NP-hard, it isn't interesting!) Some familiar examples include:

- Minesweeper (by reduction from CircuitSAT)[8]
- Tetris (by reduction from 3Partition)[9]
- Sudoku (by a complex reduction from 3SAT)[10]
- Klondike, aka "Solitaire" (by reduction from 3SAT)[11]

[8]Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000. http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.pdf

[9]Ron Breukelaar*, Erik D. Demaine, Susan Hohenberger*, Hendrik J. Hoogeboom, Walter A. Kosters, and David Liben-Nowell*. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications* 14:41–68, 2004.

[10]Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A(5):1052–1060, 2003. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

[11]Luc Longpré and Pierre McKenzie. The complexity of Solitaire. *Proceedings of the 32nd International Mathematical Foundations of Computer Science*, 182–193, 2007.

- Flood-It (by reduction from shortest common supersequence)[12]
- Pac-Man (by reduction from HamiltonianCycle)[13]
- Super Mario Brothers (by reduction from 3SAT)[14]
- Candy Crush Saga (by reduction from a variant of 3SAT)[15]

As of November 2014, nobody has published a proof that a generalization of Threes/2048 or Cookie Clicker is NP-hard, but I'm sure it's only a matter of time.[16]

*30.15 On Beyond Zebra

P and NP are only the first two steps in an enormous hierarchy of complexity classes. To close these notes, let me describe a few more classes of interest.

Polynomial Space. ***PSPACE*** is the set of decision problems that can be solved using polynomial *space*. Every problem in NP (and therefore in P) is also in PSPACE. It is generally believed that NP $\neq$ PSPACE, but nobody can even prove that P $\neq$ PSPACE. A problem Π is ***PSPACE-hard*** if, for any problem Π' that can be solved using polynomial *space*, there is a polynomial-*time* many-one reduction from Π' to Π. A problem is ***PSPACE-complete*** if it is both PSPACE-hard and in PSPACE. If any PSPACE-hard problem is in NP, then PSPACE=NP; similarly, if any PSPACE-hard problem is in P, then PSPACE=P.

The canonical PSPACE-complete problem is the *quantified boolean formula* problem, or ***QBF***: Given a boolean formula Φ that may include any number of universal or existential quantifiers, but no free variables, is Φ equivalent to True? For example, the following expression is a valid input to QBF:

$$\exists a\colon \forall b\colon \exists c\colon (\forall d\colon a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee (\exists e\colon \overline{(\bar{a} \Rightarrow e)} \vee (c \neq a \wedge e))).$$

SAT is provably equivalent the special case of QBF where the input formula contains only existential quantifiers. QBF remains PSPACE-hard even when the input formula must have all its quantifiers at the beginning, the quantifiers strictly alternate between $\exists$ and $\forall$, and the quantified proposition is in conjunctive normal form, with exactly three literals in each clause, for example:

$$\exists a\colon \forall b\colon \exists c\colon \forall d\colon \big((a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})\big)$$

[12]Raphaël Clifford, Markus Jalsenius, Ashley Montanaro, and Benjamin Sach. The complexity of flood filling games. *Proceedings of the Fifth International Conference on Fun with Algorithms* (FUN'10), 307–318, 2010. http://arxiv.org/abs/1001.4420.

[13]Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014. http://giovanniviglietta.com/papers/gaming2.pdf

[14]Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games Are (computationally) hard. *Proceedings of the Seventh International Conference on Fun with Algorithms* (FUN'14), 2014. http://arxiv.org/abs/1203.1895.

[15]Luciano Gualà, Stefano Leucci, Emanuele Natale. Bejeweled, Candy Crush and other match-three games are (NP-)hard. Preprint, March 2014. http://arxiv.org/abs/1403.5830.

[16]Princeton freshman Rahul Mehta actually claimed a proof that a certain generalization of 2048 is PSPACE-hard, but his proof appears to be flawed. [Rahul Mehta. 2048 is (PSPACE) hard, but sometimes easy. *Electronic Colloquium on Computational Complexity*, Report No. 116, 2014. http://eccc.hpi-web.de/report/2014/116/.] On the other hand, Christopher Chen proved that a different(!) generalization of 2048 is in NP, but left the hardness question open. [Christopher Chen. 2048 is in NP. *Open Endings*, March 27, 2014. http://blog.openendings.net/2014/03/2048-is-in-np.html.]

This restricted version of QBF can also be phrased as a two-player strategy question. Suppose two players, Alice and Bob, are given a 3CNF predicate with free variables $x_1, x_2, \ldots, x_n$. The players alternately assign values to the variables in order by index—Alice assigns a value to x_1, Bob assigns a value to x_2, and so on. Alice eventually assigns values to every variable with an odd index, and Bob eventually assigns values to every variable with an even index. Alice wants to make the expression TRUE, and Bob wants to make it FALSE. Assuming Alice and Bob play perfectly, who wins this game? Not surprisingly, most two-player games[17] like tic-tac-toe, reversi, checkers, go, chess, and mancala—or more accurately, appropriate generalizations of these constant-size games to arbitrary board sizes—are PSPACE-hard.

Another canonical PSPACE-hard problem is *NFA totality*: Given a non-deterministic finite-state automaton M over some alphabet Σ, does M accept every string in Σ^*? The closely related problems *NFA equivalence* (Do two given NFAs accept the same language?) and *NFA minimization* (Find the smallest NFA that accepts the same language as a given NFA) are also PSPACE-hard, as are the corresponding questions about regular expressions. (The corresponding questions about *deterministic* finite-state automata are all solvable in polynomial time.)

Exponential time. The next significantly larger complexity class, ***EXP*** (also called EXPTIME), is the set of decision problems that can be solved in exponential time, that is, using at most 2^{n^c} steps for some constant $c > 0$. Every problem in PSPACE (and therefore in NP (and therefore in P)) is also in EXP. It is generally believed that PSPACE $\subsetneq$ EXP, but nobody can even prove that NP $\neq$ EXP. A problem Π is ***EXP-hard*** if, for any problem Π' that can be solved in *exponential* time, there is a *polynomial*-time many-one reduction from Π' to Π. A problem is ***EXP-complete*** if it is both EXP-hard and in EXP. If any EXP-hard problem is in PSPACE, then EXP=PSPACE; similarly, if any EXP-hard probelm is in NP, then EXP=NP. We *do* know that P $\neq$ EXP; in particular, no EXP-hard problem is in P.

Natural generalizations of many interesting 2-player games—like checkers, chess, mancala, and go—are actually EXP-hard. The boundary between PSPACE-complete games and EXP-hard games is rather subtle. For example, there are three ways to draw in chess (the standard 8×8 game): stalemate (the player to move is not in check but has no legal moves), repeating the same board position three times, or moving fifty times without capturing a piece. The $n \times n$ generalization of chess is either in PSPACE or EXP-hard depending on how we generalize these rules. If we declare a draw after (say) n^3 capture-free moves, then every game must end after a polynomial number of moves, so we can simulate all possible games from any given position using only polynomial space. On the other hand, if we ignore the capture-free move rule entirely, the resulting game can last an exponential number of moves, so there no obvious way to detect a repeating position using only polynomial space; indeed, this version of $n \times n$ chess is EXP-hard.

Excelsior! Naturally, even exponential time is not the end of the story. ***NEXP*** is the class of decision problems that can be solve in *nondeterministic* exponential time; equivalently, a decision problem is in NEXP if and only if, for every YES instance, there is a *proof* of this fact that can be checked in exponential time. ***EXPSPACE*** is the set of decision problems that can be solved using exponential *space*. Even these larger complexity classes have hard and complete problems; for example, if we add the intersection operator $\cap$ to the syntax of regular expressions, deciding whether two such expressions describe the same language is EXPSPACE-hard. Beyond EXPSPACE

[17]For a good (but now slightly dated) overview of known results on the computational complexity of games and puzzles, see Erik D. Demaine and Robert Hearn's survey "Playing Games with Algorithms: Algorithmic Combinatorial Game Theory" at http://arxiv.org/abs/cs.CC/0106019.

are complexity classes with *doubly*-exponential resource bounds (EEXP, NEEXP, and EEXPSPACE), then *triply* exponential resource bounds (EEEXP, NEEEXP, and EEEXPSPACE), and so on ad infinitum.

All these complexity classes can be ordered by inclusion as follows:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXP} \subseteq \text{NEXP} \subseteq \text{EXPSPACE} \subseteq \text{EEXP} \subseteq \text{NEEXP} \subseteq \text{EEXPSPACE} \subseteq \text{EEEXP} \subseteq \cdots,$$

Most complexity theorists strongly believe that every inclusion in this sequence is strict; that is, no two of these complexity classes are equal. However, the strongest result that has been proved is that every class in this sequence is strictly contained in the class *three* steps later in the sequence. For example, we have proofs that P $\neq$ EXP and PSPACE $\neq$ EXPSPACE, but not whether P $\neq$ PSPACE or NP $\neq$ EXP.

The limit of this series of increasingly exponential complexity classes is the class ***ELEMENTARY*** of decision problems that can be solved using time or space bounded by a function the form $2\uparrow^k n$ for some integer k, where

$$2\uparrow^k n := \begin{cases} n & \text{if } k = 0, \\ 2^{2\uparrow^{k-1} n} & \text{otherwise.} \end{cases}$$

For example, $2\uparrow^1 n = 2^n$ and $2\uparrow^2 n = 2^{2^n}$. You might be tempted to conjecture that every natural decidable problem can be solved in elementary time, but then you would be wrong. Consider the ***extended regular expressions*** defined by recursively combining (possibly empty) strings over some finite alphabet by concatenation (xy), union ($x + y$), Kleene closure (x^*), ***and negation*** **($\overline{x}$)**. For example, the extended regular expression $\overline{(\mathtt{0}+\mathtt{1})^*\mathtt{00}(\mathtt{0}+\mathtt{1})^*}$ represents the set of strings in $\{0,1\}^*$ that do *not* contain two 0s in a row. It is possible to determine algorithmically whether two extended regular expressions describe identical languages, by recursively converting each expression into an equivalent NFA, converting each NFA into a DFA, and then minimizing the DFA. Unfortunately, however, this problem cannot be solved in only elementary time, intuitively because each layer of recursive negation exponentially increases the number of states in the final DFA.

Exercises

1. (a) Describe and analyze and algorithm to solve PARTITION in time $O(nM)$, where n is the size of the input set and M is the sum of the absolute values of its elements.

   (b) Why doesn't this algorithm imply that P=NP?

2. Consider the following problem, called BOXDEPTH: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

   (a) Describe a polynomial-time reduction from BOXDEPTH to MAXCLIQUE.

   (b) Describe and analyze a polynomial-time algorithm for BOXDEPTH. *[Hint: $O(n^3)$ time should be easy, but $O(n \log n)$ time is possible.]*

   (c) Why don't these two results imply that P=NP?

3. A boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (OR) or several *terms*, each of which is the conjunction (AND) of one or more literals. For

example, the formula

$$(\overline{x} \wedge y \wedge \overline{z}) \vee (y \wedge z) \vee (x \wedge \overline{y} \wedge \overline{z})$$

is in disjunctive normal form. DNF-SAT asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

(a) Describe a polynomial-time algorithm to solve DNF-SAT.

(b) What is the error in the following argument that P=NP?

Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,

$$(x \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y}) \iff (x \wedge \overline{y}) \vee (y \wedge \overline{x}) \vee (\overline{z} \wedge \overline{x}) \vee (\overline{z} \wedge \overline{y})$$

Now we can use the algorithm from part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time. Since 3SAT is NP-hard, we must conclude that P=NP!

4. (a) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.

   (b) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.

5. (a) Describe a polynomial-time reduction from UNDIRECTEDHAMILTONIANCYCLE to DIRECTEDHAMILTONIANCYCLE.

   (b) Describe a polynomial-time reduction from DIRECTEDHAMILTONIANCYCLE to UNDIRECTED-HAMILTONIANCYCLE.

6. (a) Describe a polynomial-time reduction from HAMILTONIANPATH to HAMILTONIANCYCLE.

   (b) Describe a polynomial-time reduction from HAMILTONIANCYCLE to HAMILTONIANPATH. *[Hint: A polynomial-time reduction may call the black-box subroutine more than once.]*

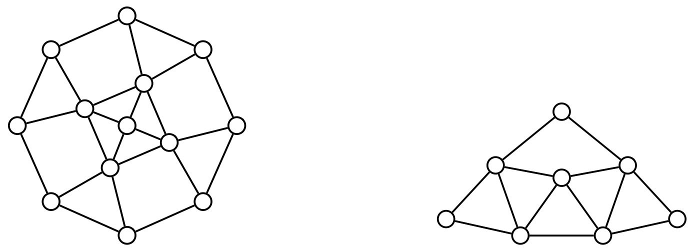
7. (a) Prove that PLANARCIRCUITSAT is NP-hard. *[Hint: Construct a gadget for crossing wires.]*

   (b) Prove that NOTALLEQUAL3SAT is NP-hard.

   (c) Prove that the following variant of 3SAT is NP-hard: Given a boolean formula Φ in conjunctive normal form where each clause contains at most 3 literals and each variable appears in at most 3 clauses, does Φ have a satisfying assignment?

8. (a) Using the gadget on the right below, prove that deciding whether a given *planar* graph is 3-colorable is NP-hard. *[Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]*

   (b) Using part (a) and the middle gadget below, prove that deciding whether a planar graph *with maximum degree 4* is 3-colorable is NP-hard. *[Hint: Replace any vertex with degree greater than 4 with a collection of gadgets connected so that no degree is greater than four.]*



(a) Gadget for planar 3-colorability. (b) Gadget for degree-4 planar 3-colorability.

9. Prove that the following problems are NP-hard.
   (a) Given two undirected graphs G and H, is G isomorphic to a subgraph of H?
   (b) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 17?
   (c) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

10. **There's something special about the number 3.**
    (a) Describe and analyze a polynomial-time algorithm for 2PARTITION. Given a set S of $2n$ positive integers, your algorithm will determine in polynomial time whether the elements of S can be split into n disjoint pairs whose sums are all equal.
    (b) Describe and analyze a polynomial-time algorithm for 2COLOR. Given an undirected graph G, your algorithm will determine in polynomial time whether G has a proper coloring that uses only two colors.
    (c) Describe and analyze a polynomial-time algorithm for 2SAT. Given a boolean formula Φ in conjunctive normal form, with exactly *two* literals per clause, your algorithm will determine in polynomial time whether Φ has a satisfying assignment.

11. **There's nothing special about the number 3.**
    (a) The problem 12PARTITION is defined as follows: Given a set S of $12n$ positive integers, determine whether the elements of S can be split into n subsets of 12 elements each whose sums are all equal. Prove that 12PARTITION is NP-hard. *[Hint: Reduce from 3PARTITION. It may be easier to consider multisets first.]*
    (b) The problem 12COLOR is defined as follows: Given an undirected graph G, determine whether we can color each vertex with one of twelve colors, so that every edge touches two different colors. Prove that 12COLOR is NP-hard. *[Hint: Reduce from 3COLOR.]*
    (c) The problem 12SAT is defined as follows: Given a boolean formula Φ in conjunctive normal form, with exactly twelve literals per clause, determine whether Φ has a satisfying assignment. Prove that 12SAT is NP-hard. *[Hint: Reduce from 3SAT.]*

*12. Describe a direct polynomial-time reduction from 4COLOR to 3COLOR. (This is a lot harder than the opposite direction.)

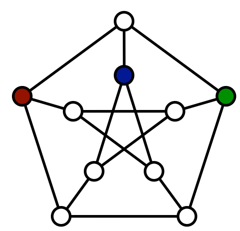
13. This exercise asks you to prove that a certain reduction from VERTEXCOVER to STEINERTREE is correct. Suppose we want to find the smallest vertex cover in a given undirected graph $G = (V, E)$. We construct a new graph $H = (V', E')$ as follows:

- $V' = V \cup E \cup \{z\}$
- $E' = \{ve \mid v \in V \text{ is an endpoint of } e \in W\} \cup \{vz \mid v \in V\}$.

Equivalently, we construct H by subdividing each edge in G with a new vertex, and then connecting all the original vertices of G to a new *apex* vertex z.

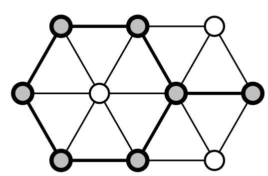
Prove that G has a vertex cover of size k if and only if there is a subtree of H with $k + |E| + 1$ vertices that contains every vertex in $E \cup \{z\}$.

14. Let $G = (V, E)$ be a graph. A ***dominating set*** in G is a subset S of the vertices such that every vertex in G is either in S or adjacent to a vertex in S. The DOMINATINGSET problem asks, given a graph G and an integer k as input, whether G contains a dominating set of size k. Prove that this problem is NP-hard.



A dominating set of size 3 in the Peterson graph.

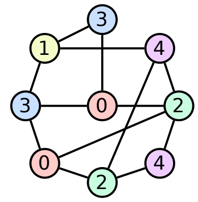
15. A subset S of vertices in an undirected graph G is called ***triangle-free*** if, for every triple of vertices $u, v, w \in S$, at least one of the three edges uv, uw, vw is *absent* from G. Prove that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.



A triangle-free subset of 7 vertices.
This is ***not*** the largest triangle-free subset in this graph.

16. *Pebbling* is a solitaire game played on an undirected graph G, where each vertex has zero or more *pebbles*. A single *pebbling move* consists of removing two pebbles from a vertex v and adding one pebble to an arbitrary neighbor of v. (Obviously, the vertex v must have at least two pebbles before the move.) The PEBBLEDESTRUCTION problem asks, given a graph $G = (V, E)$ and a pebble count $p(v)$ for each vertex v, whether is there a sequence of pebbling moves that removes all but one pebble. Prove that PEBBLEDESTRUCTION is NP-hard.

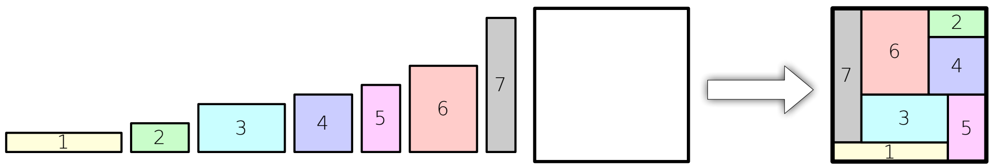
17. Recall that a 5-coloring of a graph G is a function that assigns each vertex of G an 'color' from the set $\{0, 1, 2, 3, 4\}$, such that for any edge uv, vertices u and v are assigned different 'colors'. A 5-coloring is *careful* if the colors assigned to adjacent vertices are not only



A careful 5-coloring.

distinct, but differ by more than 1 (mod 5). Prove that deciding whether a given graph has a careful 5-coloring is NP-hard. *[Hint: Reduce from the standard 5Color problem.]*

18. The RectangleTiling problem is defined as follows: Given one large rectangle and several smaller rectangles, determine whether the smaller rectangles can be placed inside the large rectangle with no gaps or overlaps. Prove that RectangleTiling is NP-hard.



A positive instance of the RectangleTiling problem.

19. For each problem below, either describe a polynomial-time algorithm or prove that the problem is NP-hard.

    (a) A *double-Eulerian circuit* in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G, does G have a double-Eulerian circuit?

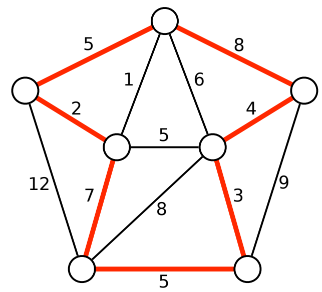
    (b) A *double-Hamiltonian circuit* in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G, does G have a double-Hamiltonian circuit?

20. (a) A *tonian path* in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-hard.

    (b) A *tonian cycle* in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-hard. *[Hint: Use part (a).]*

21. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-hard.

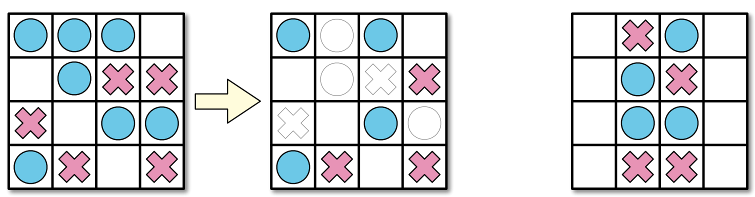
22. A boolean formula in *exclusive-or conjunctive normal form* (XCNF) is a conjunction (And) of several *clauses*, each of which is the *exclusive*-or of several literals; that is, a clause is true if and only if it contains an odd number of true literals. The XCNF-SAT problem asks



A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

whether a given XCNF formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-hard.

23. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions. An unsolvable puzzle.

Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

24. You're in charge of choreographing a musical for your local community theater, and it's time to figure out the final pose of the big show-stopping number at the end. ("Streetcar!") You've decided that each of the n cast members in the show will be positioned in a big line when the song finishes, all with their arms extended and showing off their best spirit fingers.

The director has declared that during the final flourish, each cast member must either point both their arms up or point both their arms down; it's your job to figure out who points up and who points down. Moreover, in a fit of unchecked power, the director has also given you a list of arrangements that will upset his delicate artistic temperament. Each forbidden arrangement is a subset of the cast members paired with arm positions; for example: "Marge may not point her arms up while Ned, Apu, and Smithers point their arms down."

Prove that finding an acceptable arrangement of arm positions is NP-hard.

25. The next time you are at a party, one of the guests will suggest everyone play a round of Three-Way Mumbledypeg, a game of skill and dexterity that requires three teams and

a knife. The official Rules of Three-Way Mumbledypeg (fixed during the Holy Roman Three-Way Mumbledypeg Council in 1625) require that (1) each team *must* have at least one person, (2) any two people on the same team *must* know each other, and (3) everyone watching the game *must* be on one of the three teams. Of course, it will be a really *fun* party; nobody will want to leave. There will be several pairs of people at the party who don't know each other. The host of the party, having heard thrilling tales of your prowess in all things algorithmic, will hand you a list of which pairs of party-goers know each other and ask you to choose the teams, while he sharpens the knife.

Either describe and analyze a polynomial time algorithm to determine whether the party-goers can be split into three legal Three-Way Mumbledypeg teams, or prove that the problem is NP-hard.

26. Jeff tries to make his students happy. At the beginning of class, he passes out a questionnaire that lists a number of possible course policies in areas where he is flexible. Every student is asked to respond to each possible course policy with one of "strongly favor", "mostly neutral", or "strongly oppose". Each student may respond with "strongly favor" or "strongly oppose" to at most five questions. Because Jeff's students are very understanding, each student is happy if (but only if) he or she prevails in just one of his or her strong policy preferences. Either describe a polynomial-time algorithm for setting course policy to maximize the number of happy students, or show that the problem is NP-hard.

27. The party you are attending is going great, but now it's time to line up for ***The Algorithm March (アルゴリズムこうしん)***! This dance was originally developed by the Japanese comedy duo Itsumo Kokokara (いつもここから) for the children's television show PythagoraSwitch (ピタゴラスイッチ). The Algorithm March is performed by a line of people; each person in line starts a specific sequence of movements one measure later than the person directly in front of them. Thus, the march is the dance equivalent of a musical round or canon, like "Row Row Row Your Boat".

    Proper etiquette dictates that each marcher must know the person directly in front of them in line, lest a minor mistake during lead to horrible embarrassment between strangers. Suppose you are given a complete list of which people at your party know each other. ***Prove*** that it is NP-hard to determine the largest number of party-goers that can participate in the Algorithm March. You may assume without loss of generality that there are no ninjas at your party.

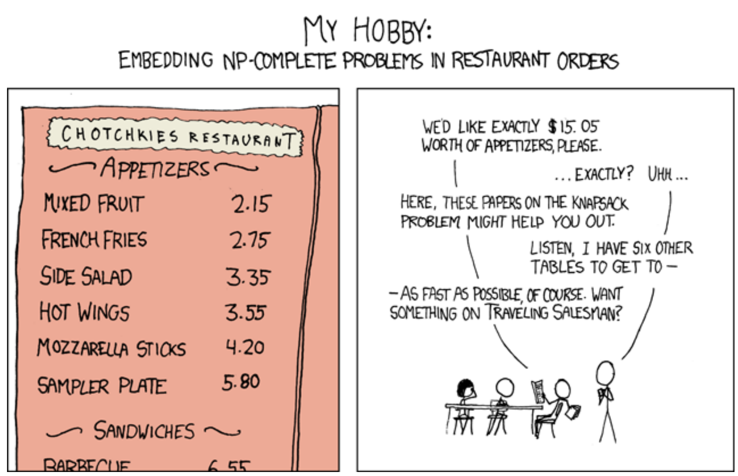
28. (a) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary weighted graph G, the length of the shortest Hamiltonian cycle in G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary weighted graph G, the shortest Hamiltonian cycle in G, using this magic black box as a subroutine.

    (b) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, the number of vertices in the largest complete subgraph of G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

(c) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, whether G is 3-colorable. Describe and analyze a **polynomial-time** algorithm that either computes a proper 3-coloring of a given graph or correctly reports that no such coloring exists, using the magic black box as a subroutine. *[Hint: The input to the magic black box is a graph. Just a graph. Vertices and edges. Nothing else.]*

(d) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary boolean formula Φ, whether Φ is satisfiable. Describe and analyze a **polynomial-time** algorithm that either computes a satisfying assignment for a given boolean formula or correctly reports that no such assignment exists, using the magic black box as a subroutine.

(e) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary set X of positive integers, whether X can be partitioned into two sets A and B such that $\sum A = \sum B$. Describe and analyze a **polynomial-time** algorithm that either computes an equal partition of a given set of positive integers or correctly reports that no such partition exists, using the magic black box as a subroutine.



[General solutions give you a 50% tip.]

— Randall Munroe, *xkcd* (http://xkcd.com/287/)

Le mieux est l'ennemi du bien. [The best is the enemy of the good.]

— Voltaire, *La Bégueule* (1772)

Who shall forbid a wise skepticism, seeing that there is no practical question on which any thing more than an approximate solution can be had?

— Ralph Waldo Emerson, *Representative Men* (1850)

Now, distrust of corporations threatens our still-tentative economic recovery; it turns out greed is bad, after all.

— Paul Krugman, "Greed is Bad", *The New York Times*, June 4, 2002.

★31 Approximation Algorithms

31.1 Load Balancing

On the future smash hit reality-TV game show *Grunt Work*, scheduled to air Thursday nights at 3am (2am Central) on ESPNπ, the contestants are given a series of utterly pointless tasks to perform. Each task has a predetermined time limit; for example, "Sharpen this pencil for 17 seconds", or "Pour pig's blood on your head and sing The Star-Spangled Banner for two minutes", or "Listen to this 75-minute algorithms lecture". The directors of the show want you to assign each task to one of the contestants, so that the last task is completed as early as possible. When your predecessor correctly informed the directors that their problem is NP-hard, he was immediately fired. "Time is money!" they screamed at him. "We don't need perfection. Wake up, dude, this is *television*!"

Less facetiously, suppose we have a set of n jobs, which we want to assign to m machines. We are given an array $T[1..n]$ of non-negative numbers, where $T[j]$ is the running time of job j. We can describe an *assignment* by an array $A[1..n]$, where $A[j] = i$ means that job j is assigned to machine i. The *makespan* of an assignment is the maximum time that any machine is busy:

$$\mathit{makespan}(A) = \max_i \sum_{A[j]=i} T[j]$$

The *load balancing* problem is to compute the assignment with the smallest possible makespan.

It's not hard to prove that the load balancing problem is NP-hard by reduction from PARTITION: The array $T[1..n]$ can be evenly partitioned if and only if there is an assignment to two machines with makespan exactly $\sum_i T[i]/2$. A slightly more complicated reduction from 3PARTITION implies that the load balancing problem is *strongly* NP-hard. If we really need the optimal solution, there is a dynamic programming algorithm that runs in time $O(nM^m)$, where M is the minimum makespan, but that's just horrible.

There is a fairly natural and efficient greedy heuristic for load balancing: consider the jobs one at a time, and assign each job to the machine i with the earliest finishing time $\mathit{Total}[i]$.

```
GreedyLoadBalance(T[1..n], m):
  for i ← 1 to m
      Total[i] ← 0

  for j ← 1 to n
      mini ← arg min_i Total[i]
      A[j] ← mini
      Total[mini] ← Total[mini] + T[j]

  return A[1..m]
```

Theorem 1. *The makespan of the assignment computed by* GreedyLoadBalance *is at most twice the makespan of the optimal assignment.*

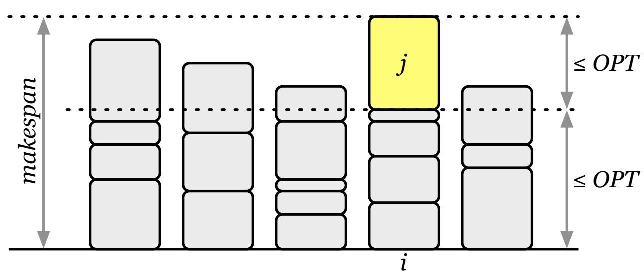
Proof: Fix an arbitrary input, and let *OPT* denote the makespan of its optimal assignment. The approximation bound follows from two trivial observations. First, the makespan of any assignment (and therefore of the optimal assignment) is at least the duration of the longest job. Second, the makespan of any assignment is at least the total duration of all the jobs divided by the number of machines.

$$OPT \geq \max_j T[j] \qquad \text{and} \qquad OPT \geq \frac{1}{m}\sum_{j=1}^{n} T[j]$$

Now consider the assignment computed by GreedyLoadBalance. Suppose machine i has the largest total running time, and let j be the last job assigned to machine i. Our first trivial observation implies that $T[j] \leq OPT$. To finish the proof, we must show that $Total[i] - T[j] \leq OPT$. Job j was assigned to machine i because it had the smallest finishing time, so $Total[i] - T[j] \leq Total[k]$ for all k. (Some values $Total[k]$ may have increased since job j was assigned, but that only helps us.) In particular, $Total[i] - T[j]$ is less than or equal to the *average* finishing time over all machines. Thus,

$$Total[i] - T[j] \leq \frac{1}{m}\sum_{i=1}^{m} Total[i] = \frac{1}{m}\sum_{j=1}^{n} T[j] \leq OPT$$

by our second trivial observation. We conclude that the makespan $Total[i]$ is at most $2 \cdot OPT$. □



Proof that GreedyLoadBalance is a 2-approximation algorithm

GreedyLoadBalance is an *online* algorithm: It assigns jobs to machines in the order that the jobs appear in the input array. Online approximation algorithms are useful in settings where inputs arrive in a stream of unknown length—for example, real jobs arriving at a real scheduling algorithm. In this online setting, it may be *impossible* to compute an optimum solution, even in cases where the offline problem (where all inputs are known in advance) can be solved in polynomial time. The study of online algorithms could easily fill an entire one-semester course (alas, not this one).

In our original offline setting, we can improve the approximation factor by sorting the jobs before piping them through the greedy algorithm.

```
SortedGreedyLoadBalance(T[1..n], m):
  sort T in decreasing order
  return GreedyLoadBalance(T, m)
```

Theorem 2. *The makespan of the assignment computed by* SORTEDGREEDYLOADBALANCE *is at most* $3/2$ *times the makespan of the optimal assignment.*

Proof: Let i be the busiest machine in the schedule computed by SORTEDGREEDYLOADBALANCE. If only one job is assigned to machine i, then the greedy schedule is actually optimal, and the theorem is trivially true. Otherwise, let j be the last job assigned to machine i. Since each of the first m jobs is assigned to a unique machine, we must have $j \geq m+1$. As in the previous proof, we know that $Total[i] - T[j] \leq OPT$.

In *any* schedule, at least two of the first $m+1$ jobs, say jobs k and ℓ, must be assigned to the same machine. Thus, $T[k] + T[\ell] \leq OPT$. Since $\max\{k, \ell\} \leq m+1 \leq j$, and the jobs are sorted in decreasing order by duration, we have

$$T[j] \leq T[m+1] \leq T[\max\{k,\ell\}] = \min\{T[k], T[\ell]\} \leq OPT/2.$$

We conclude that the makespan $Total[i]$ is at most $3 \cdot OPT/2$, as claimed. □

31.2 Generalities

Consider an arbitrary optimization problem. Let $OPT(X)$ denote the value of the optimal solution for a given input X, and let $A(X)$ denote the value of the solution computed by algorithm A given the same input X. We say that A is an ***$\alpha(n)$-approximation algorithm*** if and only if

$$\frac{OPT(X)}{A(X)} \leq \alpha(n) \quad \text{and} \quad \frac{A(X)}{OPT(X)} \leq \alpha(n)$$

for all inputs X of size n. The function $\alpha(n)$ is called the ***approximation factor*** for algorithm A. For any given algorithm, only one of these two inequalities will be important. For maximization problems, where we want to compute a solution whose cost is as small as possible, the first inequality is trivial. For maximization problems, where we want a solution whose value is as large as possible, the second inequality is trivial. A 1-approximation algorithm always returns the exact optimal solution.

Especially for problems where exact optimization is NP-hard, we have little hope of completely characterizing the optimal solution. The secret to proving that an algorithm satisfies some approximation ratio is to find a useful function of the input that provides both lower bounds on the cost of the optimal solution and upper bounds on the cost of the approximate solution. For example, if $OPT(X) \geq f(X)/2$ and $A(X) \leq 5f(X)$ for *any* function f, then A is a 10-approximation algorithm. Finding the right intermediate function can be a delicate balancing act.

31.3 Greedy Vertex Cover

Recall that the *vertex color* problem asks, given a graph G, for the smallest set of vertices of G that cover every edge. This is one of the first NP-hard problems introduced in the first week of class. There is a natural and efficient greedy heuristic[1] for computing a small vertex cover: mark the vertex with the largest degree, remove all the edges incident to that vertex, and recurse.

[1]A *heuristic* is an algorithm that doesn't work.

```
GreedyVertexCover(G):
  C ← ∅
  while G has at least one edge
      v ← vertex in G with maximum degree
      G ← G \ v
      C ← C ∪ v
  return C
```

Obviously this algorithm doesn't compute the optimal vertex cover—that would imply P=NP!—but it does compute a reasonably close approximation.

Theorem 3. *GreedyVertexCover is an $O(\log n)$-approximation algorithm.*

Proof: For all i, let G_i denote the graph G after i iterations of the main loop, and let d_i denote the maximum degree of any node in G_{i-1}. We can define these variables more directly by adding a few extra lines to our algorithm:

```
GreedyVertexCover(G):
  C ← ∅
  G_0 ← G
  i ← 0
  while G_i has at least one edge
      i ← i + 1
      v_i ← vertex in G_{i-1} with maximum degree
      d_i ← deg_{G_{i-1}}(v_i)
      G_i ← G_{i-1} \ v_i
      C ← C ∪ v_i
  return C
```

Let $|G_{i-1}|$ denote the number of edges in the graph G_{i-1}. Let C^* denote the optimal vertex cover of G, which consists of OPT vertices. Since C^* is also a vertex cover for G_{i-1}, we have

$$\sum_{v\in C^*} \deg_{G_{i-1}}(v) \geq |G_{i-1}|.$$

In other words, the *average* degree in G_i of any node in C^* is at least $|G_{i-1}|/OPT$. It follows that G_{i-1} has at least one node with degree at least $|G_{i-1}|/OPT$. Since d_i is the maximum degree of any node in G_{i-1}, we have

$$d_i \geq \frac{|G_{i-1}|}{OPT}$$

Moreover, for any $j \geq i-1$, the subgraph G_j has no more edges than G_{i-1}, so $d_i \geq |G_j|/OPT$. This observation implies that

$$\sum_{i=1}^{OPT} d_i \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{i-1}|}{OPT} \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{OPT}|}{OPT} \;=\; |G_{OPT}| \;=\; |G| - \sum_{i=1}^{OPT} d_i.$$

In other words, the first OPT iterations of GreedyVertexCover remove at least half the edges of G. Thus, after at most $OPT \lg|G| \leq 2\,OPT \lg n$ iterations, all the edges of G have been removed, and the algorithm terminates. We conclude that GreedyVertexCover computes a vertex cover of size $O(OPT \log n)$. □

So far we've only proved an *upper bound* on the approximation factor of GreedyVertexCover; perhaps a more careful analysis would imply that the approximation factor is only $O(\log\log n)$, or even $O(1)$. Alas, no such improvement is possible. For any integer n, a simple recursive construction gives us an n-vertex graph for which the greedy algorithm returns a vertex cover of size $\Omega(OPT \cdot \log n)$. Details are left as an exercise for the reader.

31.4 Set Cover and Hitting Set

The greedy algorithm for vertex cover can be applied almost immediately to two more general problems: *set cover* and *hitting set*. The input for both of these problems is a *set system* $(X, \mathcal{F})$, where X is a finite *ground set*, and $\mathcal{F}$ is a family of subsets of X.[2] A *set cover* of a set system $(X, \mathcal{F})$ is a subfamily of sets in $\mathcal{F}$ whose union is the entire ground set X. A *hitting set* for $(X, \mathcal{F})$ is a subset of the ground set X that intersects every set in $\mathcal{F}$.

An undirected graph can be cast as a set system in two different ways. In one formulation, the ground set X contains the vertices, and each edge defines a set of two vertices in $\mathcal{F}$. In this formulation, a vertex cover is a hitting set. In the other formulation, the *edges* are the ground set, the *vertices* define the family of subsets, and a vertex cover is a set cover.

Here are the natural greedy algorithms for finding a small set cover and finding a small hitting set. GreedySetCover finds a set cover whose size is at most $O(\log|\mathcal{F}|)$ times the size of smallest set cover. GreedyHittingSet finds a hitting set whose size is at most $O(\log|X|)$ times the size of the smallest hitting set.

```
GreedySetCover(X, F):
  C ← ∅
  while X ≠ ∅
      S ← arg max_{S∈F} |S ∩ X|
      X ← X \ S
      C ← C ∪ {S}
  return C
```

```
GreedyHittingSet(X, F):
  H ← ∅
  while F ≠ ∅
      x ← arg max_{x∈X} |{S ∈ F | x ∈ S}|
      F ← F \ {S ∈ F | x ∈ S}
      H ← H ∪ {x}
  return H
```

The similarity between these two algorithms is no coincidence. For any set system $(X, \mathcal{F})$, there is a *dual* set system $(\mathcal{F}, X^*)$ defined as follows. For any element $x \in X$ in the ground set, let x^* denote the subfamily of sets in $\mathcal{F}$ that contain x:

$$x^* = \{S \in \mathcal{F} \mid x \in S\}.$$

Finally, let X^* denote the collection of all subsets of the form x^*:

$$X^* = \{x^* \mid x \in S\}.$$

As an example, suppose X is the set of letters of alphabet and $\mathcal{F}$ is the set of last names of student taking CS 573 this semester. Then X^* has 26 elements, each containing the subset of CS 573 students whose last name contains a particular letter of the alphabet. For example, m^* is the set of students whose last names contain the letter m.

There is a direct one-to-one correspondence between the ground set X and the dual set family X^*. It is a tedious but instructive exercise to prove that the dual of the dual of any set system is isomorphic to the original set system—$(X^*, \mathcal{F}^*)$ is essentially the same as $(X, \mathcal{F})$. It is also easy to prove that a set cover for any set system $(X, \mathcal{F})$ is also a hitting set for the dual set system $(\mathcal{F}, X^*)$, and therefore a hitting set for any set system $(X, \mathcal{F})$ is isomorphic to a set cover for the dual set system $(\mathcal{F}, X^*)$.

31.5 Vertex Cover, Again

The greedy approach doesn't always lead to the best approximation algorithms. Consider the following alternate heuristic for vertex cover:

[2] A matroid (see the lecture note on greedy algorithms) is a special type of set system.

```
DumbVertexCover(G):
  C ← ∅
  while G has at least one edge
      (u, v) ← any edge in G
      G ← G \ {u, v}
      C ← C ∪ {u, v}
  return C
```

The minimum vertex cover—in fact, *every* vertex cover—contains at least one of the two vertices u and v chosen inside the while loop. It follows immediately that DumbVertexCover is a 2-approximation algorithm!

The same idea can be extended to approximate the minimum hitting set for any set system $(X, \mathcal{F})$, where the approximation factor is the size of the largest set in $\mathcal{F}$.

31.6 Traveling Salesman: The Bad News

The *traveling salesman problem*[3] problem asks for the shortest Hamiltonian cycle in a weighted undirected graph. To keep the problem simple, we can assume without loss of generality that the underlying graph is always the complete graph K_n for some integer n; thus, the input to the traveling salesman problem is just a list of the $\binom{n}{2}$ edge lengths.

Not surprisingly, given its similarity to the Hamiltonian cycle problem, it's quite easy to prove that the traveling salesman problem is NP-hard. Let G be an arbitrary undirected graph with n vertices. We can construct a length function for K_n as follows:

$$\ell(e) = \begin{cases} 1 & \text{if } e \text{ is an edge in } G, \\ 2 & \text{otherwise.} \end{cases}$$

Now it should be obvious that if G has a Hamiltonian cycle, then there is a Hamiltonian cycle in K_n whose length is exactly n; otherwise every Hamiltonian cycle in K_n has length at least $n+1$. We can clearly compute the lengths in polynomial time, so we have a polynomial time reduction from Hamiltonian cycle to traveling salesman. Thus, the traveling salesman problem is NP-hard, even if all the edge lengths are 1 and 2.

There's nothing special about the values 1 and 2 in this reduction; we can replace them with any values we like. By choosing values that are sufficiently far apart, we can show that even approximating the shortest traveling salesman tour is NP-hard. For example, suppose we set the length of the 'absent' edges to $n+1$ instead of 2. Then the shortest traveling salesman tour in the resulting weighted graph either has length exactly n (if G has a Hamiltonian cycle) or has length at least $2n$ (if G does not have a Hamiltonian cycle). Thus, if we could approximate the shortest traveling salesman tour within a factor of 2 in polynomial time, we would have a polynomial-time algorithm for the Hamiltonian cycle problem.

Pushing this idea to its limits us the following negative result.

Theorem 4. *For any function $f(n)$ that can be computed in time polynomial in n, there is no polynomial-time $f(n)$-approximation algorithm for the traveling salesman problem on general weighted graphs, unless P=NP.*

[3]This is sometimes bowdlerized into the traveling sales*person* problem. That's just silly. Who ever heard of a traveling salesperson sleeping with the farmer's child?

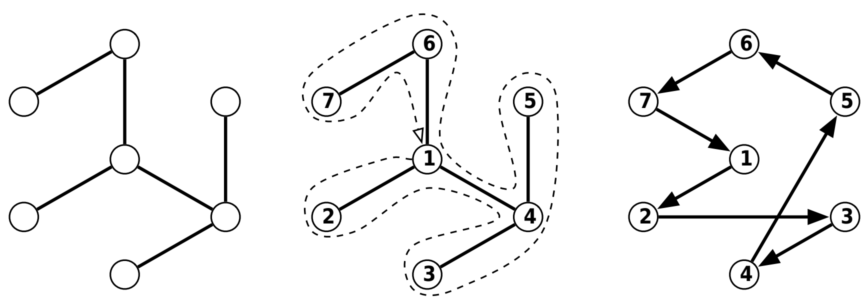
31.7 Traveling Salesman: The Good News

Even though the general traveling salesman problem can't be approximated, a common special case can be approximated fairly easily. The special case requires the edge lengths to satisfy the so-called *triangle inequality*:

$$\ell(u,w) \leq \ell(u,v) + \ell(v,w) \quad \text{for any vertices } u,v,w.$$

This inequality is satisfied for *geometric graphs*, where the vertices are points in the plane (or some higher-dimensional space), edges are straight line segments, and lengths are measured in the usual Euclidean metric. Notice that the length functions we used above to show that the general TSP is hard to approximate do not (always) satisfy the triangle inequality.

With the triangle inequality in place, we can quickly compute a 2-approximation for the traveling salesman tour as follows. First, we compute the minimum spanning tree T of the weighted input graph; this can be done in $O(n^2 \log n)$ time (where n is the number of vertices of the graph) using any of several classical algorithms. Second, we perform a depth-first traversal of T, numbering the vertices in the order that we first encounter them. Because T is a spanning tree, every vertex is numbered. Finally, we return the cycle obtained by visiting the vertices according to this numbering.



A minimum spanning tree T, a depth-first traversal of T, and the resulting approximate traveling salesman tour.

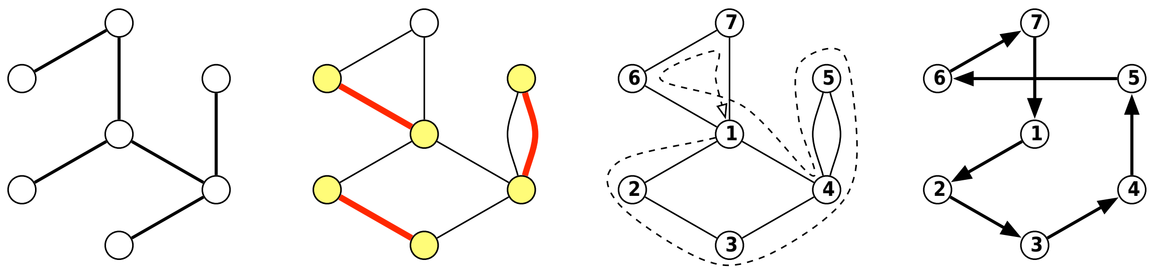
Theorem 5. *A depth-first ordering of the minimum spanning tree gives a 2-approximation of the shortest traveling salesman tour.*

Proof: Let *OPT* denote the cost of the optimal TSP tour, let *MST* denote the total length of the minimum spanning tree, and let A be the length of the tour computed by our approximation algorithm. Consider the 'tour' obtained by walking through the minimum spanning tree in depth-first order. Since this tour traverses every edge in the tree exactly twice, its length is $2 \cdot MST$. The final tour can be obtained from this one by removing duplicate vertices, moving directly from each node to the next *unvisited* node.; the triangle inequality implies that taking these shortcuts cannot make the tour longer. Thus, $A \leq 2 \cdot MST$. On the other hand, if we remove any edge from the optimal tour, we obtain a spanning tree (in fact a spanning *path*) of the graph; thus, $MST \geq OPT$. We conclude that $A \leq 2 \cdot OPT$; our algorithm computes a 2-approximation of the optimal tour. □

We can improve this approximation factor using the following algorithm discovered by Nicos Christofides in 1976. As in the previous algorithm, we start by constructing the minimum spanning tree T. Then let O be the set of vertices with *odd* degree in T; it is an easy exercise (hint, hint) to show that the number of vertices in O is even.

In the next stage of the algorithm, we compute a *minimum-cost perfect matching* M of these odd-degree vertices. A prefect matching is a collection of edges, where each edge has both endpoints in O and each vertex in O is adjacent to exactly one edge; we want the perfect matching of minimum total length. Later in the semester, we will see an algorithm to compute M in polynomial time.

Now consider the multigraph $T \cup M$; any edge in both T and M appears twice in this multigraph. This graph is connected, and every vertex has even degree. Thus, it contains an *Eulerian circuit*: a closed walk that uses every edge exactly once. We can compute such a walk in $O(n)$ time with a simple modification of depth-first search. To obtain the final approximate TSP tour, we number the vertices in the order they first appear in some Eulerian circuit of $T \cup M$, and return the cycle obtained by visiting the vertices according to that numbering.



A minimum spanning tree T, a minimum-cost perfect matching M of the odd vertices in T, an Eulerian circuit of $T \cup M$, and the resulting approximate traveling salesman tour.

Theorem 6. *Given a weighted graph that obeys the triangle inequality, the Christofides heuristic computes a (3/2)-approximation of the shortest traveling salesman tour.*

Proof: Let A denote the length of the tour computed by the Christofides heuristic; let OPT denote the length of the optimal tour; let MST denote the total length of the minimum spanning tree; let MOM denote the total length of the minimum odd-vertex matching.

The graph $T \cup M$, and therefore any Euler tour of $T \cup M$, has total length $MST + MOM$. By the triangle inequality, taking a shortcut past a previously visited vertex can only shorten the tour. Thus, $A \leq MST + MOM$.

By the triangle inequality, the optimal tour of the odd-degree vertices of T cannot be longer than OPT. Any cycle passing through of the odd vertices can be partitioned into two perfect matchings, by alternately coloring the edges of the cycle red and green. One of these two matchings has length at most $OPT/2$. On the other hand, both matchings have length at least MOM. Thus, $MOM \leq OPT/2$.

Finally, recall our earlier observation that $MST \leq OPT$.

Putting these three inequalities together, we conclude that $A \leq 3 \cdot OPT/2$, as claimed. □

31.8 *k*-center Clustering

The *k-center clustering* problem is defined as follows. We are given a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane[4] and an integer k. Our goal to find a collection of k circles that collectively enclose all the input points, such that the radius of the largest circle is as large as possible. More

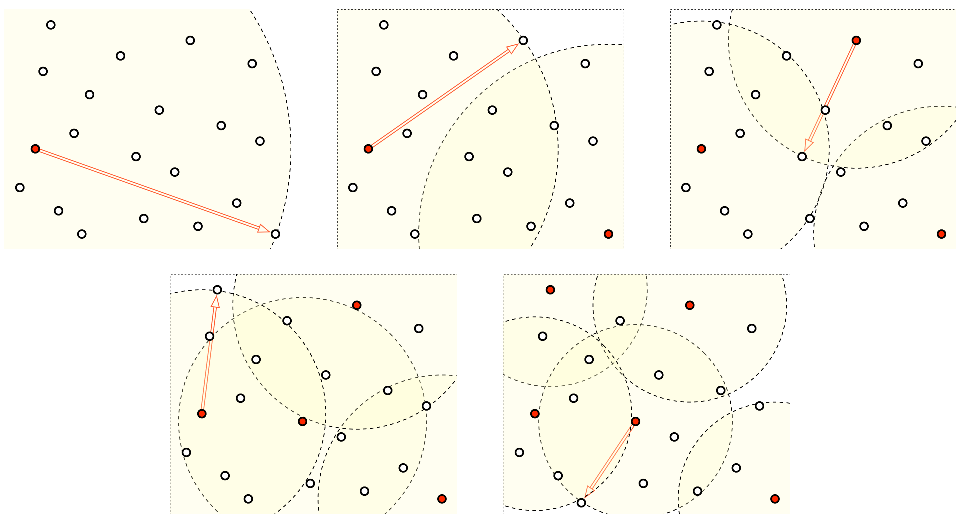
[4]The k-center problem can be defined over *any* metric space, and the approximation analysis in this section holds in any metric space as well. The analysis in the *next* section, however, does require that the points come from the Euclidean plane.

formally, we want to compute a set $C = \{c_1, c_2, \ldots, c_k\}$ of k *center points*, such that the following cost function is minimized:

$$cost(C) := \max_i \min_j |p_i\, c_j|.$$

Here, $|p_i\, c_j|$ denotes the Euclidean distance between input point p_i and center point c_j. Intuitively, each input point is assigned to its closest center point; the points assigned to a given center c_j comprise a *cluster*. The distance from c_j to the farthest point in its cluster is the *radius* of that cluster; the cluster is contained in a circle of this radius centered at c_j. The k-center clustering cost $cost(C)$ is precisely the maximum cluster radius.

This problem turns out to be NP-hard, even to approximate within a factor of roughly 1.8. However, there is a natural greedy strategy, first analyzed in 1985 by Teofilo Gonzalez[5], that is guaranteed to produce a clustering whose cost is at most twice optimal. Choose the k center points one at a time, starting with an arbitrary input point as the first center. In each iteration, choose the input point that is farthest from any earlier center point to be the next center point.



The first five iterations of Gonzalez's k-center clustering algorithm.

In the pseudocode below, d_i denotes the current distance from point p_i to its nearest center, and r_j denotes the maximum of all d_i (or in other words, the cluster radius) after the first j centers have been chosen. The algorithm includes an extra iteration to compute the final clustering radius r_k (and the next center c_{k+1}).

```
GonzalezKCenter(P, k):
  for i ← 1 to n
      d_i ← ∞
  c_1 ← p_1
  for j ← 1 to k
      r_j ← 0
      for i ← 1 to n
          d_i ← min{d_i, |p_i c_j|}
          if r_j < d_i
              r_j ← d_i;  c_{j+1} ← p_i
  return {c_1, c_2, ..., c_k}
```

[5]Teofilo F. Gonzalez. Clustering to minimize the maximum inter-cluster distance. *Theoretical Computer Science* 38:293-306, 1985.

GonzalezKCenter clearly runs in $O(nk)$ time. Using more advanced data structures, Tomas Feder and Daniel Greene[6] described an algorithm to compute exactly the same clustering in only $O(n \log k)$ time.

Theorem 7. *GonzalezKCenter computes a 2-approximation to the optimal k-center clustering.*

Proof: Let *OPT* denote the optimal k-center clustering radius for P. For any index i, let c_i and r_i denote the ith center point and ith clustering radius computed by GonzalezKCenter.

By construction, each center point c_j has distance at least r_{j-1} from any center point c_i with $i < j$. Moreover, for any $i < j$, we have $r_i \geq r_j$. Thus, $|c_i c_j| \geq r_k$ for all indices i and j.

On the other hand, at least one cluster in the optimal clustering contains at least two of the points $c_1, c_2, \ldots, c_{k+1}$. Thus, by the triangle inequality, we must have $|c_i c_j| \leq 2 \cdot OPT$ for some indices i and j. We conclude that $r_k \leq 2 \cdot OPT$, as claimed. □

*31.9 Approximation Schemes

With just a little more work, we can compute an arbitrarily close approximation of the optimal k-clustering, using a so-called *approximation scheme*. An approximation scheme accepts both an instance of the problem and a value $\varepsilon > 0$ as input, and it computes a $(1+\varepsilon)$-approximation of the optimal output for that instance. As I mentioned earlier, computing even a 1.8-approximation is NP-hard, so we cannot expect our approximation scheme to run in polynomial time; nevertheless, at least for small values of k, the approximation scheme will be considerably more efficient than any exact algorithm.

Our approximation scheme works in three phases:

1. Compute a 2-approximate clustering of the input set P using GonzalezKCenter. Let r be the cost of this clustering.

2. Create a regular grid of squares of width $\delta = \varepsilon r / 2\sqrt{2}$. Let Q be a subset of P containing one point from each non-empty cell of this grid.

3. Compute an *optimal* set of k centers for Q. Return these k centers as the approximate k-center clustering for P.

The first phase requires $O(nk)$ time. By our earlier analysis, we have $r^* \leq r \leq 2r^*$, where r^* is the optimal k-center clustering cost for P.

The second phase can be implemented in $O(n)$ time using a hash table, or in $O(n \log n)$ time by standard sorting, by associating approximate coordinates $(\lfloor x/\delta \rfloor, \lfloor y/\delta \rfloor)$ to each point $(x, y) \in P$ and removing duplicates. The key observation is that the resulting point set Q is significantly smaller than P. We know P can be covered by k balls of radius r^*, each of which touches $O(r^*/\delta^2) = O(1/\varepsilon^2)$ grid cells. It follows that $|Q| = O(k/\varepsilon^2)$.

Let $T(n, k)$ be the running time of an *exact* k-center clustering algorithm, given n points as input. If this were a computational geometry class, we might see a "brute force" algorithm that runs in time $T(n, k) = O(n^{k+2})$; the fastest algorithm currently known[7] runs in time $T(n, k) = n^{O(\sqrt{k})}$. If we use this algorithm, our third phase requires $(k/\varepsilon^2)^{O(\sqrt{k})}$ time.

[6]Tomas Feder* and Daniel H. Greene. Optimal algorithms for approximate clustering. *Proc. 20th STOC*, 1988. Unlike Gonzalez's algorithm, Feder and Greene's faster algorithm does not work over arbitrary metric spaces; it requires that the input points come from some $\mathbb{R}^d$ and that distances are measured in some L_p metric. The time analysis also assumes that the distance between any two points can be computed in $O(1)$ time.

[7]R. Z. Hwang, R. C. T. Lee, and R. C. Chan. The slab dividing approach to solve the Euclidean p-center problem. *Algorithmica* 9(1):1–22, 1993.

It remains to show that the optimal clustering for Q implies a $(1+\varepsilon)$-approximation of the optimal clustering for P. Suppose the optimal clustering of Q consists of k balls $B_1, B_2, \ldots, B_k$, each of radius $\tilde{r}$. Clearly $\tilde{r} \le r^*$, since any set of k balls that cover P also cover any subset of P. Each point in $P \setminus Q$ shares a grid cell with some point in Q, and therefore is within distance $\delta\sqrt{2}$ of some point in Q. Thus, if we increase the radius of each ball B_i by $\delta\sqrt{2}$, the expanded balls must contain every point in P. We conclude that the optimal centers for Q gives us a k-center clustering for P of cost at most $r^* + \delta\sqrt{2} \le r^* + \varepsilon r/2 \le r^* + \varepsilon r^* = (1+\varepsilon)r^*$.

The total running time of the approximation scheme is $O(nk + (k/\varepsilon^2)^{O(\sqrt{k})})$. This is still exponential in the input size if k is large (say $\sqrt{n}$ or $n/100$), but if k and ε are fixed constants, the running time is linear in the number of input points.

*31.10 An FPTAS for Subset Sum

An approximation scheme whose running time, for any fixed ε, is polynomial in n is called a *polynomial-time approximation scheme* or *PTAS* (usually pronounced "pee taz"). If in addition the running time depends only polynomially on ε, the algorithm is called a *fully* polynomial-time approximation scheme or *FPTAS* (usually pronounced "eff pee taz"). For example, an approximation scheme with running time $O(n^2/\varepsilon^2)$ is an FPTAS; an approximation scheme with running time $O(n^{1/\varepsilon^6})$ is a PTAS but not an FPTAS; and our approximation scheme for k-center clustering is not a PTAS.

The last problem we'll consider is the SubsetSum problem: Given a set X containing n positive integers and a target integer t, determine whether X has a subset whose elements sum to t. The lecture notes on NP-completeness include a proof that SubsetSum is NP-hard. As stated, this problem doesn't allow any sort of approximation—the answer is either True or False.[8] So we will consider a related optimization problem instead: Given set X and integer t, find the subset of X whose sum is as large as possible but no larger than t.

We have already seen a dynamic programming algorithm to solve the decision version SubsetSum in time $O(nt)$; a similar algorithm solves the optimization version in the same time bound. Here is a different algorithm, whose running time does not depend on t:

```
SubsetSum(X[1..n], t):
  S_0 ← {0}
  for i ← 1 to n
      S_i ← S_{i−1} ∪ (S_{i−1} + X[i])
      remove all elements of S_i bigger than t
  return max S_n
```

Here $S_{i-1} + X[i]$ denotes the set $\{s + X[i] \mid s \in S_{i-1}\}$. If we store each S_i in a sorted array, the ith iteration of the for-loop requires time $O(|S_{i-1}|)$. Each set S_i contains all possible subset sums for the first i elements of X; thus, S_i has at most 2^i elements. On the other hand, since every element of S_i is an integer between 0 and t, we also have $|S_i| \le t+1$. It follows that the total running time of this algorithm is $\sum_{i=1}^{n} O(|S_{i-1}|) = O(\min\{2^n, nt\})$.

Of course, this is only an estimate of worst-case behavior. If several subsets of X have the same sum, the sets S_i will have fewer elements, and the algorithm will be faster. The key idea for finding an approximate solution quickly is to 'merge' nearby elements of S_i—if two subset sums are nearly equal, ignore one of them. On the one hand, merging similar subset sums will introduce some error into the output, but hopefully not too much. On the other hand, by

[8] Do, or do not. There is no 'try'. (Are old one thousand when years you, alphabetical also in order talk will you.)

reducing the size of the set of sums we need to maintain, we will make the algorithm faster, hopefully significantly so.

Here is our approximation algorithm. We make only two changes to the exact algorithm: an initial sorting phase and an extra FILTERing step inside the main loop.

```
FILTER(Z[1..k], δ):
  SORT(Z)
  j ← 1
  Y[j] ← Z[i]
  for i ← 2 to k
      if Z[i] > (1 + δ) · Y[j]
          j ← j + 1
          Y[j] ← Z[i]
  return Y[1..j]
```

```
APPROXSUBSETSUM(X[1..n], k, ε):
  SORT(X)
  R_0 ← {0}
  for i ← 1 to n
      R_i ← R_{i−1} ∪ (R_{i−1} + X[i])
      R_i ← FILTER(R_i, ε/2n)
      remove all elements of R_i bigger than t
  return max R_n
```

Theorem 8. *APPROXSUBSETSUM returns a $(1+\varepsilon)$-approximation of the optimal subset sum, given any ε such that $0 < \varepsilon \leq 1$.*

Proof: The theorem follows from the following claim, which we prove by induction:

$$\boxed{\text{For any element } s \in S_i \text{, there is an element } r \in R_i \text{ such that } r \leq s \leq r \cdot (1 + \varepsilon n/2)^i.}$$

The claim is trivial for $i = 0$. Let s be an arbitrary element of S_i, for some $i > 0$. There are two cases to consider: either $x \in S_{i-1}$, or $x \in S_{i-1} + x_i$.

(1) Suppose $s \in S_{i-1}$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \leq s \leq r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' \in R_i$, the claim obviously holds. On the other hand, if $r' \notin R_i$, there must be an element $r \in R_i$ such that $r < r' \leq r(1 + \varepsilon n/2)$, which implies that

$$r < r' \leq s \leq r' \cdot (1 + \varepsilon n/2)^{i-1} \leq r \cdot (1 + \varepsilon n/2)^i,$$

so the claim holds.

(2) Suppose $s \in S_{i-1} + x_i$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \leq s - x_i \leq r' \cdot (1 + \varepsilon n/2)^{i-1}$. If $r' + x_i \in R_i$, the claim obviously holds. On the other hand, if $r' + x_i \notin R_i$, there must be an element $r \in R_i$ such that $r < r' + x_i \leq r(1 + \varepsilon n/2)$, which implies that

$$\begin{aligned} r < r' + x_i \leq s &\leq r' \cdot (1 + \varepsilon n/2)^{i-1} + x_i \\ &\leq (r - x_i) \cdot (1 + \varepsilon n/2)^i + x_i \\ &\leq r \cdot (1 + \varepsilon n/2)^i - x_i \cdot ((1 + \varepsilon n/2)^i - 1) \\ &\leq r \cdot (1 + \varepsilon n/2)^i. \end{aligned}$$

so the claim holds.

Now let $s^* = \max S_n$ and $r^* = \max R_n$. Clearly $r^* \leq s^*$, since $R_n \subseteq S_n$. Our claim implies that there is some $r \in R_n$ such that $s^* \leq r \cdot (1 + \varepsilon/2n)^n$. But r cannot be bigger than r^*, so $s^* \leq r^* \cdot (1 + \varepsilon/2n)^n$. The inequalities $e^x \geq 1 + x$ for all x, and $e^x \leq 2x + 1$ for all $0 \leq x \leq 1$, imply that $(1 + \varepsilon/2n)^n \leq e^{\varepsilon/2} \leq 1 + \varepsilon$. □

Theorem 9. *APPROXSUBSETSUM runs in $O((n^3 \log n)/\varepsilon)$ time.*

Proof: Assuming we keep each set R_i in a sorted array, we can merge the two sorted arrays R_{i-1} and $R_{i-1} + x_i$ in $O(|R_{i-1}|)$ time. FILTERin R_i and removing elements larger than t also requires only $O(|R_{i-1}|)$ time. Thus, the overall running time of our algorithm is $O(\sum_i |R_i|)$; to express this in terms of n and ε, we need to prove an upper bound on the size of each set R_i.

Let $\delta = \varepsilon/2n$. Because we consider the elements of X in increasing order, every element of R_i is between 0 and $i \cdot x_i$. In particular, every element of $R_{i-1} + x_i$ is between x_i and $i \cdot x_i$. After FILTERing, at most one element $r \in R_i$ lies in the range $(1+\delta)^k \le r < (1+\delta)^{k+1}$, for any k. Thus, at most $\lceil \log_{1+\delta} i \rceil$ elements of $R_{i-1} + x_i$ survive the call to FILTER. It follows that

$$\begin{aligned}
|R_i| &= |R_{i-1}| + \left\lceil \frac{\log i}{\log(1+\delta)} \right\rceil \\
&\le |R_{i-1}| + \left\lceil \frac{\log n}{\log(1+\delta)} \right\rceil && [i \le n] \\
&\le |R_{i-1}| + \left\lceil \frac{2\ln n}{\delta} \right\rceil && [e^x \le 1 + 2x \text{ for all } 0 \le x \le 1] \\
&\le |R_{i-1}| + \left\lceil \frac{n \ln n}{\varepsilon} \right\rceil && [\delta = \varepsilon/2n]
\end{aligned}$$

Unrolling this recurrence into a summation gives us the upper bound $|R_i| \le i \cdot \lceil (n \ln n)/\varepsilon \rceil = O((n^2 \log n)/\varepsilon)$.

We conclude that the overall running time of APPROXSUBSETSUM is $O((n^3 \log n)/\varepsilon)$, as claimed. □

Exercises

1. (a) Prove that for any set of jobs, the makespan of the greedy assignment is at most $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.

   (b) Describe a set of jobs such that the makespan of the greedy assignment is exactly $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.

   (c) Describe an efficient algorithm to solve the minimum makespan scheduling problem *exactly* if every processing time $T[i]$ is a power of two.

2. (a) Find the smallest graph (minimum number of edges) for which GREEDYVERTEXCOVER does not return the smallest vertex cover.

   (b) For any integer n, describe an n-vertex graph for which GREEDYVERTEXCOVER returns a vertex cover of size $OPT \cdot \Omega(\log n)$.

3. (a) Find the smallest graph (minimum number of edges) for which DUMBVERTEXCOVER does not return the smallest vertex cover.

   (b) Describe an infinite family of graphs for which DUMBVERTEXCOVER returns a vertex cover of size $2 \cdot OPT$.

4. Consider the following heuristic for constructing a vertex cover of a connected graph G: return the set of non-leaf nodes in any depth-first spanning tree of G.

(a) Prove that this heuristic returns a vertex cover of G.

(b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G.

(c) Describe an infinite family of graphs for which this heuristic returns a vertex cover of size $2 \cdot OPT$.

5. Consider the following optimization version of the PARTITION problem. Given a set X of positive integers, our task is to partition X into disjoint subsets A and B such that $\max\{\sum A, \sum B\}$ is as small as possible. This problem is clearly NP-hard. Determine the approximation ratio of the following polynomial-time approximation algorithm. Prove your answer is correct.

```
PARTITION(X[1..n]):
  Sort X in increasing order
  a ← 0; b ← 0
  for i ← 1 to n
      if a < b
          a ← a + X[i]
      else
          b ← b + X[i]
  return max{a, b}
```

6. The *chromatic number* $\chi(G)$ of a graph G is the minimum number of colors required to color the vertices of the graph, so that every edge has endpoints with different colors. Computing the chromatic number exactly is NP-hard.

   Prove that the following problem is also NP-hard: Given an n-vertex graph G, return any integer between $\chi(G)$ and $\chi(G) + 573$. *[Note: This does not contradict the possibility of a constant **factor** approximation algorithm.]*

7. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem is a special case.

   (a) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G. Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$.

   (b) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G. Suppose we assign each vertex in G a *random* color from the set {red, green, blue}. Prove that the expected number of interesting edges is at least $\frac{2}{3} wow(G)$.

8. Consider the following algorithm for coloring a graph G.

```
TREECOLOR(G):
  T ← any spanning tree of G
  Color the tree T with two colors
  c ← 2

  for each edge (u, v) ∈ G \ T
      T ← T ∪ {(u, v)}

      if color(u) = color(v)    ⟨⟨Try recoloring u with an existing color⟩⟩
          for i ← 1 to c
              if no neighbor of u in T has color i
                  color(u) ← i

      if color(u) = color(v)    ⟨⟨Try recoloring v with an existing color⟩⟩
          for i ← 1 to c
              if no neighbor of v in T has color i
                  color(v) ← i

      if color(u) = color(v)    ⟨⟨Give up and create a new color⟩⟩
          c ← c + 1
          color(u) ← c
```

(a) Prove that this algorithm colors any bipartite graph with just two colors.

(b) Let $\Delta(G)$ denote the maximum degree of any vertex in G. Prove that this algorithm colors any graph G with at most $\Delta(G)$ colors. This trivially implies that TREECOLOR is a $\Delta(G)$-approximation algorithm.

(c) Prove that TREECOLOR is *not* a constant-factor approximation algorithm.

9. The KNAPSACK problem can be defined as follows. We are given a finite set of elements X where each element $x \in X$ has a non-negative *size* and a non-negative *value*, along with an integer *capacity* c. Our task is to determine the maximum total value among all subsets of X whose total size is at most c. This problem is NP-hard. Specifically, the optimization version of SUBSETSUM is a special case, where each element's value is equal to its size.

   Determine the approximation ratio of the following polynomial-time approximation algorithm. Prove your answer is correct.

```
APPROXKNAPSACK(X, c):
  return max{GREEDYKNAPSACK(X, c), PICKBESTONE(X, c)}
```

```
GREEDYKNAPSACK(X, c):
  Sort X in decreasing order by the ratio value/size
  S ← 0;  V ← 0
  for i ← 1 to n
      if S + size(x_i) > c
          return V
      S ← S + size(x_i)
      V ← V + value(x_i)
  return V
```

```
PICKBESTONE(X, c):
  Sort X in increasing order by size
  V ← 0
  for i ← 1 to n
      if size(x_i) > c
          return V
      if value(x_i) > V
          V ← value(x_i)
  return V
```

10. In the *bin packing* problem, we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. It's not hard to show that this problem is NP-Hard; this question

asks you to analyze a few common approximation algorithms. In each case, the input is an array $W[1..n]$ of weights, and the output is the number of bins used.

```
NextFit(W[1..n]):
  b ← 0
  Total[0] ← ∞
  for i ← 1 to n
      if Total[b] + W[i] > 1
          b ← b + 1
          Total[b] ← W[i]
      else
          Total[b] ← Total[b] + W[i]
  return b
```

```
FirstFit(W[1..n]):
  b ← 0
  for i ← 1 to n
      j ← 1; found ← False
      while j ≤ b and found = False
          if Total[j] + W[i] ≤ 1
              Total[j] ← Total[j] + W[i]
              found ← True
          j ← j + 1
      if found = False
          b ← b + 1
          Total[b] = W[i]
  return b
```

(a) Prove that NextFit uses at most twice the optimal number of bins.

(b) Prove that FirstFit uses at most twice the optimal number of bins.

*(c) Prove that if the weight array W is initially sorted in decreasing order, then FirstFit uses at most $(4 \cdot OPT + 1)/3$ bins, where OPT is the optimal number of bins. The following facts may be useful (but you need to prove them if your proof uses them):

- In the packing computed by FirstFit, every item with weight more than $1/3$ is placed in one of the first OPT bins.
- FirstFit places at most $OPT - 1$ items outside the first OPT bins.

11. Given a graph G with edge weights and an integer k, suppose we wish to partition the the vertices of G into k subsets $S_1, S_2, \ldots, S_k$ so that the sum of the weights of the edges that cross the partition (that is, have endpoints in different subsets) is as large as possible.

(a) Describe an efficient $(1 - 1/k)$-approximation algorithm for this problem.

(b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for the new problem? Justify your answer.

12. The lecture notes describe a $(3/2)$-approximation algorithm for the metric traveling salesman problem. Here, we consider computing minimum-cost Hamiltonian *paths*. Our input consists of a graph G whose edges have weights that satisfy the triangle inequality. Depending upon the problem, we are also given zero, one, or two endpoints.

(a) If our input includes zero endpoints, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path.

(b) If our input includes one endpoint u, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u.

(c) If our input includes two endpoints u and v, describe a $(5/3)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u and ends at v.

13. Suppose we are given a collection of n *jobs* to execute on a machine containing a row of m processors. When the ith job is executed, it occupies a *contiguous* set of $prox[i]$ processors for $time[i]$ seconds. A *schedule* for a set of jobs assigns each job an interval of processors and a starting time, so that no processor works on more than one job at any time. The *makespan* of a schedule is the time from the start to the finish of all jobs. Finally, the *parallel scheduling problem* asks us to compute the schedule with the smallest possible makespan.

    (a) Prove that the parallel scheduling problem is NP-hard.

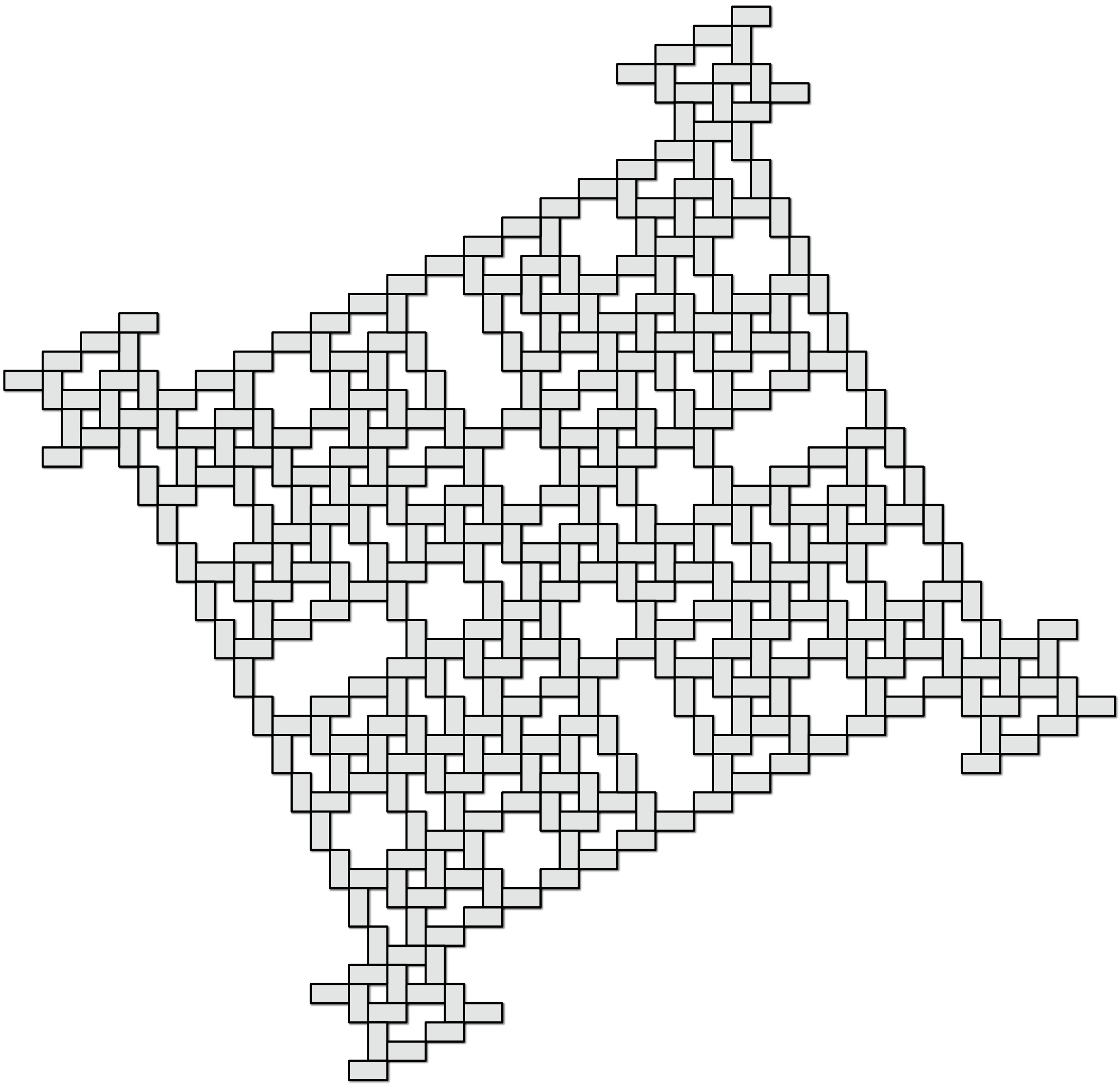
    (b) Give an algorithm that computes a 3-approximation of the minimum makespan of a set of jobs in $O(m \log m)$ time. That is, if the minimum makespan is M, your algorithm should compute a schedule with make-span at most $3M$. You can assume that n is a power of 2.

14. Consider the greedy algorithm for metric TSP: start at an arbitrary vertex u, and at each step, travel to the closest unvisited vertex.

    (a) Show that the greedy algorithm for metric TSP is an $O(\log n)$-approximation, where n is the number of vertices. *[Hint: Argue that the kth least expensive edge in the tour output by the greedy algorithm has weight at most* $\text{OPT}/(n-k+1)$*; try* $k=1$ *and* $k=2$ *first.]*

    *(b) Show that the greedy algorithm for metric TSP is no better than an $O(\log n)$-approximation. That is, describe an infinite family of weighted graphs such that the greedy algorithm returns a cycle whose weight is $\Omega(\log n)$ times the optimal TSP tour.

Appendices



Jeder Genießende meint, dem Baume habe es an der Frucht gelegen;
aber ihm lag am Samen.
[Everyone who enjoys thinks that the fundamental thing about trees is the fruit,
but in fact it is the seed.]

— Friedrich Wilhelm Nietzsche,
Vermischte Meinungen und Sprüche [Mixed Opinions and Maxims] (1879)

In view of all the deadly computer viruses that have been spreading lately,
Weekend Update would like to remind you:
When you link up to another computer,
you're linking up to every computer that that computer has ever linked up to.

— Dennis Miller, "Saturday Night Live", (c. 1985)

Anything that, in happening, causes itself to happen again, happens again.

— Douglas Adams (2005)

The Curling Stone slides; and, having slid,
Passes me toward thee on this Icy Grid,
If what's reached is passed for'll Crystals amid,
Th'Stone Reaches thee in its Eternal Skid.

— Iraj Kalantari (2007)
writing as "Harak A'Myomy (12th century),
translated by Walt Friz De Gradde (1897)"

Proof by Induction

Induction is a method for proving universally quantified propositions—statements about *all* elements of a (usually infinite) set. Induction is also the single most useful tool for reasoning about, developing, and analyzing algorithms. These notes give several examples of inductive proofs, along with a standard boilerplate and some motivation to justify (and help you remember) why induction works.

1 Prime Divisors: Proof by Smallest Counterexample

A ***divisor*** of a positive integer n is a positive integer p such that the ratio n/p is an integer. The integer 1 is a divisor of every positive integer (because $n/1 = n$), and every integer is a divisor of itself (because $n/n = 1$). A ***proper divisor*** of n is any divisor of n other than n itself. A positive integer is ***prime*** if it has *exactly two* divisors, which must be 1 and itself; equivalently; a number is prime if and only if 1 is its only proper divisor. A positive integer is ***composite*** if it has more than two divisors (or equivalently, more than one proper divisor). The integer 1 is neither prime nor composite, because it has exactly one divisor, namely itself.

Let's prove our first theorem:

Theorem 1. *Every integer greater than* 1 *has a prime divisor.*

The very first thing that you should notice, after reading just one word of the theorem, is that this theorem is *universally quantified*—it's a statement about *all* the elements of a set, namely, the set of positive integers larger than 1. If we were forced at gunpoint to write this sentence using fancy logic notation, the first character would be the universal quantifier $\forall$, pronounced 'for all'. Fortunately, that won't be necessary.

There are only two ways to prove a universally quantified statement: directly or by contradiction. Let's say that again, louder: ***There are only two ways to prove a universally quantified statement: directly or by contradiction.*** Here are the standard templates for these two methods, applied to Theorem 1:

Direct proof: Let n be an arbitrary integer greater than 1.

. . . blah blah blah . . .

Thus, n has at least one prime divisor. $\square$

Proof by contradiction: For the sake of argument, assume there is an integer greater than 1 with no prime divisor.

Let n be an arbitrary integer greater than 1 with no prime divisor.

. . . blah blah blah . . .

But that's just silly. Our assumption must be incorrect. $\square$

The shaded boxes *. . . blah blah blah . . .* indicate missing proof details (that you will fill in).

Most people usually find proofs by contradiction easier to discover than direct proofs, so let's try that first.

Proof by contradiction: For the sake of argument, assume there is an integer greater than 1 with no prime divisor.

Let n be an arbitrary integer greater than 1 with no prime divisor.
Since n is a divisor of n, and n has no prime divisors, n cannot be prime.
Thus, n must have at least one divisor d such that $1 < d < n$.
Let d be an arbitrary divisor of n such that $1 < d < n$.
Since n has no prime divisors, d cannot be prime.
Thus, d has at least one divisor d' such that $1 < d' < d$.
Let d' be an arbitrary divisor of d such that $1 < d' < d$.
Because d/d' is an integer, $n/d' = (n/d) \cdot (d/d')$ is also an integer.
Thus, d' is also a divisor of n.
Since n has no prime divisors, d' cannot be prime.
Thus, d' has at least one divisor d'' such that $1 < d'' < d'$.
Let d'' be an arbitrary divisor of d' such that $1 < d'' < d'$.
Because d'/d'' is an integer, $n/d'' = (n/d') \cdot (d'/d'')$ is also an integer.
Thus, d'' is also a divisor of n.
Since n has no prime divisors, d'' cannot be prime.

. . . blah HELP! blah I'M STUCK IN AN INFINITE LOOP! blah . . .

But that's just silly. Our assumption must be incorrect. $\square$

We seem to be stuck in an infinite loop, looking at smaller and smaller divisors $d > d' > d'' > \cdots$, none of which are prime. But this loop can't really be infinite. There are only $n-1$ positive integers smaller than n, so the proof *must* end after *at most* $n-1$ iterations. But how do we turn this observation into a formal proof? We need a single, self-contained proof for all integers n; we're not allowed to write longer proofs for bigger integers. The trick is to jump directly to the ***smallest*** counterexample.

Proof by smallest counterexample: For the sake of argument, assume that there is an integer greater than 1 with no prime divisor.
Let n be ***the smallest*** integer greater than 1 with no prime divisor.
Since n is a divisor of n, and n has no prime divisors, n cannot be prime.
Thus, n has a divisor d such that $1 < d < n$.
Let d be a divisor of n such that $1 < d < n$.
Because n is the smallest counterexample, d has a prime divisor.
Let p be a prime divisor of d.
Because d/p is an integer, $n/p = (n/d) \cdot (d/p)$ is also an integer.
Thus, p is also a divisor of n.
But this contradicts our assumption that n has no prime divisors!
So our assumption must be incorrect. □

Hooray, our first proof! We're done!

Um... well... no, we're definitely *not* done. That's a first draft up there, not a final polished proof. We don't write proofs just to convince ourselves; proofs are primarily a tool to convince other people. (In particular, 'other people' includes the people grading your homeworks and exams.) And while proofs by contradiction are usually easier to *write*, direct proofs are almost always easier to *read*. So as a service to our audience (and our grade), let's transform our minimal-counterexample proof into a direct proof.

Let's first rewrite the indirect proof slightly, to make the structure more apparent. First, we break the assumption that n is the smallest counterexample into three simpler assumptions: (1) n is an integer greater than 1; (2) n has no prime divisors; and (3) there are no smaller counterexamples. Second, instead of dismissing the possibility than n is prime out of hand, we include an explicit case analysis.

Proof by smallest counterexample: Let n be an arbitrary integer greater than 1.
For the sake of argument, suppose n has no prime divisor.
Assume that every integer k such that $1 < k < n$ has a prime divisor.
There are two cases to consider: Either n is prime, or n is composite.

- Suppose n is prime.
  Then n is a prime divisor of n.
- Suppose n is composite.
  Then n has a divisor d such that $1 < d < n$.
  Let d be a divisor of n such that $1 < d < n$.
  Because no counterexample is smaller than n, d has a prime divisor.
  Let p be a prime divisor of d.
  Because d/p is an integer, $n/p = (n/d) \cdot (d/p)$ is also an integer.
  Thus, p is a prime divisor of n.

In each case, we conclude that n has a prime divisor.
But this contradicts our assumption that n has no prime divisors!
So our assumption must be incorrect. □

Now let's look carefully at the structure of this proof. First, we assumed that the statement we want to prove is false. Second, we proved that the statement we want to prove is true. Finally, we concluded from the contradiction that our assumption that the statement we want to prove is false is incorrect, so the statement we want to prove must be true.

But that's just silly. Why do we need the first and third steps? After all, the second step is a proof all by itself! Unfortunately, this redundant style of proof by contradiction is *extremely* common, even in professional papers. Fortunately, it's also very easy to avoid; just remove the first and third steps!

> **Proof by induction:** Let n be an arbitrary integer greater than 1.
> ***Assume that every integer k such that $1 < k < n$ has a prime divisor.***
> There are two cases to consider: Either n is prime or n is composite.
> - First, suppose n is prime.
>   Then n is a prime divisor of n.
> - Now suppose n is composite.
>   Then n has a divisor d such that $1 < d < n$.
>   Let d be a divisor of n such that $1 < d < n$.
>   ***Because no counterexample is smaller than n***, d has a prime divisor.
>   Let p be a prime divisor of d.
>   Because d/p is an integer, $n/p = (n/d) \cdot (d/p)$ is also an integer.
>   Thus, p is a prime divisor of n.
>
> In both cases, we conclude that n has a prime divisor. □

This style of proof is called ***induction***.[1] The assumption that there are no counterexamples smaller than n is called the ***induction hypothesis***. The two cases of the proof have different names. The first case, which we argue directly, is called the ***base case***. The second case, which actually uses the induction hypothesis, is called the ***inductive case***. You may find it helpful to actually label the induction hypothesis, the base case(s), and the inductive case(s) in your proof.

The following point cannot be emphasized enough: The only difference between a proof by induction and a proof by smallest counterexample is the way we write down the argument. The essential structure of the proofs are exactly the same. The core of our original indirect argument is a proof of the following implication for all n:

$$n \text{ has no prime divisor} \implies \text{some number smaller than } n \text{ has no prime divisor.}$$

The core of our direct proof is the following logically equivalent implication:

$$\text{every number smaller than } n \text{ has a prime divisor} \implies n \text{ has a prime divisor}$$

The left side of this implication is just the induction hypothesis.

The proofs we've been playing with have been very careful and explicit; until you're comfortable writing your own proofs, you should be equally careful. A more mature proof-writer might express the same proof more succinctly as follows:

> **Proof by induction:** Let n be an arbitrary integer greater than 1. Assume that every integer k such that $1 < k < n$ has a prime divisor. If n is prime, then n is a prime divisor of n. On the other hand, if n is composite, then n has a proper divisor; call it d. The induction hypothesis implies that d has a prime divisor p. The integer p is also a divisor of n. □

A proof in this more succinct form is still worth full credit, provided the induction hypothesis is written explicitly and the case analysis is obviously exhaustive.

A professional mathematician would write the proof even more tersely:

> **Proof:** Induction. □

And you can write that tersely, too, when you're a professional mathematician.

---

[1]Many authors use the high-falutin' name *the principle of mathematical induction*, to distinguish it from *inductive reasoning*, the informal process by which we conclude that pigs can't whistle, horses can't fly, and NP-hard problems cannot be solved in polynomial time. We already know that every proof is mathematical (and arguably, all mathematics is proof), so as a description of a *proof* technique, the adjective 'mathematical' is simply redundant.

2 The Axiom of Induction

Why does this work? Well, let's step back to the original proof by smallest counterexample. How do we know that a smallest counterexample exists? This seems rather obvious, but in fact, it's impossible to prove without using the following seemingly trivial observation, called the ***Well-Ordering Principle***:

> *Every non-empty set of positive integers has a smallest element.*

Every set X of positive integers is the set of counterexamples to some proposition $P(n)$ (specifically, the proposition $n \notin X$). Thus, the Well-Ordering Principle can be rewritten as follows:

> *If the proposition $P(n)$ is false for some positive integer n,*
> *then*
> *the proposition $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1) \wedge \neg P(n))$ is true for some positive integer n.*

Equivalently, in English:

> *If some statement about positive integers has a counterexample,*
> *then*
> *that statement has a smallest counterexample.*

We can write this implication in contrapositive form as follows:

> *If the proposition $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1) \wedge \neg P(n))$ is false for every positive integer n,*
> *then*
> *the proposition $P(n)$ is true for every positive integer n.*

or less formally,

> *If some statement about positive integers has no smallest counterexample,*
> *then*
> *that statement is true for all positive integers.*

Finally, let's rewrite the first half of this statement in a logically equivalent form, by replacing $\neg(p \wedge \neg q)$ with $p \rightarrow q$.

> *If the implication $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1)) \rightarrow P(n)$ is true for every positive integer n,*
> *then*
> *the proposition $P(n)$ is true for every positive integer n.*

This formulation is usually called the ***Axiom of Induction***. In a proof by induction that $P(n)$ holds for all n, the conjunction $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1))$ is the inductive hypothesis.

A ***proof by induction*** for the proposition "$P(n)$ for every positive integer n" is nothing but a direct proof of the more complex proposition "$(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1)) \rightarrow P(n)$ for every positive integer n". Because it's a direct proof, it *must* start by considering an arbitrary positive integer, which we might as well call n. Then, to prove the implication, we explicitly assume the hypothesis $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1))$ and then prove the conclusion $P(n)$ *for that particular value of n*. The proof almost always breaks down into two or more cases, each of which may or may not actually use the inductive hypothesis.

Here is the boilerplate for *every* induction proof. Read it. Learn it. Use it.

Theorem: $P(n)$ for every positive integer n.

Proof by induction: Let n be an arbitrary positive integer.
Assume inductively that $P(k)$ is true for every positive integer $k < n$.
There are several cases to consider:
- Suppose n is *... blah blah blah ...*
  Then $P(n)$ is true.
- Suppose n is *... blah blah blah ...*
  The inductive hypothesis implies that *... blah blah blah ...*
  Thus, $P(n)$ is true.

In each case, we conclude that $P(n)$ is true. □

Some textbooks distinguish between several different types of induction: 'regular' induction versus 'strong' induction versus 'complete' induction versus 'structural' induction versus 'transfinite' induction versus 'Noetherian' induction. Distinguishing between these different types of induction is pointless hairsplitting; I won't even define them. Every 'different type' of induction proof is provably equivalent to a proof by smallest counterexample. (Later we will consider inductive proofs of statements about partially ordered sets other than the positive integers, for which 'smallest' has a different meaning, but this difference will prove to be inconsequential.)

3 Stamps and Recursion

Let's move on to a completely different example.

Theorem 2. *Given an unlimited supply of 5-cent stamps and 7-cent stamps, we can make any amount of postage larger than 23 cents.*

We could prove this by contradiction, using a smallest-counterexample argument, but let's aim for a direct proof by induction this time. We start by writing down the induction boilerplate, using the standard induction hypothesis: *There is no counterexample smaller than n.*

Proof by induction: Let n be an arbitrary integer greater than 23.
Assume that for any integer k such that $23 < k < n$, we can make k cents in postage.

... blah blah blah ...

Thus, we can make n cents in postage. □

How do we fill in the details? One approach is to think about what you would actually do if you really had to make n cents in postage. For example, you might start with a 5-cent stamp, and then try to make $n-5$ cents in postage. The inductive hypothesis says you can make *any* amount of postage bigger than 23 cents and less than n cents. So if $n-5 > 23$, then *you already know* that you can make $n-5$ cents in postage! (You don't know *how* to make $n-5$ cents in postage, but so what?)

Let's write this observation into our proof as two separate cases: either $n > 28$ (where our approach works) or $n \le 28$ (where we don't know what to do yet).

> **Proof by induction:** Let n be an arbitrary integer greater than 23.
> Assume that for any integer k such that $23 < k < n$, we can make k cents in postage.
> There are two cases to consider: Either $n > 28$ or $n \le 28$.
> - Suppose $n > 28$.
>   Then $23 < n - 5 < n$.
>   Thus, ***the induction hypothesis*** implies that we can make $n - 5$ cents in postage.
>   Adding one more 5-cent stamp gives us n cents in postage.
> - Now suppose $n \le 28$.
>
>   *. . . blah blah blah . . .*
>
> In both cases, we can make n cents in postage. □

What do we do in the second case? Fortunately, this case considers only five integers: 24, 25, 26, 27, and 28. There might be a clever way to solve all five cases at once, but why bother? They're small enough that we can find a solution by brute force in less than a minute. To make the proof more readable, I'll unfold the nested cases and list them in increasing order.

> **Proof by induction:** Let n be an arbitrary integer greater than 23.
> Assume that for any integer k such that $23 < k < n$, we can make k cents in postage.
> There are six cases to consider: $n = 24$, $n = 25$, $n = 26$, $n = 27$, $n = 28$, and $n > 28$.
> - $24 = 7 + 7 + 5 + 5$
> - $25 = 5 + 5 + 5 + 5 + 5$
> - $26 = 7 + 7 + 7 + 5$
> - $27 = 7 + 5 + 5 + 5 + 5$
> - $28 = 7 + 7 + 7 + 7$
> - Suppose $n > 28$.
>   Then $23 < n - 5 < n$.
>   Thus, ***the induction hypothesis*** implies that we can make $n - 5$ cents in postage.
>   Adding one more 5-cent stamp gives us n cents in postage.
>
> In all cases, we can make n cents in postage. □

Voilà! An induction proof! More importantly, we now have a recipe for *discovering* induction proofs.

1. ***Write down the boilerplate.*** Write down the universal invocation ('Let n be an arbitrary. . . '), the induction hypothesis, and the conclusion, with enough blank space for the remaining details. Don't be clever. Don't even think. Just write. ***This is the easy part.*** To emphasize the common structure, the boilerplate will be indicated in green for the rest of this handout.

2. ***Think big.*** *Don't* think how to solve the problem all the way down to the ground; you'll only make yourself dizzy. *Don't* think about piddly little numbers like 1 or 5 or 10^{100}. Instead, think about how to reduce the proof about some *absfoluckingutely ginormous* value of n to a proof about some other number(s) smaller than n. ***This is the hard part.***

3. ***Look for holes.*** Look for cases where your inductive argument breaks down. Solve those cases directly. Don't be clever here; be stupid but thorough.

4. ***Rewrite everything.*** Your first proof is a rough draft. Rewrite the proof so that your argument is easier for your (unknown?) reader to follow.

The cases in an inductive proof always fall into two categories. Any case that uses the inductive hypothesis is called an *inductive case*. Any case that does not use the inductive hypothesis is called a *base case*. Typically, but *not* always, base cases consider a few small values of n, and the inductive cases consider everything else. Induction proofs are usually clearer if we present the base cases first, but I find it much easier to *discover* the inductive cases first. In other words, I recommend writing induction proofs backwards.

Well-written induction proofs *very* closely resemble well-written recursive programs. We computer scientists use induction primarily to reason about recursion, so maintaining this resemblance is extremely useful—we only have to keep one mental pattern, called 'induction' when we're writing proofs and 'recursion' when we're writing code. Consider the following C and Scheme programs for making n cents in postage:

```
void postage(int n)
{
    assert(n>23);
    switch ($n$)
    {
        case 24:  printf("7+7+5+5");    break;
        case 25:  printf("5+5+5+5+5");  break;
        case 26:  printf("7+7+7+5");    break;
        case 27:  printf("7+5+5+5+5");  break;
        case 28:  printf("7+7+7+7");    break;
        default:
            postage(n-5);
            printf("+5");
    }
}
```

```
(define (postage n)
  (cond ((= n 24) (5 5 7 7))
        ((= n 25) (5 5 5 5 5))
        ((= n 26) (5 7 7 7))
        ((= n 27) (5 5 5 5 7))
        ((= n 28) (7 7 7 7))
        ((> n 28) (cons 5 (postage (- n 5))))))
```

The C program begins by declaring the input parameter ("Let n be an arbitrary integer. . . ") and asserting its range (". . . greater than 23."). (Scheme programs don't have type declarations.) In both languages, the code branches into six cases: five that are solved directly, plus one that is handled ~~by invoking the inductive hypothesis~~ recursively.

4 More on Prime Divisors

Before we move on to different examples, let's prove another fact about prime numbers:

Theorem 3. *Every positive integer is a product of prime numbers.*

First, let's write down the boilerplate. Hey! I saw that! You were *thinking*, weren't you? Stop that this instant! Don't make me turn the car around. ***First*** we write down the boilerplate.

> **Proof by induction:** Let n be an arbitrary positive integer.
>
> Assume that any positive integer $k < n$ is a product of prime numbers.
>
> There are *some* cases to consider:
>
> *... blah blah blah ...*
>
> Thus, n is a product of prime numbers. □

Now let's think about how you would actually factor a positive integer n into primes. There are a couple of different options here. One possibility is to find a prime divisor p of n, as guaranteed by Theorem 1, and recursively factor the integer n/p. This argument works as long as $n \geq 2$, but what about $n = 1$? The answer is simple: 1 *is the product of the* ***empty*** *set of primes*. What else could it be?

> **Proof by induction:** Let n be an arbitrary positive integer.
>
> Assume that any positive integer $k < n$ is a product of prime numbers.
>
> There are two cases to consider: either $n = 1$ or $n \geq 2$.
>
> - If $n = 1$, then n is the product of the elements of the empty set, each of which is prime, green, sparkly, vanilla, and hemophagic.
>
> - Suppose $n > 1$. Let p be a prime divisor of n, as guaranteed by Theorem 2. The inductive hypothesis implies that the positive integer n/p is a product of primes, and clearly $n = (n/p) \cdot p$.
>
> In both cases, n is a product of prime numbers. □

But an even simpler method is to factor n into any two proper divisors, and recursively handle them both. This method works as long as n is composite, since otherwise there is no way to factor n into smaller integers. Thus, we need to consider prime numbers separately, as well as the special case 1.

> **Proof by induction:** Let n be an arbitrary positive integer.
>
> Assume that any positive integer $k < n$ is a product of prime numbers.
>
> There are three cases to consider: either $n = 1$, n is prime, or n is composite.
>
> - If $n = 1$, then n is the product of the elements of the empty set, each of which is prime, red, broody, chocolate, and lycanthropic.
>
> - If n is prime, then n is the product of one prime number, namely n.
>
> - Suppose n is composite. Let d be any proper divisor of n (guaranteed by the definition of 'composite'), and let $m = n/d$. Since both d and m are positive integers smaller than n, the inductive hypothesis implies that d and m are both products of prime numbers. We clearly have $n = d \cdot m$.
>
> In both cases, n is a product of prime numbers. □

5 Summations

Here's an easy one.

Theorem 4. $\displaystyle\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$ *for every non-negative integer n.*

First let's write down the induction boilerplate, which empty space for the details we'll fill in later.

> **Proof by induction:** Let n be an arbitrary non-negative integer.
>
> Assume inductively that $\displaystyle\sum_{i=0}^{k} 3^i = \frac{3^{k+1}-1}{2}$ for every non-negative integer $k < n$.
>
> There are *some number of* cases to consider:
>
> *. . . blah blah blah . . .*
>
> We conclude that $\displaystyle\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$. □

Now imagine you are part of an infinitely long assembly line of mathematical provers, each assigned to a particular non-negative integer. Your task is to prove this theorem for the integer 8675310. The regulations of the Mathematical Provers Union require you not to think about any other integer but your own. The assembly line starts with the Senior Master Prover, who proves the theorem for the case $n = 0$. Next is the Assistant Senior Master Prover, who proves the theorem for $n = 1$. After him is the Assistant Assistant Senior Master Prover, who proves the theorem for $n = 2$. Then the Assistant Assistant Assistant Senior Master Prover proves the theorem for $n = 3$. As the work proceeds, you start to get more and more bored. You attempt strike up a conversation with Jenny, the prover to your left, but she ignores you, preferring to focus on the proof. Eventually, you fall into a deep, dreamless sleep. An undetermined time later, Jenny wakes you up by shouting, "Hey, doofus! It's your turn!" As you look around, bleary-eyed, you realize that Jenny and everyone to your left has finished their proofs, and that everyone is waiting for you to finish yours. What do you do?

What you do, after wiping the drool off your chin, is stop and think for a moment about what you're trying to prove. What does that $\sum$ notation actually mean? Intuitively, we can expand the notation as follows:

$$\sum_{i=0}^{8675310} 3^i = 3^0 + 3^1 + \cdots + 3^{8675309} + 3^{8675310}.$$

Notice that this expression also contains the summation that Jenny just finished proving something about:

$$\sum_{i=0}^{8675309} 3^i = 3^0 + 3^1 + \cdots + 3^{8675308} + 3^{8675309}.$$

Putting these two expressions together gives us the following identity:

$$\sum_{i=0}^{8675310} 3^i = \sum_{i=0}^{8675309} 3^i \; + \; 3^{8675310}$$

In fact, this recursive identity is the *definition* of $\sum$. Jenny just proved that the summation on the right is equal to $(3^{8675310} - 1)/2$, so we can plug that into the right side of our equation:

$$\sum_{i=0}^{8675310} 3^i = \sum_{i=0}^{8675309} 3^i \; + \; 3^{8675310} = \frac{3^{8675310}-1}{2} + 3^{8675310}.$$

And it's all downhill from here. After a little bit of algebra, you simplify the right side of this equation to $(3^{8675311} - 1)/2$, wake up the prover to your right, and start planning your well-earned vacation.

Let's insert this argument into our boilerplate, only using a generic 'big' integer n instead of the specific integer 8675310:

> **Proof by induction:** Let n be an arbitrary non-negative integer.
>
> Assume inductively that $\sum_{i=0}^{k} 3^i = \frac{3^{k+1}-1}{2}$ for every non-negative integer $k < n$.
>
> There are two cases to consider: Either n *is big* or n *is small*.
>
> - If n *is big*, then
>
> $$\begin{aligned} \sum_{i=0}^{n} 3^i &= \sum_{i=0}^{n-1} 3^i + 3^n && \text{[definition of } \textstyle\sum\text{]} \\ &= \frac{3^n-1}{2} + 3^n && \text{[induction hypothesis, with } k = n-1\text{]} \\ &= \frac{3^{n+1}-1}{2} && \text{[algebra]} \end{aligned}$$
>
> - On the other hand, if n *is small*, then *. . . blah blah blah . . .*
>
> In both cases, we conclude that $\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$. □

Now, how big is 'big', and what do we do when n is 'small'? To answer the first question, let's look at where our existing inductive argument breaks down. In order to apply the induction hypothesis when $k = n-1$, the integer $n-1$ must be non-negative; equivalently, n must be *positive*. But that's the only assumption we need: ***The only case we missed is $n = 0$.*** Fortunately, this case is easy to handle directly.

> **Proof by induction:** Let n be an arbitrary non-negative integer.
>
> Assume inductively that $\sum_{i=0}^{k} 3^i = \frac{3^{k+1}-1}{2}$ for every non-negative integer $k < n$.
>
> There are two cases to consider: Either $n = 0$ or $n \geq 1$.
>
> - If $n = 0$, then $\sum_{i=0}^{n} 3^i = 3^0 = 1$, and $\frac{3^{n+1}-1}{2} = \frac{3^1-1}{2} = 1$.
> - On the other hand, if $n \geq 1$, then
>
> $$\begin{aligned} \sum_{i=0}^{n} 3^i &= \sum_{i=0}^{n-1} 3^i + 3^n && \text{[definition of } \textstyle\sum\text{]} \\ &= \frac{3^n-1}{2} + 3^n && \text{[induction hypothesis, with } k = n-1\text{]} \\ &= \frac{3^{n+1}-1}{2} && \text{[algebra]} \end{aligned}$$
>
> In both cases, we conclude that $\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$. □

Here is the same proof, written more tersely; the non-standard symbol $\stackrel{IH}{=\!=}$ indicates the use of the induction hypothesis.

Proof by induction: Let n be an arbitrary non-negative integer, and assume inductively that $\sum_{i=0}^{k} 3^i = (3^{k+1}-1)/2$ for every non-negative integer $k < n$. The base case $n = 0$ is trivial, and for any $n \geq 1$, we have

$$\sum_{i=0}^{n} 3^i = \sum_{i=0}^{n-1} 3^i + 3^n \overset{IH}{=\!=} \frac{3^n - 1}{2} + 3^n = \frac{3^{n+1}-1}{2}.$$

□

This is not the only way to prove this theorem by induction; here is another:

Proof by induction: Let n be an arbitrary non-negative integer, and assume inductively that $\sum_{i=0}^{k} 3^i = (3^{k+1}-1)/2$ for every non-negative integer $k < n$. The base case $n = 0$ is trivial, and for any $n \geq 1$, we have

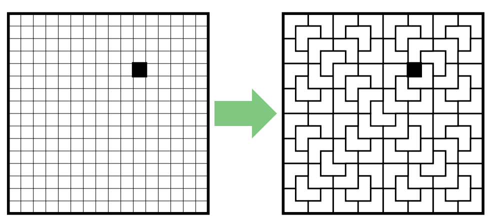
$$\sum_{i=0}^{n} 3^i = 3^0 + \sum_{i=1}^{n} 3^i = 3^0 + 3 \cdot \sum_{i=0}^{n-1} 3^i \overset{IH}{=\!=} 3^0 + 3 \cdot \frac{3^n - 1}{2} = \frac{3^{n+1}-1}{2}.$$

□

In the remainder of these notes, I'll give several more examples of induction proofs. In some cases, I give multiple proofs for the same theorem. Unlike the earlier examples, I will not describe the thought process that lead to the proof; in each case, I followed the basic outline on page 7.

6 Tiling with Triominos

The next theorem is about *tiling* a square checkerboard with *triominos*. A triomino is a shape composed of three squares meeting in an L-shape. Our goal is to cover as much of a $2^n \times 2^n$ grid with triominos as possible, without any two triominos overlapping, and with all triominos inside the square. We can't cover every square in the grid—the number of squares is 4^n, which is not a multiple of 3—but we can cover all but one square. In fact, as the next theorem shows, we can choose *any* square to be the one we don't want to cover.



Almost tiling a 16×16 checkerboard with triominos.

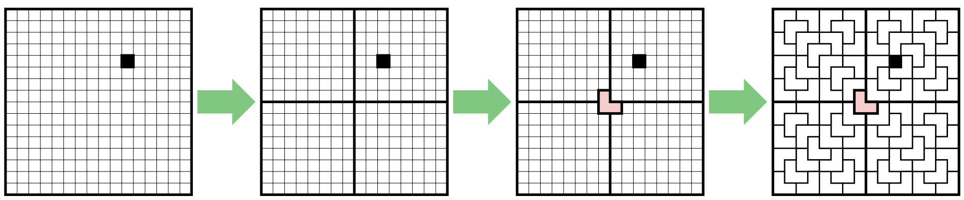
Theorem 5. *For any non-negative integer n, the $2^n \times 2^n$ checkerboard with any square removed can be tiled using L-shaped triominos.*

Here are two inductive proofs for this theorem, one 'top down', the other 'bottom up'.

Proof by top-down induction: Let n be an arbitrary non-negative integer. Assume that for any non-negative integer $k < n$, the $2^k \times 2^k$ grid with any square removed can be tiled using triominos. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The $2^0 \times 2^0$ grid has a single square, so removing one square leaves nothing, which we can tile with zero triominos.
- Suppose $n \geq 1$. In this case, the $2^n \times 2^n$ grid can be divided into four smaller $2^{n-1} \times 2^{n-1}$ grids. Without loss of generality, suppose the deleted square is in the upper right quarter. With a single L-shaped triomino at the center of the board, we can cover one square in each of the other three quadrants. The induction hypothesis implies that we can tile each of the quadrants, minus one square.

In both cases, we conclude that the $2^n \times 2^n$ grid with any square removed can be tiled with triominos. □

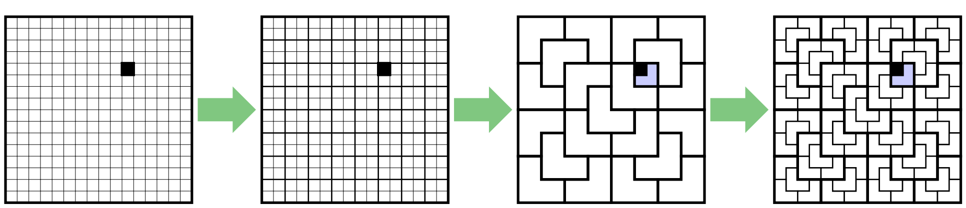


Top-down inductive proof of Theorem 4.

Proof by bottom-up induction: Let n be an arbitrary non-negative integer. Assume that for any non-negative integer $k < n$, the $2^k \times 2^k$ grid with any square removed can be tiled using triominos. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The $2^0 \times 2^0$ grid has a single square, so removing one square leaves nothing, which we can tile with zero triominos.
- Suppose $n \geq 1$. Then by clustering the squares into 2×2 *blocks*, we can transform any $2^n \times 2^n$ grid into a $2^{n-1} \times 2^{n-1}$ grid. Suppose square (i, j) has been removed from the $2^n \times 2^n$ grid. The induction hypothesis implies that the $2^{n-1} \times 2^{n-1}$ grid with block $(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$ removed can be tiled with double-size triominos. Each double-size triomono can be tiled with four smaller triominos, and block $(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$ with square (i, j) removed is another triomino.

In both cases, we conclude that the $2^n \times 2^n$ grid with any square removed can be tiled with triominos. □



Second proof of Theorem 4.

7 Binary Numbers Exist

Theorem 6. *Every non-negative integer can be written as the sum of distinct powers of 2.*

Intuitively, this theorem states that every number can be represented in binary. (That's not a proof, by the way; it's just a restatement of the theorem.) I'll present *four* distinct inductive proofs for this theorem. The first two are standard, by-the-book induction proofs.

Proof by top-down induction: Let n be an arbitrary non-negative integer. Assume that any non-negative integer less than n can be written as the sum of distinct powers of 2. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The base case $n = 0$ is trivial—the elements of the empty set are distinct and sum to zero.
- Suppose $n \geq 1$. Let k be the largest integer such that $2^k \leq n$, and let $m = n - 2^k$. Observe that $m < 2^{k+1} - 2^k = 2^k$. Because $0 \leq m < n$, the inductive hypothesis implies that m can be written as the sum of distinct powers of 2. Moreover, in the summation for m, each power of 2 is at most m, and therefore less than 2^k. Thus, $m + 2^k$ is the sum of distinct powers of 2.

In either case, we conclude that n can be written as the sum of distinct powers of 2. □

Proof by bottom-up induction: Let n be an arbitrary non-negative integer. Assume that any non-negative integer less than n can be written as the sum of distinct powers of 2. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The base case $n = 0$ is trivial—the elements of the empty set are distinct and sum to zero.
- Suppose $n \geq 1$, and let $m = \lfloor n/2 \rfloor$. Because $0 \leq m < n$, the inductive hypothesis implies that m can be written as the sum of distinct powers of 2. Thus, $2m$ can also be written as the sum of distinct powers of 2, each of which is greater than 2^0. If n is even, then $n = 2m$ and we are done; otherwise, $n = 2m + 2^0$ is the the sum of distinct powers of 2.

In either case, we conclude that n can be written as the sum of distinct powers of 2. □

The third proof deviates slightly from the induction boilerplate. At the top level, this proof doesn't actually use induction at all! However, a key step requires its own (straightforward) inductive proof.

Proof by algorithm: Let n be an arbitrary non-negative integer. Let S be a multiset containing n copies of 2^0. Modify S by running the following algorithm:

> while S has more than one copy of *any* element 2^i
>   Remove two copies of 2^i from S
>   Insert one copy of 2^{i+1} into S

Each iteration of this algorithm reduces the cardinality of S by 1, so the algorithm must eventually halt. When the algorithm halts, the elements of S are distinct. We claim that just after each iteration of the while loop, the elements of S sum to n.

> **Proof by induction:** Consider an arbitrary iteration of the loop. Assume inductively that just after each previous iteration, the elements of S sum to n. Before any iterations of the loop, the elements of S sum to n by definition. The induction hypothesis implies that just before the current iteration begins, the elements of S sum to n. The loop replaces two copies of some number 2^i with their sum 2^{i+1}, leaving the total sum of S unchanged. Thus, when the iteration ends, the elements of S sum to n. □

Thus, when the algorithm halts, the elements of S are distinct powers of 2 that sum to n. We conclude that n can be written as the sum of distinct powers of 2. □

The fourth proof uses so-called 'weak' induction, where the inductive hypothesis can only be applied at $n-1$. Not surprisingly, tying all but one hand behind our backs makes the resulting proof longer, more complicated, and harder to read. It doesn't help that the algorithm used in the proof is overly specific. Nevertheless, this is the first approach that occurs to most students who have not truly accepted the Recursion Fairy into their hearts.

Proof by baby-step induction: Let n be an arbitrary non-negative integer. Assume that any non-negative integer less than n can be written as the sum of distinct powers of 2. There are two cases to consider: Either $n = 0$ or $n \geq 1$.

- The base case $n = 0$ is trivial—the elements of the empty set are distinct and sum to zero.

- Suppose $n \geq 1$. The inductive hypothesis implies that $n - 1$ can be written as the sum of distinct powers of 2. Thus, n can be written as the sum of powers of 2, which are distinct except possibly for two copies of 2^0. Let S be this multiset of powers of 2.

  Now consider the following algorithm:

  ```
  i ← 0
  while S has more than one copy of 2^i
      Remove two copies of 2^i from S
      Insert one copy of 2^(i+1) into S
      i ← i + 1
  ```

  Each iteration of this algorithm reduces the cardinality of S by 1, so the algorithm must eventually halt. We claim that for every non-negative integer i, the following invariants are satisfied after the ith iteration of the while loop (or before the algorithm starts if $i = 0$):

  – The elements of S sum to n.

  Proof by induction: Let i be an arbitrary non-negative integer. Assume that for any non-negative integer $j \leq i$, after the jth iteration of the while loop, the elements of S sum to n. If $i = 0$, the elements of S sum to n by definition of S. Otherwise, the induction hypothesis implies that just *before* the ith iteration, the elements of S sum to n; the ith iteration replaces two copies of 2^i with 2^{i+1}, leaving the sum unchanged. □

  – The elements in S are distinct, except possibly for two copies of 2^i.

  Proof by induction: Let i be an arbitrary non-negative integer. Assume that for any non-negative integer $j \leq i$, after the jth iteration of the while loop, the elements of S are distinct except possibly for two copies of 2^j. If $i = 0$, the invariant holds by definition of S. So suppose $i > 0$. The induction hypothesis implies that just *before* the ith iteration, the elements of S are distinct except possibly for two copies of 2^i. If there are two copies of 2^i, the algorithm replaces them both with 2^{i+1}, and the invariant is established; otherwise, the algorithm halts, and the invariant is again established. □

  The second invariant implies that when the algorithm halts, the elements of S are distinct.

In either case, we conclude that n can be written as the sum of distinct powers of 2. □

Repeat after me: "Doctor! Doctor! It hurts when I do this!"

8 Irrational Numbers Exist

Theorem 7. *$\sqrt{2}$ is irrational.*

Proof: I will prove that $p^2 \neq 2q^2$ (and thus $p/q \neq \sqrt{2}$) for all positive integers p and q.

Let p and q be arbitrary positive integers. Assume that for any positive integers $i < p$ and $j < q$, we have $i^2 \neq 2j^2$. Let $i = \lfloor p/2 \rfloor$ and $j = \lfloor q/2 \rfloor$. There are three cases to consider:

- Suppose p is odd. Then $p^2 = (2i+1)^2 = 4i^2 + 4i + 1$ is odd, but $2q^2$ is even.
- Suppose p is even and q is odd. Then $p^2 = 4i^2$ is divisible by 4, but $2q^2 = 2(2j+1)^2 = 4(2j^2+2j)+2$ is not divisible by 4.
- Finally, suppose p and q are both even. The induction hypothesis implies that $i^2 \neq 2j^2$. Thus, $p^2 = 4i^2 \neq 8j^2 = 2q^2$.

In every case, we conclude that $p^2 \neq 2q^2$. □

This proof is usually presented as a proof by ***infinite descent***, which is just another form of proof by smallest counterexample. Notice that the induction hypothesis assumed that *both* p and q were as small as possible. Notice also that the 'base cases' included every pair of integers p and q where at least one of the integers is odd.

9 Fibonacci Parity

The ***Fibonacci numbers*** $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$ are recursively defined as follows:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2 \end{cases}$$

Theorem 8. *For all non-negative integers n, F_n is even if and only if n is divisible by 3.*

Proof: Let n be an arbitrary non-negative integer. Assume that for all non-negative integers $k < n$, F_k is even if and only if n is divisible by 3. There are three cases to consider: $n = 0$, $n = 1$, and $n \geq 2$.

- If $n = 0$, then n is divisible by 3, and $F_n = 0$ is even.
- If $n = 1$, then n is not divisible by 3, and $F_n = 1$ is odd.
- If $n \geq 2$, there are two subcases to consider: Either n is divisible by 3, or it isn't.
  - Suppose n is divisible by 3. Then neither $n-1$ nor $n-2$ is divisible by 3. Thus, the inductive hypothesis implies that both F_{n-1} and F_{n-2} are odd. So F_n is the sum of two odd numbers, and is therefore even.
  - Suppose n is not divisible by 3. Then exactly one of the numbers $n-1$ and $n-2$ is divisible by 3. Thus, the inductive hypothesis implies that exactly one of the numbers F_{n-1} and F_{n-2} is even, and the other is odd. So F_n is the sum of an even number and an odd number, and is therefore odd.

In all cases, F_n is even if and only if n is divisible by 3. □

10 Recursive Functions

Theorem 9. *Suppose the function $F : \mathbb{N} \to \mathbb{N}$ is defined recursively by setting $F(0) = 0$ and $F(n) = 1 + F(\lfloor n/2 \rfloor)$ for every positive integer n. Then for every positive integer n, we have $F(n) = 1 + \lfloor \log_2 n \rfloor$.*

Proof: Let n be an arbitrary positive integer. Assume that $F(k) = 1 + \lfloor \log_2 k \rfloor$ for every positive integer $k < n$. There are two cases to consider: Either $n = 1$ or $n \geq 2$.

- Suppose $n = 1$. Then $F(n) = F(1) = 1 + F(\lfloor 1/2 \rfloor) = 1 + F(0) = 1$ and $1 + \lfloor \log_2 n \rfloor = 1 + \lfloor \log_2 1 \rfloor = 1 + \lfloor 0 \rfloor = 1$.
- Suppose $n \geq 2$. Because $1 \leq \lfloor n/2 \rfloor < n$, the induction hypothesis implies that $F(\lfloor n/2 \rfloor) = 1 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor$. The definition of $F(n)$ now implies that $F(n) = 1 + F(\lfloor n/2 \rfloor) = 2 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor$.

  Now there are two subcases to consider: n is either even or odd.

  - If n is even, then $\lfloor n/2 \rfloor = n/2$, which implies

$$\begin{aligned} F(n) &= 2 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor \\ &= 2 + \lfloor \log_2 (n/2) \rfloor \\ &= 2 + \lfloor (\log_2 n) - 1 \rfloor \\ &= 2 + \lfloor \log_2 n \rfloor - 1 \\ &= 1 + \lfloor \log_2 n \rfloor. \end{aligned}$$

  - If n is odd, then $\lfloor n/2 \rfloor = (n-1)/2$, which implies

$$\begin{aligned} F(n) &= 2 + \lfloor \log_2 \lfloor n/2 \rfloor \rfloor \\ &= 2 + \lfloor \log_2 ((n-1)/2) \rfloor \\ &= 1 + \lfloor \log_2 (n-1) \rfloor \\ &= 1 + \lfloor \log_2 n \rfloor \end{aligned}$$

    by the algebra in the even case. Because $n > 1$ and n is odd, n cannot be a power of 2; thus, $\lfloor \log_2 n \rfloor = \lfloor \log_2 (n-1) \rfloor$.

In all cases, we conclude that $F(n) = 1 + \lfloor \log_2 n \rfloor$. □

11 Trees

Recall that a *tree* is a connected undirected graph with no cycles. A *subtree* of a tree T is a connected subgraph of T; a *proper* subtree is any tree except T itself.

Theorem 10. *In every tree, the number of vertices is one more than the number of edges.*

This one is actually pretty easy to prove directly from the definition of 'tree': a connected acyclic graph.

Proof: Let T be an arbitrary tree. Choose an arbitrary vertex v of T to be the root, and direct every edge of T outward from v. Because T is connected, every node except v has *at least* one edge directed into it. Because T is acyclic, every node has *at most* one edge directed into it, and no edge is directed into v. Thus, for every node $x \neq v$, there is *exactly* one edge directed into x. We conclude that the number of edges is one less than the number of nodes. □

But we can prove this theorem by induction as well, in several different ways. Each inductive proof is structured around a different recursive definition of 'tree'. First, a tree is either a single node, or two trees joined by an edge.

> **Proof:** Let T be an arbitrary tree. Assume that in any proper subtree of T, the number of vertices is one more than the number of edges. There are two cases to consider: Either T has one vertex, or T has more than one vertex.
>
> - If T has one vertex, then it has no edges.
> - Suppose T has more than one vertex. Because T is connected, every pair of vertices is joined by a path. Thus, T must contain at least one edge. Let e be an arbitrary edge of T, and consider the graph $T \setminus e$ obtained by deleting e from T.
>
>   Because T is acyclic, there is no path in $T \setminus e$ between the endpoints of e. Thus, T has at least two connected components. On the other hand, because T is connected, $T \setminus e$ has at most two connected components. Thus, $T \setminus e$ has exactly two connected components; call them A and B.
>
>   Because T is acyclic, subgraphs A and B are also acyclic. Thus, A and B are subtrees of T, and therefore the induction hypothesis implies that $|E(A)| = |V(A)| - 1$ and $|E(B)| = |V(B)| - 1$.
>
>   Because A and B do not share any vertices or edges, we have $|V(T)| = |V(A)| + |V(B)|$ and $|E(T)| = |E(A)| + |E(B)| + 1$.
>
>   Simple algebra now implies that $|E(T)| = |V(T)| - 1$.
>
> In both cases, we conclude that the number of vertices in T is one more than the number of edges in T. □

Second, a tree is a single node connected by edges to a finite set of trees.

> **Proof:** Let T be an arbitrary tree. Assume that in any proper subtree of T, the number of vertices is one more than the number of edges. There are two cases to consider: Either T has one vertex, or T has more than one vertex.
>
> - If T has one vertex, then it has no edges.
> - Suppose T has more than one vertex. Let v be an arbitrary vertex of T, and let d be the degree of v. Delete v and all its incident edges from T to obtain a new graph G. This graph has exactly d connected components; call them $G_1, G_2, \ldots, G_d$. Because T is acyclic, every subgraph of T is acyclic. Thus, every subgraph G_i is a proper subtree of G. So the induction hypothesis implies that $|E(G_i)| = |V(G_i)| - 1$ for each i. We conclude that
>
> $$|E(T)| = d + \sum_{i=1}^{d} |E(G_i)| = d + \sum_{i=1}^{d} (|V(G_i)| - 1) = \sum_{i=1}^{d} |V(G_i)| = |V(T)| - 1.$$
>
> In both cases, we conclude that the number of vertices in T is one more than the number of edges in T. □

But you should ***never*** attempt to argue like this:

> **Not a Proof:** The theorem is clearly true for the 1-node tree. So let T be an arbitrary tree with at least two nodes. Assume inductively that the number of vertices in T is one more than the number of edges in T. Suppose we add one more leaf to T to get a new tree T'. This new tree has one more vertex than T and one more edge than T. Thus, the number of vertices in T' is one more than the number of edges in T'. □

This is not a proof. Every sentence is true, and the connecting logic is correct, but it does not imply the theorem, because it doesn't *explicitly* consider *all possible* trees. Why should the reader believe that their favorite tree can be recursively constructed by adding leaves to a 1-node tree? It's *true*, of course, but that argument doesn't *prove* it. Remember: ***There are only two ways to prove any universally quantified statement:*** Directly ("Let T be an arbitrary tree...") or by contradiction ("Suppose some tree T doesn't...").

Here is a *correct* inductive proof using the same underlying idea. In this proof, I don't have to prove that the proof considers arbitrary trees; it says so right there on the first line! As usual, the proof very strongly resembles a recursive algorithm, including a subroutine to find a leaf.

> **Proof:** Let T be an arbitrary tree. Assume that in any proper subtree of T, the number of vertices is one more than the number of edges. There are two cases to consider: Either T has one vertex, or T has more than one vertex.
>
> - If T has one vertex, then it has no edges.
> - Otherwise, T must have at least one vertex of degree 1, otherwise known as a leaf.
>
>   > **Proof:** Consider a walk through the graph T that starts at an arbitrary vertex and continues as long as possible without repeating any edge. The walk can never visit the same vertex more than once, because T is acyclic. Whenever the walk visits a vertex of degree at least 2, it can continue further, because that vertex has at least one unvisited edge. But the walk must eventually end, because T is finite. Thus, the walk must eventually reach a vertex of degree 1. □
>
>   Let ℓ be an arbitrary leaf of T, and let T' be the tree obtained by deleting ℓ from T. Then we have the identity
>
> $$|E(T)| = |E(T')| + 1 = |V(T')| = |V(T)| - 1,$$
>
>   where the first and third equalities follow from the definition of T', and the second equality follows from the inductive hypothesis.
>
> In both cases, we conclude that the number of vertices in T is one more than the number of edges in T. □

Exercises

1. Prove that given an unlimited supply of 6-cent coins, 10-cent coins, and 15-cent coins, one can make any amount of change larger than 29 cents.

2. Prove that $\displaystyle\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$ for every non-negative integer n and every real number $r \neq 1$.

3. Prove that $\displaystyle\left(\sum_{i=0}^{n} i\right)^2 = \sum_{i=0}^{n} i^3$ for every non-negative integer n.

4. Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and

$F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all non-negative integers n and m.

(a) $\sum_{i=0}^{n} F_i = F_{n+2} - 1$

(b) $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$

*(c) If n is an integer multiple of m, then F_n is an integer multiple of F_m.

5. Prove that every integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example:

$$42 = 3^4 - 3^3 - 3^2 - 3^1$$
$$25 = 3^3 - 3^1 + 3^0$$
$$17 = 3^3 - 3^2 - 3^0$$

6. Prove that every integer (positive, negative, or zero) can be written in the form $\sum_i (-2)^i$, where the exponents i are distinct non-negative integers. For example:

$$42 = (-2)^6 + (-2)^5 + (-2)^4 + (-2)^0$$
$$25 = (-2)^6 + (-2)^5 + (-2)^3 + (-2)^0$$
$$17 = (-2)^4 + (-2)^0$$

7. (a) Prove that every non-negative integer can be written as the sum of distinct, non-consecutive Fibonacci numbers. That is, if the Fibonacci number F_i appears in the sum, it appears exactly once, and its neighbors F_{i-1} and F_{i+1} do not appear at all. For example:

$$17 = F_7 + F_4 + F_2$$
$$42 = F_9 + F_6$$
$$54 = F_9 + F_7 + F_5 + F_3$$

(b) Prove that every positive integer can be written as the sum of distinct Fibonacci numbers *with no consecutive gaps*. That is, for any index $i \geq 1$, if the consecutive Fibonacci numbers F_i or F_{i+1} do not appear in the sum, then no larger Fibonacci number F_j with $j > i$ appears in the sum. In particular, the sum *must* include either F_1 or F_2. For example:

$$16 = F_6 + F_5 + F_3 + F_2$$
$$42 = F_8 + F_7 + F_5 + F_3 + F_1$$
$$54 = F_8 + F_7 + F_6 + F_5 + F_4 + F_3 + F_2 + F_1$$

(c) The Fibonacci sequence can be extended backward to negative indices by rearranging the defining recurrence: $F_n = F_{n+2} - F_{n+1}$. Here are the first several negative-index Fibonacci numbers:

n	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
F_n	-55	34	-21	13	-8	5	-3	2	-1	1

Prove that $F_{-n} = (-1)^{n+1} F_n$.

*(d) Prove that *every* integer—positive, negative, or zero—can be written as the sum of distinct, non-consecutive Fibonacci numbers *with negative indices*. For example:

$$\begin{aligned} 17 &= F_{-7} + F_{-5} + F_{-2} \\ -42 &= F_{-10} + F_{-7} \\ 54 &= F_{-9} + F_{-7} + F_{-5} + F_{-3} + F_{-1}. \end{aligned}$$

8. Consider the following game played with a finite number of a identical coins, which are arranged into *stacks*. Each coin belongs to exactly one stack. Let n_i denote the number of coins in stack i. In each turn, you must make one of the following moves:

   - For some i and j such that $n_j \le n_i - 2$, move one coin from stack i to stack j.
   - Move one coin from any stack into a new stack.
   - Find a stack containing only one coin, and remove that coin from the game.

   The game ends when all coins are gone. For example, the following sequence of turns describes a complete game; each vector lists the number of coins in each non-empty stack:

$$\begin{aligned} \langle 4,2,1\rangle &\implies \langle 4,1,1,1\rangle \implies \langle 3,2,1,1\rangle \implies \langle 2,2,2,1\rangle \implies \langle 2,2,1,1,1\rangle \\ &\implies \langle 2,1,1,1,1,1\rangle \implies \langle 2,1,1,1,1\rangle \implies \langle 2,1,1,1\rangle \implies \langle 2,1,1\rangle \\ &\implies \langle 2,1\rangle \implies \langle 2\rangle \implies \langle 1,1\rangle \implies \langle 1\rangle \implies \langle\,\rangle \end{aligned}$$

   (a) Prove that this game ends after a finite number of turns.

   (b) What are the minimum and maximum number of turns in a game, if we start with a single stack of n coins? Prove your answers are correct.

   (c) Now suppose each time you remove a coin from a stack, you must place *two* coins onto smaller stacks. In each turn, you must make one of the following moves:

   - For some indices i, j, and k such that $n_j \le n_i - 2$ and $n_k \le n_i - 2$ and $j \ne k$, remove a coin from stack i, add a coin to stack j, and add a coin to stack k.
   - For some i and j such that $n_j \le n_i - 2$, remove a coin from stack i, add a coin to stack j, and create a new stack with one coin.
   - Remove one coin from any stack and create two new stacks, each with one coin.
   - Find a stack containing only one coin, and remove that coin from the game.

   For example, the following sequence of turns describes a complete game:

$$\begin{aligned} \langle 4,2,1\rangle &\implies \langle 3,3,2\rangle \implies \langle 3,2,2,1,1\rangle \implies \langle 3,2,2,1\rangle \implies \langle 3,2,2\rangle \implies \langle 3,2,1,1,1\rangle \\ &\implies \langle 2,2,2,2,1\rangle \implies \langle 2,2,2,2\rangle \implies \langle 2,2,2,1,1,1\rangle \implies \langle 2,2,2,1,1\rangle \\ &\implies \langle 2,2,2,1\rangle \implies \langle 2,2,2\rangle \implies \langle 2,2,1,1,1\rangle \implies \langle 2,2,1,1\rangle \implies \langle 2,2,1\rangle \\ &\implies \langle 2,2\rangle \implies \langle 2,1,1,1\rangle \implies \langle 1,1,1,1,1,1\rangle \implies \langle 1,1,1,1,1\rangle \implies \langle 1,1,1,1\rangle \\ &\implies \langle 1,1,1\rangle \implies \langle 1,1\rangle \implies \langle 1\rangle \implies \langle\,\rangle. \end{aligned}$$

   Prove that this modified game still ends after a finite number of turns.

   (d) What are the minimum and maximum number of turns in this modified game, starting with a single stack of n coins? Prove your answers are correct.

9. (a) Prove that $|A \times B| = |A| \times |B|$ for all finite sets A and B.

   (b) Prove that for all *non-empty* finite sets A and B, there are exactly $|B|^{|A|}$ functions from A to B.

10. Recall that a binary tree is *full* if every node has either two children (an internal node) or no children (a leaf). Give at least *four different* proofs of the following fact: *In any full binary tree, the number of leaves is exactly one more than the number of internal nodes.*

11. The nth *Fibonacci binary tree* $\mathcal{F}_n$ is defined recursively as follows:

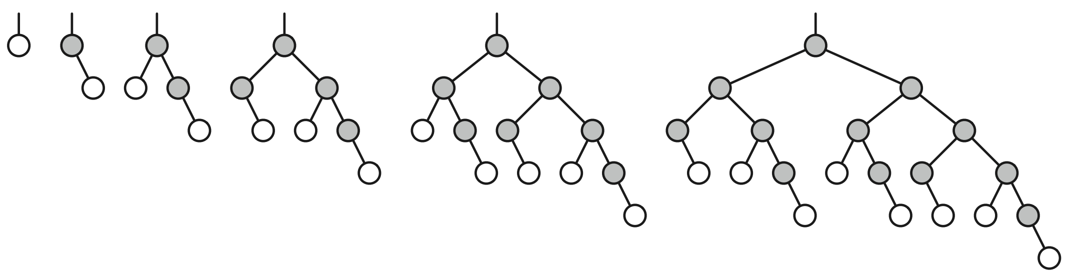
    - $\mathcal{F}_1$ is a single root node with no children.
    - For all $n \geq 2$, $\mathcal{F}_n$ is obtained from $\mathcal{F}_{n-1}$ by adding a right child to every leaf and adding a left child to every node that has only one child.

    (a) Prove that the number of leaves in $\mathcal{F}_n$ is precisely the nth Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$.

    (b) How many nodes does $\mathcal{F}_n$ have? Give an exact, closed-form answer in terms of Fibonacci numbers, and prove your answer is correct.

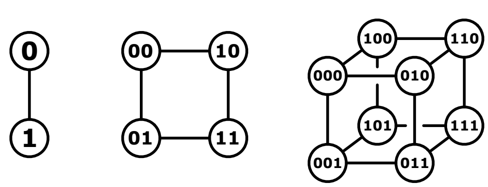
    (c) Prove that for all $n \geq 2$, the right subtree of $\mathcal{F}_n$ is a copy of $\mathcal{F}_{n-1}$.

    (d) Prove that for all $n \geq 3$, the left subtree of $\mathcal{F}_n$ is a copy of $\mathcal{F}_{n-2}$.



The first six Fibonacci binary trees. In each tree $\mathcal{F}_n$, the subtree of gray nodes is $\mathcal{F}_{n-1}$.

12. The d-dimensional hypercube is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if and only if their labels differ in exactly one bit.



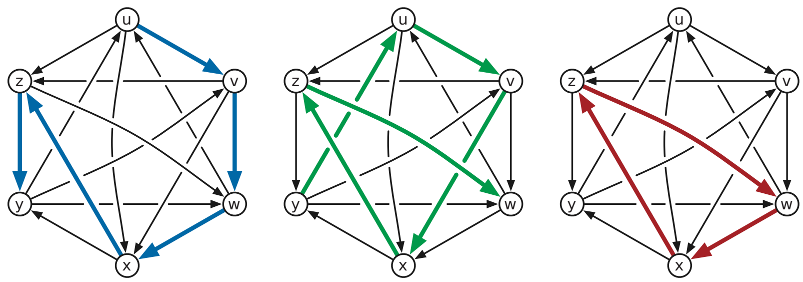
The 1-dimensional, 2-dimensional, and 3-dimensional hypercubes.

Recall that a Hamiltonian cycle is a closed walk that visits each vertex in a graph exactly once. Prove that for every integer $d \geq 2$, the d-dimensional hypercube has a Hamiltonian cycle.

13. A ***tournament*** is a directed graph with exactly one directed edge between each pair of vertices. That is, for any vertices v and w, a tournament contains either an edge $v \rightarrow w$ or an edge $w \rightarrow v$, but not both. A ***Hamiltonian path*** in a directed graph G is a directed path that visits every vertex of G exactly once.

    (a) Prove that every tournament contains a Hamiltonian path.

    (b) Prove that every tournament contains either *exactly one* Hamiltonian path or a directed cycle of length three.



A tournament with two Hamiltonian paths $u \rightarrow v \rightarrow w \rightarrow x \rightarrow z \rightarrow y$ and $y \rightarrow u \rightarrow v \rightarrow x \rightarrow z \rightarrow w$ and a directed triangle $w \rightarrow x \rightarrow z \rightarrow w$.

14. Scientists recently discovered a planet, tentatively named "Ygdrasil", that is inhabited by a bizarre species called "nertices" (singular "nertex"). All nertices trace their ancestry back to a particular nertex named Rudy. Rudy is still quite alive, as is every one of his many descendants. Nertices reproduce asexually; every nertex has exactly one parent (except Rudy, who sprang forth fully formed from the planet's core). There are three types of nertices—red, green, and blue. The color of each nertex is correlated exactly with the number and color of its children, as follows:

    - Each red nertex has two children, exactly one of which is green.
    - Each green nertex has exactly one child, which is not green.
    - Blue nertices have no children.

    In each of the following problems, let R, G, and B respectively denote the number of red, green, and blue nertices on Ygdrasil.

    (a) Prove that $B = R + 1$.

    (b) Prove that either $G = R$ or $G = B$.

    (c) Prove that $G = B$ if and only if Rudy is green.

15. ***Well-formed formulas*** (wffs) are defined recursively as follows:

    - T is a wff.
    - F is a wff.
    - Any proposition variable is a wff.
    - If X is a wff, then $(\neg X)$ is also a wff.
    - If X and Y are wffs, then $(X \wedge Y)$ is also a wff.

- If X and Y are wffs, then $(X \vee Y)$ is also a wff.

We say that a formula is in ***De Morgan normal form*** if it satisfies the following conditions. ("De Morgan normal form" is not standard terminology; I just made it up.)

- Every negation in the formula is applied to a variable, not to a more complicated subformula.
- Either the entire formula is T, or the formula does not contain T.
- Either the entire formula is F, or the formula does not contain F.

Prove that for every wff, there is a logically equivalent wff in De Morgan normal form. For example, the well-formed formula

$$(\neg((p \wedge q) \vee \neg r)) \wedge (\neg(p \vee \neg r) \wedge q)$$

is logically equivalent to the following wff in De Morgan normal form:

$$(((\neg p \vee \neg q) \wedge r)) \wedge ((\neg p \wedge r) \wedge q)$$

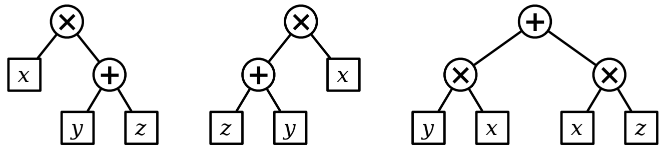
16. A ***polynomial*** is a function $f : \mathbb{R} \to \mathbb{R}$ of the form $f(x) = \sum_{i=0}^{d} a_i x^i$ for some non-negative integer d (called the degree) and some real numbers $a_0, a_1, \ldots, a_d$ (called the coefficients).
    (a) Prove that the sum of two polynomials is a polynomial.
    (b) Prove that the product of two polynomials is a polynomial.
    (c) Prove that the composition $f(g(x))$ of two polynomials $f(x)$ and $g(x)$ is a polynomial.
    (d) Prove that the derivative f' of a polynomial f is a polynomial, using ***only*** the following facts:
    - Constant rule: If f is constant, then f' is identically zero.
    - Sum rule: $(f + g)' = f' + g'$.
    - Product rule: $(f \cdot g)' = f' \cdot g + f \cdot g'$.

*17. An ***arithmetic expression tree*** is a binary tree where every leaf is labeled with a variable, every internal node is labeled with an arithmetic operation, and every internal node has exactly two children. For this problem, assume that the only allowed operations are **+** and **×**. Different leaves may or may not represent different variables.

Every arithmetic expression tree represents a function, transforming input values for the leaf variables into an output value for the root, by following two simple rules: (1) The value of any $+$-node is the sum of the values of its children. (2) The value of any $\times$-node is the product of the values of its children.

Two arithmetic expression trees are ***equivalent*** if they represent the same function; that is, the same input values for the leaf variables always leads to the same output value at both roots. An arithmetic expression tree is in ***normal form*** if the parent of every $+$-node (if any) is another $+$-node.

Prove that for any arithmetic expression tree, there is an equivalent arithmetic expression tree in normal form. *[Hint: This is harder than it looks.]*



Three equivalent expression trees. Only the third is in normal form.

*18. A ***Gaussian integer*** is a complex number of the form $x + yi$, where x and y are integers. Prove that any Gaussian integer can be expressed as the sum of distinct powers of the complex number $\alpha = -1 + i$. For example:

$$\begin{aligned}
4 &= 16 + (-8 - 8i) + 8i + (-4) &&= \alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 \\
-8 &= (-8 - 8i) + 8i &&= \alpha^7 + \alpha^6 \\
15i &= (-16 + 16i) + 16 + (-2i) + (-1 + i) + 1 &&= \alpha^9 + \alpha^8 + \alpha^2 + \alpha^1 + \alpha^0 \\
1 + 6i &= (8i) + (-2i) + 1 &&= \alpha^6 + \alpha^2 + \alpha^0 \\
2 - 3i &= (4 - 4i) + (-4) + (2 + 2i) + (-2i) + (-1 + i) + 1 &&= \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha^1 + \alpha^0 \\
-4 + 2i &= (-16 + 16i) + 16 + (-8 - 8i) + (4 - 4i) + (-2i) &&= \alpha^9 + \alpha^8 + \alpha^7 + \alpha^5 + \alpha^2
\end{aligned}$$

The following list of values may be helpful:

$\alpha^0 = 1$	$\alpha^4 = -4$	$\alpha^8 = 16$	$\alpha^{12} = -64$
$\alpha^1 = -1 + i$	$\alpha^5 = 4 - 4i$	$\alpha^9 = -16 + 16i$	$\alpha^{13} = 64 - 64i$
$\alpha^2 = -2i$	$\alpha^6 = 8i$	$\alpha^{10} = -32i$	$\alpha^{14} = 128i$
$\alpha^3 = 2 + 2i$	$\alpha^7 = -8 - 8i$	$\alpha^{11} = 32 + 32i$	$\alpha^{15} = -128 - 128i$

[Hint: How do you write $-2 - i$?]

*19. *Lazy binary* is a variant of standard binary notation for representing natural numbers where we allow each "bit" to take on one of three values: 0, 1, or 2. Lazy binary notation is defined inductively as follows.

- The lazy binary representation of zero is 0.
- Given the lazy binary representation of any non-negative integer n, we can construct the lazy binary representation of $n + 1$ as follows:
  (a) increment the rightmost digit;
  (b) if any digit is equal to 2, replace the rightmost 2 with 0 and increment the digit immediately to its left.

Here are the first several natural numbers in lazy binary notation:

0, 1, 10, 11, 20, 101, 110, 111, 120, 201, 210, 1011, 1020, 1101, 1110, 1111, 1120, 1201, 1210, 2011, 2020, 2101, 2110, 10111, 10120, 10201, 10210, 11011, 11020, 11101, 11110, 11111, 11120, 11201, 11210, 12011, 12020, 12101, 12110, 20111, 20120, 20201, 20210, 21011, 21020, 21101, 21110, 101111, 101120, 101201, 101210, 102011, 102020, . . .

(a) Prove that in any lazy binary number, between any two 2s there is at least one 0, and between two 0s there is at least one 2.

(b) Prove that for any natural number N, the sum of the digits of the lazy binary representation of N is exactly $\lfloor \lg(N + 1) \rfloor$.

★20. Consider the following recursively defined sequence of rational numbers:

$$R_0 = 0$$
$$R_n = \frac{1}{2\lfloor R_{n-1} \rfloor - R_{n-1} + 1} \qquad \text{for all } n \geq 1$$

The first several elements of this sequence are

$$0, 1, \frac{1}{2}, 2, \frac{1}{3}, \frac{3}{2}, \frac{2}{3}, 3, \frac{1}{4}, \frac{4}{3}, \frac{3}{5}, \frac{5}{2}, \frac{2}{5}, \frac{5}{3}, \frac{3}{4}, 4, \frac{1}{5}, \ldots$$

Prove that every non-negative rational number appears in this sequence exactly once.

21. Let $f : \mathbb{R}^+ \to \mathbb{R}^+$ be an ***arbitrary*** (not necessarily continuous) function such that

- $f(x) > 0$ for all $x > 0$, and
- $f(x) = \pi f(x/\sqrt{2})$ for all $x > 1$.

Prove by induction that $f(x) = \Theta(x)$ (as $x \to \infty$). Yes, this is induction over the real numbers.

*22. There is a natural generalization of induction to the real numbers that is familiar to analysts but relatively unknown in computer science. The precise formulation given below is was proposed independently by Hathaway[2] and Clark[3] fairly recently, but the idea dates back to at least to the 1920s. Recall that there are four types of intervals for any real numbers a and z:

- The ***open*** interval $\boldsymbol{(a,z)} := \{t \in \mathbb{R} \mid a \leq t < z\}$,
- The ***half-open*** intervals $\boldsymbol{[a,z)} := \{t \in \mathbb{R} \mid a \leq t < z\}$ and $\boldsymbol{(a,z]} := \{t \in \mathbb{R} \mid a < t \leq z\}$
- The ***closed*** interval $\boldsymbol{[a,z]} := \{t \in \mathbb{R} \mid a \leq t \leq z\}$.

Theorem 11 (Continuous Induction). *Fix a closed interval $[a,z] \subset \mathbb{R}$. Suppose some subset $S \subseteq [a,z]$ has following properties:*

(a) $a \in S$.

(b) If $a \leq s < z$ and $s \in S$, then $[s,u] \subseteq S$ for some $u > s$.

(c) If $a \leq s \leq z$ and $[a,s) \subseteq S$, then $s \in S$.

Then $S = [a,z]$.

[2]Dan Hathaway. Using continuity induction. *College Math. J.* 42:229–231, 2011.
[3]Pete L. Clark. The instructor's guide to real induction. arXiv:1208.0973.

> **Proof:** For the sake of argument, let S be a proper subset of $[a,b]$. Let $T = [a,z] \setminus S$. Because $\bar{S}$ is bounded but non-empty, it has a greatest lower bound $\ell \in [a,z]$. More explicitly, ℓ be the largest real number such that $\ell \le t$ for all $t \in T$. There are three cases to consider:
>
> - Suppose $\ell = a$. Condition (a) and (b) imply that $[a,u] \in S$ for some $u > a$. But then we have $\ell = a < u \le t$ for all $t \in T$, contradicting the fact that ℓ is the *greatest* lower bound of T.
> - Suppose $\ell > a$ and $\ell \in S$. If $\ell = z$, then $S = [a,z]$, contradicting our initial assumption. Otherwise, by condition (b), we have $[\ell,u] \subseteq S$ for some $u > \ell$, again contradicting the fact that ℓ is the *greatest* lower bound of T.
> - Finally, suppose $\ell > a$ and $\ell \in \overline{S}$. Because no element of T is smaller than ℓ, we have $[a,\ell) \subseteq S$. But then condition (c) implies that $\ell \in S$, and we have a contradiction.
>
> In all cases, we have a contradiction. □

Continuous induction hinges on the ***axiom of completeness***—every non-empty set of positive real numbers has a greatest lower bound—just as standard induction requires the ***well-ordering principle***—every non-empty set of positive integers has a smallest element. Thus, continuous induction cannot be used to prove properties of *rational* numbers, because the greatest lower bound of a set of rational numbers need not be rational.

Fix real numbers $a \le z$. Recall that a function $f : [a,z] \to \mathbb{R}$ is ***continuous*** if it satisfies the following condition: for any $t \in [a,z]$ and any $\varepsilon > 0$, there is some $\delta > 0$ such that for all $u \in [a,z]$ with $|t-u| \le \delta$, we have $|f(t) - f(u)| \le \varepsilon$. Prove the following theorems using continuous induction.

(a) ***Connectedness:*** *There is no continuous function from $[a,z]$ to the set $\{0,1\}$.*

(b) ***Intermediate Value Theorem:*** *For any continuous function $f : [a,z] \to \mathbb{R} \setminus \{0\}$, if $f(a) > 0$, then $f(t) > 0$ for all $a \le t \le z$.*

(c) ***Extreme Value Theorem:*** *Any continuous function $f : [a,z] \to \mathbb{R}$ attains its maximum value; that is, there is some $t \in [a,z]$ such that $f(t) \ge f(u)$ for all $u \in [a,z]$.*

★(d) ***The Heine-Borel Theorem:*** *The interval $[a,z]$ is compact.*

This one requires some expansion.

- A set $X \subseteq \mathbb{R}$ is ***open*** if every point in X lies inside an open interval contained in X.
- An ***open cover*** of $[a,z]$ is a (possibly uncountably infinite) family $\mathcal{U} = \{U_i \mid i \in I\}$ of open sets U_i such that $[a,z] \subseteq \bigcup_{i \in I} U_i$.
- A ***subcover*** of $\mathcal{U}$ is a subset $\mathcal{V} \subseteq \mathcal{U}$ that is also a cover of $[a,z]$.
- A cover $\mathcal{U}$ is ***finite*** if it contains a finite number of open sets.
- Finally, a set $X \subseteq \mathbb{R}$ is ***compact*** if every open cover of X has a finite subcover.

The Heine-Borel theorem is one of the most fundamental results in real analysis, and the proof usually requires several pages. But the continuous-induction proof is shorter than the list of definitions!

Change is certain. Peace is followed by disturbances; departure of evil men by their return. Such recurrences should not constitute occasions for sadness but realities for awareness, so that one may be happy in the interim.

— *I Ching [The Book of Changes]* (c. 1100 BC)

To endure *the idea of the recurrence one needs: freedom from morality; new means against the fact of pain (pain conceived as a tool, as the father of pleasure; there is no cumulative consciousness of displeasure); the enjoyment of all kinds of uncertainty, experimentalism, as a counterweight to this extreme fatalism; abolition of the concept of necessity; abolition of the "will"; abolition of "knowledge-in-itself."*

— Friedrich Nietzsche *The Will to Power* (1884)
[translated by Walter Kaufmann]

Wil Wheaton: *Embrace the dark side!*
Sheldon: *That's not even from your franchise!*

— "The Wheaton Recurrence", *Bing Bang Theory*, April 12, 2010

Solving Recurrences

1 Introduction

A ***recurrence*** is a recursive description of a function, or in other words, a description of a function in terms of itself. Like all recursive structures, a recurrence consists of one or more *base cases* and one or more *recursive cases*. Each of these cases is an equation or inequality, with some function value $f(n)$ on the left side. The base cases give explicit values for a (typically finite, typically small) subset of the possible values of n. The recursive cases relate the function value $f(n)$ to function value $f(k)$ for one or more integers $k < n$; typically, each recursive case applies to an infinite number of possible values of n.

For example, the following recurrence (written in two different but standard ways) describes the identity function $f(n) = n$:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1) + 1 & \text{otherwise} \end{cases} \qquad\qquad \begin{aligned} &f(0) = 0 \\ &f(n) = f(n-1) + 1 \text{ for all } n > 0 \end{aligned}$$

In both presentations, the first line is the only base case, and the second line is the only recursive case. The same function can satisfy *many* different recurrences; for example, both of the following recurrences also describe the identity function:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) & \text{otherwise} \end{cases} \qquad f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot f(n/2) & \text{if } n \text{ is even and } n > 0 \\ f(n-1) + 1 & \text{if } n \text{ is odd} \end{cases}$$

We say that a particular function ***satisfies*** a recurrence, or is the ***solution*** to a recurrence, if each of the statements in the recurrence is true. Most recurrences—at least, those that we will encounter in this class—have a solution; moreover, if every case of the recurrence is an equation, that solution is unique. Specifically, if we transform the recursive formula into a recursive *algorithm*, the solution to the recurrence is the function computed by that algorithm!

Recurrences arise naturally in the analysis of algorithms, especially recursive algorithms. In many cases, we can express the running time of an algorithm as a recurrence, where the recursive cases of the recurrence correspond exactly to the recursive cases of the algorithm. Recurrences are also useful tools for solving counting problems—How many objects of a particular kind exist?

By itself, a recurrence is not a satisfying description of the running time of an algorithm or a bound on the number of widgets. Instead, we need a ***closed-form*** solution to the recurrence; this is a *non-recursive* description of a function that satisfies the recurrence. For recurrence *equations*, we sometimes prefer an *exact* closed-form solution, but such a solution may not exist, or may be too complex to be useful. Thus, for most recurrences, especially those arising in algorithm analysis, we are satisfied with an *asymptotic* solution of the form $\Theta(g(n))$, for some explicit (non-recursive) function $g(n)$.

For recursive *inequalities*, we prefer a ***tight*** solution; this is a function that would still satisfy the recurrence if all the inequalities were replaced with the corresponding equations. Again, exactly tight solutions may not exist, or may be too complex to be useful, in which case we seek either a looser bound or an asymptotic solution of the form $O(g(n))$ or $\Omega(g(n))$.

2 The Ultimate Method: Guess and Confirm

Ultimately, there is only one fail-safe method to solve *any* recurrence:

Guess the answer, and then prove it correct by induction.

Later sections of these notes describe techniques to generate guesses that are guaranteed to be correct, provided you use them correctly. But if you're faced with a recurrence that doesn't seem to fit any of these methods, or if you've forgotten how those techniques work, don't despair! If you guess a closed-form solution and then try to verify your guess inductively, usually either the proof will succeed, in which case you're done, or the proof will fail, in which case *your failure will help you refine your guess*. Where you get your initial guess is utterly irrelevant[1]—from a classmate, from a textbook, on the web, from the answer to a different problem, scrawled on a bathroom wall in Siebel, included in a care package from your mom, dictated by the machine elves, whatever. If you can prove that the answer is correct, then it's correct!

2.1 Tower of Hanoi

The classical Tower of Hanoi problem gives us the recurrence $\boldsymbol{T(n) = 2T(n-1) + 1}$ with base case $\boldsymbol{T(0) = 0}$. Just looking at the recurrence we can guess that $T(n)$ is something like 2^n. If we write out the first few values of $T(n)$, we discover that they are each one less than a power of two.

$$T(0) = 0, \quad T(1) = 1, \quad T(2) = 3, \quad T(3) = 7, \quad T(4) = 15, \quad T(5) = 31, \quad T(6) = 63, \quad \ldots,$$

It looks like $\boldsymbol{T(n) = 2^n - 1}$ might be the right answer. Let's check.

$$T(0) = 0 = 2^0 - 1 \quad \checkmark$$

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 && \text{[induction hypothesis]} \\ &= 2^n - 1 \quad \checkmark && \text{[algebra]} \end{aligned}$$

[1]. . . except of course during exams, where you aren't supposed to use *any* outside sources

We were right! Hooray, we're done!

Another way we can guess the solution is by ***unrolling*** the recurrence, by substituting it into itself:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2\,(2T(n-2)+1)+1 \\ &= 4T(n-2)+3 \\ &= 4\,(2T(n-3)+1)+3 \\ &= 8T(n-3)+7 \\ &= \cdots \end{aligned}$$

It looks like unrolling the initial Hanoi recurrence k times, for any non-negative integer k, will give us the new recurrence $T(n) = 2^k T(n-k) + (2^k - 1)$. Let's prove this by induction:

$$T(n) = 2T(n-1)+1 \quad \checkmark \qquad [k=0, \text{ by definition}]$$

$$\begin{aligned} T(n) &= 2^{k-1}T(n-(k-1)) + (2^{k-1}-1) && \text{[inductive hypothesis]} \\ &= 2^{k-1}\big(2T(n-k)+1\big) + (2^{k-1}-1) && [\text{initial recurrence for } T(n-(k-1))] \\ &= 2^k T(n-k) + (2^k - 1) \quad \checkmark && \text{[algebra]} \end{aligned}$$

Our guess was correct! In particular, unrolling the recurrence n times give us the recurrence $T(n) = 2^n T(0) + (2^n - 1)$. Plugging in the base case $T(0) = 0$ give us the closed-form solution $T(n) = 2^n - 1$.

2.2 Fibonacci numbers

Let's try a less trivial example: the Fibonacci numbers $\boldsymbol{F_n = F_{n-1} + F_{n-2}}$ with base cases $\boldsymbol{F_0 = 0}$ and $\boldsymbol{F_1 = 1}$. There is no obvious pattern in the first several values (aside from the recurrence itself), but we can reasonably guess that F_n is exponential in n. Let's try to prove inductively that $F_n \le \alpha \cdot c^n$ for some constants $a > 0$ and $c > 1$ and see how far we get.

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &\le \alpha \cdot c^{n-1} + \alpha \cdot c^{n-2} && [\text{“induction hypothesis”}] \\ &\le \alpha \cdot c^n \ \textbf{???} \end{aligned}$$

The last inequality is satisfied if $c^n \ge c^{n-1} + c^{n-2}$, or more simply, if $c^2 - c - 1 \ge 0$. The smallest value of c that works is $\phi = (1+\sqrt{5})/2 \approx 1.618034$; the other root of the quadratic equation has smaller absolute value, so we can ignore it.

So we have *most* of an inductive proof that $\boldsymbol{F_n \le \alpha \cdot \phi^n}$ for *some* constant α. All that we're missing are the base cases, which (we can easily guess) must determine the value of the coefficient a. We quickly compute

$$\frac{F_0}{\phi^0} = \frac{0}{1} = 0 \quad \text{and} \quad \frac{F_1}{\phi^1} = \frac{1}{\phi} \approx 0.618034 > 0,$$

so the base cases of our induction proof are correct as long as $\alpha \ge 1/\phi$. It follows that $\boldsymbol{F_n \le \phi^{n-1}}$ for all $n \ge 0$.

What about a matching lower bound? Essentially the same inductive proof implies that $F_n \ge \beta \cdot \phi^n$ for some constant β, but the only value of β that works for *all* n is the trivial $\beta = 0$!

We could try to find some lower-order term that makes the base case non-trivial, but an easier approach is to recall that asymptotic $\Omega(\,)$ bounds only have to work for *sufficiently large n*. So let's ignore the trivial base case $F_0 = 0$ and assume that $F_2 = 1$ is a base case instead. Some more easy calculation gives us

$$\frac{F_2}{\phi^2} = \frac{1}{\phi^2} \approx 0.381966 < \frac{1}{\phi}.$$

Thus, the new base cases of our induction proof are correct as long as $\beta \le 1/\phi^2$, which implies that $\boldsymbol{F_n \ge \phi^{n-2}}$ for all $n \ge 1$.

Putting the upper and lower bounds together, we obtain the tight asymptotic bound $\boldsymbol{F_n = \Theta(\phi^n)}$. It *is* possible to get a more exact solution by speculatively refining and conforming our current bounds, but it's not easy. Fortunately, if we really need it, we can get an exact solution using the *annihilator* method, which we'll see later in these notes.

2.3 Mergesort

Mergesort is a classical recursive divide-and-conquer algorithm for sorting an array. The algorithm splits the array in half, recursively sorts the two halves, and then merges the two sorted subarrays into the final sorted array.

```
MergeSort(A[1..n]):
  if (n > 1)
      m ← ⌊n/2⌋
      MergeSort(A[1..m])
      MergeSort(A[m+1..n])
      Merge(A[1..n], m)
```

```
Merge(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i+1
      else if i > m
          B[k] ← A[j]; j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i+1
      else
          B[k] ← A[j]; j ← j+1

  for k ← 1 to n
      A[k] ← B[k]
```

Let $T(n)$ denote the worst-case running time of MergeSort when the input array has size n. The Merge subroutine clearly runs in $\Theta(n)$ time, so the function $T(n)$ satisfies the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + \Theta(n) & \text{otherwise.} \end{cases}$$

For now, let's consider the special case where n is a power of 2; this assumption allows us to take the floors and ceilings out of the recurrence. (We'll see how to deal with the floors and ceilings later; the short version is that they don't matter.)

Because the recurrence itself is given only asymptotically—in terms of $\Theta(\,)$ expressions—we can't hope for anything but an asymptotic solution. So we can safely simplify the recurrence further by removing the Θ's; any asymptotic solution to the simplified recurrence will also satisfy the original recurrence. (This simplification is actually important for another reason; if we kept the asymptotic expressions, we might be tempted to simplify them inappropriately.)

Our simplified recurrence now looks like this:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise.} \end{cases}$$

To guess at a solution, let's try unrolling the recurrence.

$$\begin{aligned}T(n) &= 2T(n/2) + n \\ &= 2\big(2T(n/4) + n/2\big) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n = \cdots\end{aligned}$$

It looks like $T(n)$ satisfies the recurrence $T(n) = 2^k T(n/2^k) + kn$ for *any* positive integer k. Let's verify this by induction.

$$T(n) = 2T(n/2) + n = 2^1 T(n/2^1) + 1 \cdot n \quad \checkmark \qquad [k = 1, \text{ given recurrence}]$$

$$\begin{aligned}T(n) &= 2^{k-1} T(n/2^{k-1}) + (k-1)n && \text{[inductive hypothesis]} \\ &= 2^{k-1}\big(2T(n/2^k) + n/2^{k-1}\big) + (k-1)n && \text{[substitution]} \\ &= 2^k T(n/2^k) + kn \quad \checkmark && \text{[algebra]}\end{aligned}$$

Our guess was right! The recurrence becomes trivial when $n/2^k = 1$, or equivalently, when $k = \log_2 n$:

$$T(n) = nT(1) + n\log_2 n = n\log_2 n + n.$$

Finally, we have to put back the Θ's we stripped off; our final closed-form solution is $\boldsymbol{T(n) = \Theta(n \log n)}$.

2.4 An uglier divide-and-conquer example

Consider the divide-and-conquer recurrence $\boldsymbol{T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n}$. This doesn't fit into the form required by the Master Theorem (which we'll see below), but it still sort of resembles the Mergesort recurrence—the total size of the subproblems at the first level of recursion is n—so let's *guess* that $T(n) = O(n \log n)$, and then try to prove that our guess is correct. (We could also attack this recurrence by unrolling, but let's see how far just guessing will take us.)

Let's start by trying to prove an upper bound $T(n) \le a\, n \lg n$ for all sufficiently large n and some constant a to be determined later:

$$\begin{aligned}T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\ &\le \sqrt{n} \cdot a\,\sqrt{n} \lg \sqrt{n} + n && \text{[induction hypothesis]} \\ &= (a/2) n \lg n + n && \text{[algebra]} \\ &\le an \lg n \;\checkmark && \text{[algebra]}\end{aligned}$$

The last inequality assumes only that $1 \le (a/2)\log n$,or equivalently, that $n \ge 2^{2/a}$. In other words, the induction proof is correct if n is sufficiently large. So we were right!

But before you break out the champagne, what about the multiplicative constant a? The proof worked for *any* constant a, no matter how small. This strongly suggests that our upper bound $T(n) = O(n \log n)$ is not tight. Indeed, if we try to prove a matching lower bound $T(n) \ge b\, n \log n$ for sufficiently large n, we run into trouble.

$$\begin{aligned}T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\ &\ge \sqrt{n} \cdot b\,\sqrt{n} \log \sqrt{n} + n && \text{[induction hypothesis]} \\ &= (b/2) n \log n + n \\ &\not\ge bn \log n\end{aligned}$$

The last inequality would be correct only if $1 > (b/2)\log n$, but that inequality is false for large values of n, no matter which constant b we choose.

Okay, so $\Theta(n \log n)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n \geq n \checkmark$$

But an inductive proof of the upper bound fails.

$$\begin{aligned} T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\ &\leq \sqrt{n} \cdot a\sqrt{n} + n && \text{[induction hypothesis]} \\ &= (a+1)n && \text{[algebra]} \\ &\not\leq an \end{aligned}$$

Hmmm. So what's bigger than n and smaller than $n \lg n$? How about $n\sqrt{\lg n}$?

$$\begin{aligned} T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\leq \sqrt{n} \cdot a\sqrt{n}\sqrt{\lg \sqrt{n}} + n && \text{[induction hypothesis]} \\ &= (a/\sqrt{2})\, n\sqrt{\lg n} + n && \text{[algebra]} \\ &\leq a\, n\sqrt{\lg n} \quad \text{for large enough } n \checkmark \end{aligned}$$

Okay, the upper bound checks out; how about the lower bound?

$$\begin{aligned} T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\geq \sqrt{n} \cdot b\sqrt{n}\sqrt{\lg \sqrt{n}} + n && \text{[induction hypothesis]} \\ &= (b/\sqrt{2})\, n\sqrt{\lg n} + n && \text{[algebra]} \\ &\not\geq b\, n\sqrt{\lg n} \end{aligned}$$

No, the last step doesn't work. So $\Theta(n\sqrt{\lg n})$ doesn't work.

Okay... what else is between n and $n \lg n$? How about $n \lg\lg n$?

$$\begin{aligned} T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\leq \sqrt{n} \cdot a\sqrt{n} \lg\lg \sqrt{n} + n && \text{[induction hypothesis]} \\ &= a\, n \lg\lg n - a\, n + n && \text{[algebra]} \\ &\leq a\, n \lg\lg n \qquad \textbf{if } \boldsymbol{a \geq 1} \checkmark \end{aligned}$$

Hey look at that! For once, our upper bound proof requires a constraint on the hidden constant a. This is an good indication that we've found the right answer. Let's try the lower bound:

$$\begin{aligned} T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n &\geq \sqrt{n} \cdot b\sqrt{n} \lg\lg \sqrt{n} + n && \text{[induction hypothesis]} \\ &= b\, n \lg\lg n - b\, n + n && \text{[algebra]} \\ &\geq b\, n \lg\lg n \qquad \textbf{if } \boldsymbol{b \leq 1} \checkmark \end{aligned}$$

Hey, it worked! We have most of an inductive proof that $T(n) \leq an \lg\lg n$ for any $a \geq 1$ and most of an inductive proof that $T(n) \geq bn \lg\lg n$ for any $b \leq 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $\boldsymbol{T(n) = \Theta(n \log\log n)}$.

3 Divide and Conquer Recurrences (Recursion Trees)

Many divide and conquer algorithms give us running-time recurrences of the form

$$\boxed{T(n) = a\,T(n/b) + f(n)} \tag{1}$$

where a and b are constants and $f(n)$ is some other function. There is a simple and general technique for solving many recurrences in this and similar forms, using a ***recursion tree***. The root of the recursion tree is a box containing the value $f(n)$; the root has a children, each of which is the root of a (recursively defined) recursion tree for the function $T(n/b)$.

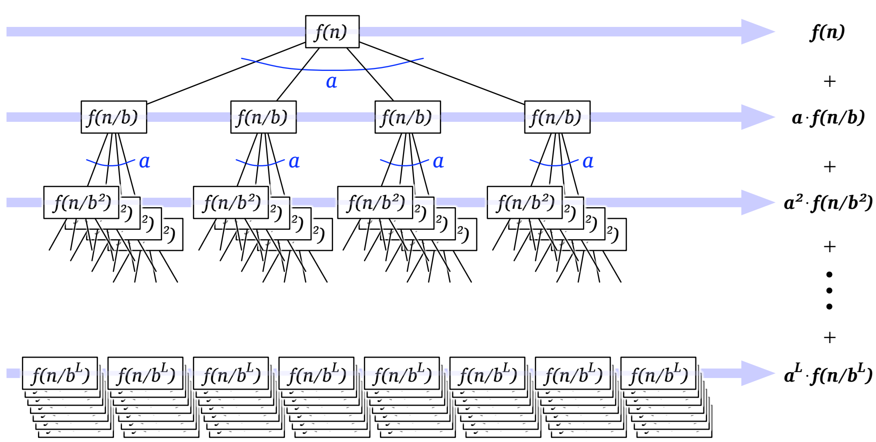
Equivalently, a recursion tree is a complete a-ary tree where each node at depth i contains the value $f(n/b^i)$. The recursion stops when we get to the base case(s) of the recurrence. Because we're only looking for asymptotic bounds, the exact base case doesn't matter; we can safely assume that $T(1) = \Theta(1)$, or even that $T(n) = \Theta(1)$ for all $n \le 10^{100}$. I'll also assume for simplicity that n is an integral power of b; we'll see how to avoid this assumption later (but to summarize: it doesn't matter).

Now $T(n)$ is just the sum of all values stored in the recursion tree. For each i, the ith level of the tree contains a^i nodes, each with value $f(n/b^i)$. Thus,

$$\boxed{T(n) = \sum_{i=0}^{L} a^i f(n/b^i)} \qquad (\Sigma)$$

where L is the depth of the recursion tree. We easily see that $\boldsymbol{L = \log_b n}$, because $n/b^L = 1$. The base case $f(1) = \Theta(1)$ implies that the last non-zero term in the summation is $\Theta(a^L) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$.

For *most* divide-and-conquer recurrences, the level-by-level sum (Σ) is a *geometric* series—each term is a constant factor larger or smaller than the previous term. In this case, only the largest term in the geometric series matters; all of the other terms are swallowed up by the $\Theta(\cdot)$ notation.



A recursion tree for the recurrence $T(n) = a\,T(n/b) + f(n)$

Here are several examples of the recursion-tree technique in action:

- **Mergesort (simplified): $\boldsymbol{T(n) = 2T(n/2) + n}$**

  There are 2^i nodes at level i, each with value $n/2^i$, so every term in the level-by-level sum (Σ) is the same:
  $$T(n) = \sum_{i=0}^{L} n.$$
  The recursion tree has $L = \log_2 n$ levels, so $\boldsymbol{T(n) = \Theta(n \log n)}$.

- **Randomized selection: $T(n) = T(3n/4) + n$**

  The recursion tree is a single path. The node at depth i has value $(3/4)^i n$, so the level-by-level sum (Σ) is a decreasing geometric series:

  $$T(n) = \sum_{i=0}^{L} (3/4)^i n.$$

  This geometric series is dominated by its initial term n, so $\boldsymbol{T(n) = \Theta(n)}$. The recursion tree has $L = \log_{4/3} n$ levels, but so what?

- **Karatsuba's multiplication algorithm: $T(n) = 3T(n/2) + n$**

  There are 3^i nodes at depth i, each with value $n/2^i$, so the level-by-level sum (Σ) is an increasing geometric series:

  $$T(n) = \sum_{i=0}^{L} (3/2)^i n.$$

  This geometric series is dominated by its final term $(3/2)^L n$. Each leaf contributes 1 to this term; thus, the final term is equal to the number of leaves in the tree! The recursion tree has $L = \log_2 n$ levels, and therefore $3^{\log_2 n} = n^{\log_2 3}$ leaves, so $\boldsymbol{T(n) = \Theta(n^{\log_2 3})}$.

- **$T(n) = 2T(n/2) + n/\lg n$**

  The sum of all the nodes in the ith level is $n/(\lg n - i)$. This implies that the depth of the tree is at most $\lg n - 1$. The level sums are neither constant nor a geometric series, so we just have to evaluate the overall sum directly.

  Recall (or if you're seeing this for the first time: Behold!) that the nth *harmonic number* H_n is the sum of the reciprocals of the first n positive integers:

  $$H_n := \sum_{i=1}^{n} \frac{1}{i}$$

  It's not hard to show that $H_n = \Theta(\log n)$; in fact, we have the stronger inequalities $\ln(n+1) \le H_n \le \ln n + 1$.

  $$T(n) = \sum_{i=0}^{\lg n - 1} \frac{n}{\lg n - i} = \sum_{j=1}^{\lg n} \frac{n}{j} = nH_{\lg n} = \boldsymbol{\Theta(n \lg \lg n)}$$

- **$T(n) = 4T(n/2) + n \lg n$**

  There are 4^i nodes at each level i, each with value $(n/2^i)\lg(n/2^i) = (n/2^i)(\lg n - i)$; again, the depth of the tree is at most $\lg n - 1$. We have the following summation:

  $$T(n) = \sum_{i=0}^{\lg n - 1} n 2^i (\lg n - i)$$

  We can simplify this sum by substituting $j = \lg n - i$:

  $$T(n) = \sum_{j=i}^{\lg n} n 2^{\lg n - j} j = \sum_{j=i}^{\lg n} \frac{n^2 j}{2^j} = n^2 \sum_{j=i}^{\lg n} \frac{j}{2^j} = \boldsymbol{\Theta(n^2)}$$

The last step uses the fact that $\sum_{i=1}^{\infty} j/2^j = 2$. Although this is not quite a geometric series, it is still dominated by its largest term.

- **Ugly divide and conquer: $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n$**

  We solved this recurrence earlier by guessing the right answer and verifying, but we can use recursion trees to get the correct answer directly. The *degree* of the nodes in the recursion tree is no longer constant, so we have to be a bit more careful, but the same basic technique still applies. It's not hard to see that the nodes in any level sum to n. The depth L satisfies the identity $n^{2^{-L}} = 2$ (we can't get all the way down to 1 by taking square roots), so $L = \lg\lg n$ and $\boldsymbol{T(n) = \Theta(n \lg\lg n)}$.

- **Randomized quicksort: $T(n) = T(3n/4) + T(n/4) + n$**

  This recurrence isn't in the standard form described earlier, but we can still solve it using recursion trees. Now modes in the same level of the recursion tree have different values, and different leaves are at different levels. However, the nodes in any *complete* level (that is, above any of the leaves) sum to n. Moreover, every leaf in the recursion tree has depth between $\log_4 n$ and $\log_{4/3} n$. To derive an upper bound, we overestimate $T(n)$ by ignoring the base cases and extending the tree downward to the level of the *deepest* leaf. Similarly, to derive a lower bound, we overestimate $T(n)$ by counting only nodes in the tree up to the level of the *shallowest* leaf. These observations give us the upper and lower bounds $n \log_4 n \le T(n) \le n \log_{4/3} n$. Since these bounds differ by only a constant factor, we have $\boldsymbol{T(n) = \Theta(n \log n)}$.

- **Deterministic selection: $T(n) = T(n/5) + T(7n/10) + n$**

  Again, we have a lopsided recursion tree. If we look only at complete levels of the tree, we find that the level sums form a descending geometric series $T(n) = n + 9n/10 + 81n/100 + \cdots$. We can get an upper bound by ignoring the base cases entirely and growing the tree out to infinity, and we can get a lower bound by only counting nodes in complete levels. Either way, the geometric series is dominated by its largest term, so $\boldsymbol{T(n) = \Theta(n)}$.

- **Randomized search trees: $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$**

  This looks like a divide-and-conquer recurrence, but what does it mean to have a quarter of a child? The right approach is to imagine that each node in the recursion tree has a *weight* in addition to its value. Alternately, we get a standard recursion tree again if we add a second real parameter to the recurrence, defining $T(n) = T(n, 1)$, where

  $$T(n, \alpha) = T(n/4, \alpha/4) + T(3n/4, 3\alpha/4) + \alpha.$$

  In each complete level of the tree, the (weighted) node values sum to exactly 1. The leaves of the recursion tree are at different levels, but all between $\log_4 n$ and $\log_{4/3} n$. So we have upper and lower bounds $\log_4 n \le T(n) \le \log_{4/3} n$, which differ by only a constant factor, so $\boldsymbol{T(n) = \Theta(\log n)}$.

- **Ham-sandwich trees: $T(n) = T(n/2) + T(n/4) + 1$**

  Again, we have a lopsided recursion tree. If we only look at complete levels, we find that the level sums form an *ascending* geometric series $T(n) = 1 + 2 + 4 + \cdots$, so the solution is dominated by the number of leaves. The recursion tree has $\log_4 n$ complete levels, so there are more than $2^{\log_4 n} = n^{\log_4 2} = \sqrt{n}$; on the other hand, every leaf has depth at most $\log_2 n$, so the total number of leaves is at most $2^{\log_2 n} = n$. Unfortunately, the crude bounds $\boldsymbol{\sqrt{n} \ll T(n) \ll n}$ are the best we can derive using the techniques we know so far!

The following theorem completely describes the solution for any divide-and-conquer recurrence in the 'standard form' $T(n) = aT(n/b) + f(n)$, where a and b are constants and $f(n)$ is a polynomial. This theorem allows us to bypass recursion trees for "standard" divide-and-conquer recurrences, but many people (including Jeff) find it harder to even remember the statement of the theorem than to use the more powerful and general recursion-tree technique. Your mileage may vary.

The Master Theorem. *The recurrence $T(n) = aT(n/b) + f(n)$ can be solved as follows.*

- *If $a\,f(n/b) = \kappa\, f(n)$ for some constant $\kappa < 1$, then $T(n) = \Theta(f(n))$.*
- *If $a\,f(n/b) = K\, f(n)$ for some constant $K > 1$, then $T(n) = \Theta(n^{\log_b a})$.*
- *If $a\,f(n/b) = f(n)$, then $T(n) = \Theta(f(n)\log_b n)$.*
- *If none of these three cases apply, you're on your own.*

Proof: If $f(n)$ is a *constant factor larger* than $a\,f(b/n)$, then by induction, the level sums define a descending geometric series. The sum of any geometric series is a constant times its largest term. In this case, the largest term is the first term $f(n)$.

If $f(n)$ is a *constant factor smaller* than $a\,f(b/n)$, then by induction, the level sums define an ascending geometric series. The sum of any geometric series is a constant times its largest term. In this case, this is the last term, which by our earlier argument is $\Theta(n^{\log_b a})$.

Finally, if $a\,f(b/n) = f(n)$, then by induction, each of the $L+1$ terms in the sum is equal to $f(n)$, and the recursion tree has depth $L = \Theta(\log_b n)$. □

★4 The Nuclear Bomb

Finally, let me describe *without proof* a powerful generalization of the recursion tree method, first published by Lebanese researchers Mohamad Akra and Louay Bazzi in 1998. Consider a general divide-and-conquer recurrence of the form

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + f(n),$$

where k is a constant, $a_i > 0$ and $b_i > 1$ are constants for all i, and $f(n) = \Omega(n^c)$ and $f(n) = O(n^d)$ for some constants $0 < c \le d$. (As usual, we assume the standard base case $T(\Theta(1)) = \Theta(1)$.) Akra and Bazzi prove that this recurrence has the closed-form asymptotic solution

$$T(n) = \Theta\left(n^\rho \left(1 + \int_1^n \frac{f(u)}{u^{\rho+1}}\, du\right)\right),$$

where ρ is the unique real solution to the equation

$$\sum_{i=1}^{k} a_i / b_i^\rho = 1.$$

In particular, the Akra-Bazzi theorem immediately implies the following form of the Master Theorem:

$$T(n) = aT(n/b) + n^c \implies T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } c < \log_b a - \varepsilon \\ \Theta(n^c \log n) & \text{if } c = \log_b a \\ \Theta(n^c) & \text{if } c > \log_b a + \varepsilon \end{cases}$$

The Akra-Bazzi theorem does not require that the parameters a_i and b_i are integers, or even rationals; on the other hand, even when all parameters are integers, the characteristic equation $\sum_i a_i/b_i^\rho = 1$ may have no analytical solution.

Here are a few examples of recurrences that are difficult (or impossible) for recursion trees, but have easy solutions using the Akra-Bazzi theorem.

- **Randomized quicksort: $T(n) = T(3n/4) + T(n/4) + n$**

  The equation $(3/4)^\rho + (1/4)^\rho = 1$ has the unique solution $\rho = 1$, and therefore

  $$T(n) = \Theta\left(n\left(1 + \int_1^n \frac{1}{u}\,du\right)\right) = O(n \log n).$$

- **Deterministic selection: $T(n) = T(n/5) + T(7n/10) + n$**

  The equation $(1/5)^\rho + (7/10)^\rho = 1$ has no analytical solution. However, we easily observe that $(1/5)^x + (7/10)^x$ is a decreasing function of x, and therefore $0 < \rho < 1$. Thus, we have

  $$\int_1^n \frac{f(u)}{u^{\rho+1}}\,du = \int_1^n u^{-\rho}\,du = \left.\frac{u^{1-\rho}}{1-\rho}\right|_{u=1}^n = \frac{n^{1-\rho}-1}{1-\rho} = \Theta(n^{1-\rho}),$$

  and therefore

  $$T(n) = \Theta(n^\rho \cdot (1 + \Theta(n^{1-\rho})) = \Theta(n).$$

  (A bit of numerical computation gives the approximate value $\rho \approx 0.83978$, but why bother?)

- **Randomized search trees: $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$**

  The equation $\frac{1}{4}(\frac{1}{4})^\rho + \frac{3}{4}(\frac{3}{4})^\rho = 1$ has the unique solution $\rho = 0$, and therefore

  $$T(n) = \Theta\left(1 + \int_1^n \frac{1}{u}\,du\right) = \Theta(\log n).$$

- **Ham-sandwich trees: $T(n) = T(n/2) + T(n/4) + 1$.** Recall that we could only prove the very weak bounds $\sqrt{n} \ll T(n) \ll n$ using recursion trees. The equation $(1/2)^\rho + (1/4)^\rho = 1$ has the unique solution $\rho = \log_2((1+\sqrt{5})/2) \approx 0.69424$, which can be obtained by setting $x = 2^\rho$ and solving for x. Thus, we have

  $$\int_1^n \frac{1}{u^{\rho+1}}\,du = \left.\frac{u^{-\rho}}{-\rho}\right|_{u=1}^n = \frac{1-n^{-\rho}}{\rho} = \Theta(1)$$

  and therefore

  $$T(n) = \Theta\left(n^\rho(1 + \Theta(1))\right) = \Theta(n^{\lg\phi}).$$

The Akra-Bazzi method is that it can solve *almost* any divide-and-conquer recurrence with just a few lines of calculation. (There are a few nasty exceptions like $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$ where we have to fall back on recursion trees.) On the other hand, the steps appear to be magic, which makes the method hard to remember, and for most divide-and-conquer recurrences that arise in practice, there are much simpler solution techniques.

*5 Linear Recurrences (Annihilators)

Another common class of recurrences, called ***linear*** recurrences, arises in the context of recursive backtracking algorithms and counting problems. These recurrences express each function value $f(n)$ as a *linear* combination of a small number of nearby values $f(n-1), f(n-2), f(n-3), \ldots$. The Fibonacci recurrence is a typical example:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

It turns out that the solution to *any* linear recurrence is a simple combination of polynomial and exponential functions in n. For example, we can verify by induction that the linear recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \text{ or } n = 2 \\ 3T(n-1) - 8T(n-2) + 4T(n-3) & \text{otherwise} \end{cases}$$

has the closed-form solution $\boldsymbol{T(n) = (n-3)2^n + 4}$. First we check the base cases:

$$\begin{aligned} T(0) &= (0-3)2^0 + 4 = 1 \quad \checkmark \\ T(1) &= (1-3)2^1 + 4 = 0 \quad \checkmark \\ T(2) &= (2-3)2^2 + 4 = 0 \quad \checkmark \end{aligned}$$

And now the recursive case:

$$\begin{aligned} T(n) &= 3T(n-1) - 8T(n-2) + 4T(n-3) \\ &= 3((n-4)2^{n-1} + 4) - 8((n-5)2^{n-2} + 4) + 4((n-6)2^{n-3} + 4) \\ &= \left(\frac{3}{2} - \frac{8}{4} + \frac{4}{8}\right) n \cdot 2^n - \left(\frac{12}{2} - \frac{40}{4} + \frac{24}{8}\right) 2^n + (2 - 8 + 4) \cdot 4 \\ &= (n-3) \cdot 2^n + 4 \quad \checkmark \end{aligned}$$

But how could we have possibly come up with that solution? In this section, I'll describe a general method for solving linear recurrences that's arguably easier than the induction proof!

5.1 Operators

Our technique for solving linear recurrences relies on the theory of ***operators***. Operators are higher-order functions, which take one or more functions as input and produce different functions as output. For example, your first two semesters of calculus focus almost exclusively on the *differential* and *integral* operators $\frac{d}{dx}$ and $\int dx$. All the operators we will need are combinations of three elementary building blocks:

- **Sum:** $(f+g)(n) := f(n)+g(n)$
- **Scale:** $(\alpha \cdot f)(n) := \alpha \cdot (f(n))$
- **Shift:** $(\boldsymbol{E}f)(n) := f(n+1)$

The shift and scale operators are ***linear***, which means they can be distributed over sums; for example, for any functions f, g, and h, we have $\boldsymbol{E}(f-3(g-h)) = \boldsymbol{E}f + (-3)\boldsymbol{E}g + 3\boldsymbol{E}h$.

We can combine these building blocks to obtain more complex *compound* operators. For example, the compound operator $\boldsymbol{E}-2$ is defined by setting $(\boldsymbol{E}-2)f := \boldsymbol{E}f + (-2)f$ for any function f. We can also apply the shift operator twice: $(\boldsymbol{E}(\boldsymbol{E}f))(n) = f(n+2)$; we write usually $\boldsymbol{E}^2 f$ as a synonym for $\boldsymbol{E}(\boldsymbol{E}f)$. More generally, for any positive integer k, the operator $\boldsymbol{E}^k$ shifts its argument k times: $\boldsymbol{E}^k f(n) = f(n+k)$. Similarly, $(\boldsymbol{E}-2)^2$ is shorthand for the operator $(\boldsymbol{E}-2)(\boldsymbol{E}-2)$, which applies $(\boldsymbol{E}-2)$ twice.

For example, here are the results of applying different operators to the exponential function $f(n) = 2^n$:

$$\begin{aligned}
2f(n) &= 2\cdot 2^n = 2^{n+1}\\
3f(n) &= 3\cdot 2^n\\
\boldsymbol{E}f(n) &= 2^{n+1}\\
\boldsymbol{E}^2 f(n) &= 2^{n+2}\\
(\boldsymbol{E}-2)f(n) &= \boldsymbol{E}f(n) - 2f(n) = 2^{n+1} - 2^{n+1} = 0\\
(\boldsymbol{E}^2-1)f(n) &= \boldsymbol{E}^2 f(n) - f(n) = 2^{n+2} - 2^n = 3\cdot 2^n
\end{aligned}$$

These compound operators can be manipulated exactly as though they were polynomials over the "variable" $\boldsymbol{E}$. In particular, we can factor compound operators into "products" of simpler operators, which can be applied in any order. For example, the compound operators $\boldsymbol{E}^2 - 3\boldsymbol{E} + 2$ and $(\boldsymbol{E}-1)(\boldsymbol{E}-2)$ are equivalent:

$$\text{Let}\quad g(n) := (\boldsymbol{E}-2)f(n) = f(n+1) - 2f(n).$$

$$\begin{aligned}
\text{Then}\quad (\boldsymbol{E}-1)(\boldsymbol{E}-2)f(n) &= (\boldsymbol{E}-1)g(n)\\
&= g(n+1) - g(n)\\
&= (f(n+2) - 2f(n-1)) - (f(n+1) - 2f(n))\\
&= f(n+2) - 3f(n+1) + 2f(n)\\
&= (\boldsymbol{E}^2 - 3\boldsymbol{E} + 2)f(n). \quad \checkmark
\end{aligned}$$

It is an easy exercise to confirm that $\boldsymbol{E}^2 - 3\boldsymbol{E} + 2$ is also equivalent to the operator $(\boldsymbol{E}-2)(\boldsymbol{E}-1)$.

The following table summarizes everything we need to remember about operators.

Operator	**Definition**
addition	$(f+g)(n) := f(n)+g(n)$
subtraction	$(f-g)(n) := f(n)-g(n)$
multiplication	$(\alpha\cdot f)(n) := \alpha\cdot(f(n))$
shift	$\boldsymbol{E}f(n) := f(n+1)$
k-fold shift	$\boldsymbol{E}^k f(n) := f(n+k)$
composition	$(\boldsymbol{X}+\boldsymbol{Y})f := \boldsymbol{X}f + \boldsymbol{Y}f$
	$(\boldsymbol{X}-\boldsymbol{Y})f := \boldsymbol{X}f - \boldsymbol{Y}f$
	$\boldsymbol{X}\boldsymbol{Y}f := \boldsymbol{X}(\boldsymbol{Y}f) = \boldsymbol{Y}(\boldsymbol{X}f)$
distribution	$\boldsymbol{X}(f+g) = \boldsymbol{X}f + \boldsymbol{X}g$

5.2 Annihilators

An ***annihilator*** of a function f is any nontrivial operator that transforms f into the zero function. (We can trivially annihilate any function by multiplying it by zero, so as a technical matter, we do not consider the zero operator to be an annihilator.) Every compound operator we consider annihilates a specific class of functions; conversely, every function composed of polynomial and exponential functions has a unique (minimal) annihilator.

We have already seen that the operator $(\boldsymbol{E}-2)$ annihilates the function 2^n. It's not hard to see that the operator $(\boldsymbol{E}-c)$ annihilates the function $\alpha \cdot c^n$, for any constants c and α. More generally, the operator $(\boldsymbol{E}-c)$ annihilates the function a^n if and only if $c=a$:

$$(\boldsymbol{E}-c)a^n = \boldsymbol{E}a^n - c\cdot a^n = a^{n+1} - c\cdot a^n = (a-c)a^n.$$

Thus, $(\boldsymbol{E}-2)$ is essentially the *only* annihilator of the function 2^n.

What about the function 2^n+3^n? The operator $(\boldsymbol{E}-2)$ annihilates the function 2^n, but leaves the function 3^n unchanged. Similarly, $(\boldsymbol{E}-3)$ annihilates 3^n while *negating* the function 2^n. But if we apply *both* operators, we annihilate both terms:

$$\begin{aligned}(\boldsymbol{E}-2)(2^n+3^n) &= \boldsymbol{E}(2^n+3^n) - 2(2^n+3^n)\\ &= (2^{n+1}+3^{n+1}) - (2^{n+1}+2\cdot 3^n) = 3^n\\ \implies (\boldsymbol{E}-3)(\boldsymbol{E}-2)(2^n+3^n) &= (\boldsymbol{E}-3)3^n = 0\end{aligned}$$

In general, for any integers $a \neq b$, the operator $(\boldsymbol{E}-a)(\boldsymbol{E}-b) = (\boldsymbol{E}-b)(\boldsymbol{E}-a) = (\boldsymbol{E}^2-(a+b)\boldsymbol{E}+ab)$ annihilates any function of the form $\alpha a^n + \beta b^n$, but nothing else.

What about the operator $(\boldsymbol{E}-a)(\boldsymbol{E}-a) = (\boldsymbol{E}-a)^2$? It turns out that this operator annihilates all functions of the form $(\alpha n+\beta)a^n$:

$$\begin{aligned}(\boldsymbol{E}-a)((\alpha n+\beta)a^n) &= (\alpha(n+1)+\beta)a^{n+1} - a(\alpha n+\beta)a^n\\ &= \alpha a^{n+1}\\ \implies (\boldsymbol{E}-a)^2((\alpha n+\beta)a^n) &= (\boldsymbol{E}-a)(\alpha a^{n+1}) = 0\end{aligned}$$

More generally, the operator $(\boldsymbol{E}-a)^d$ annihilates all functions of the form $p(n)\cdot a^n$, where $p(n)$ is a polynomial of degree at most $d-1$. For example, $(\boldsymbol{E}-1)^3$ annihilates any polynomial of degree at most 2.

The following table summarizes everything we need to remember about annihilators.

Operator	Functions annihilated	
$\boldsymbol{E}-1$	α	
$\boldsymbol{E}-a$	αa^n	
$(\boldsymbol{E}-a)(\boldsymbol{E}-b)$	$\alpha a^n + \beta b^n$	[if $a \neq b$]
$(\boldsymbol{E}-a_0)(\boldsymbol{E}-a_1)\cdots(\boldsymbol{E}-a_k)$	$\sum_{i=0}^{k} \alpha_i a_i^n$	[if a_i distinct]
$(\boldsymbol{E}-1)^2$	$\alpha n+\beta$	
$(\boldsymbol{E}-a)^2$	$(\alpha n+\beta)a^n$	
$(\boldsymbol{E}-a)^2(\boldsymbol{E}-b)$	$(\alpha n+\beta)a^b + \gamma b^n$	[if $a \neq b$]
$(\boldsymbol{E}-a)^d$	$\left(\sum_{i=0}^{d-1} \alpha_i n^i\right)a^n$	

If $\boldsymbol{X}$ annihilates f, then $\boldsymbol{X}$ also annihilates $\boldsymbol{E}f$.
If $\boldsymbol{X}$ annihilates both f and g, then $\boldsymbol{X}$ also annihilates $f \pm g$.
If $\boldsymbol{X}$ annihilates f, then $\boldsymbol{X}$ also annihilates αf, for any constant α.

If $\boldsymbol{X}$ annihilates f and $\boldsymbol{Y}$ annihilates g, then $\boldsymbol{XY}$ annihilates $f \pm g$.

5.3 Annihilating Recurrences

Given a linear recurrence for a function, it's easy to extract an annihilator for that function. For many recurrences, we only need to rewrite the recurrence in operator notation. Once we have an annihilator, we can factor it into operators of the form $(\boldsymbol{E} - c)$; the table on the previous page then gives us a generic solution with some unknown coefficients. If we are given explicit base cases, we can determine the coefficients by examining a few small cases; in general, this involves solving a small system of linear equations. If the base cases are not specified, the generic solution almost always gives us an asymptotic solution. Here is the technique step by step:

1. Write the recurrence in operator form
2. Extract an annihilator for the recurrence
3. Factor the annihilator (if necessary)
4. Extract the *generic solution* from the annihilator
5. Solve for coefficients using base cases (if known)

Here are several examples of the technique in action:

- ***$r(n) = 5r(n-1)$, where $r(0) = 3$.***

  1. We can write the recurrence in operator form as follows:
  $$r(n) = 5r(n-1) \implies r(n+1) - 5r(n) = 0 \implies (\boldsymbol{E}-5)r(n) = 0.$$
  2. We immediately see that $(\boldsymbol{E}-5)$ annihilates the function $r(n)$.
  3. The annihilator $(\boldsymbol{E}-5)$ is already factored.
  4. Consulting the annihilator table on the previous page, we find the generic solution $r(n) = \alpha 5^n$ for some constant α.
  5. The base case $r(0) = 3$ implies that $\alpha = 3$.

  We conclude that $\boldsymbol{r(n) = 3 \cdot 5^n}$. We can easily verify this closed-form solution by induction:
  $$r(0) = 3 \cdot 5^0 = 3 \quad \checkmark \qquad \text{[definition]}$$
  $$\begin{aligned} r(n) &= 5r(n-1) && \text{[definition]}\\ &= 5 \cdot (3 \cdot 5^{n-1}) && \text{[induction hypothesis]}\\ &= 5^n \cdot 3 \quad \checkmark && \text{[algebra]} \end{aligned}$$

- **Fibonacci numbers: $F(n) = F(n-1) + F(n-2)$, where $F(0) = 0$ and $F(1) = 1$.**

  1. We can rewrite the recurrence as $(\boldsymbol{E}^2 - \boldsymbol{E} - 1)F(n) = 0$.
  2. The operator $\boldsymbol{E}^2 - \boldsymbol{E} - 1$ clearly annihilates $F(n)$.
  3. The quadratic formula implies that the annihilator $\boldsymbol{E}^2 - \boldsymbol{E} - 1$ factors into $(\boldsymbol{E}-\phi)(\boldsymbol{E}-\hat{\phi})$, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$ is the golden ratio and $\hat{\phi} = (1-\sqrt{5})/2 = 1-\phi = -1/\phi \approx -0.618034$.
  4. The annihilator implies that $\boldsymbol{F(n) = \alpha\phi^n + \hat{\alpha}\hat{\phi}^n}$ for some unknown constants α and $\hat{\alpha}$.

5. The base cases give us two equations in two unknowns:

$$F(0) = 0 = \alpha + \hat{\alpha}$$
$$F(1) = 1 = \alpha\phi + \hat{\alpha}\hat{\phi}$$

Solving this system of equations gives us $\alpha = 1/(2\phi - 1) = 1/\sqrt{5}$ and $\hat{\alpha} = -1/\sqrt{5}$.

We conclude with the following exact closed form for the nth Fibonacci number:

$$\boldsymbol{F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n}$$

With all the square roots in this formula, it's quite amazing that Fibonacci numbers are integers. However, if we do all the math correctly, all the square roots cancel out when i is an integer. (In fact, this is pretty easy to prove using the binomial theorem.)

- **Towers of Hanoi: $T(n) = 2T(n-1) + 1$, where $T(0) = 0$.** This is our first example of a *non-homogeneous* recurrence, which means the recurrence has one or more non-recursive terms.

  1. We can rewrite the recurrence as $(\boldsymbol{E} - 2)T(n) = 1$.
  2. The operator $(\boldsymbol{E}-2)$ doesn't quite annihilate the function; it leaves a *residue* of 1. But we can annihilate the residue by applying the operator $(\boldsymbol{E} - 1)$. Thus, the compound operator $(\boldsymbol{E} - 1)(\boldsymbol{E} - 2)$ annihilates the function.
  3. The annihilator is already factored.
  4. The annihilator table gives us the generic solution $T(n) = \alpha 2^n + \beta$ for some unknown constants α and β.
  5. The base cases give us $T(0) = 0 = \alpha 2^0 + \beta$ and $T(1) = 1 = \alpha 2^1 + \beta$. Solving this system of equations, we find that $\alpha = 1$ and $\beta = -1$.

  We conclude that $\boldsymbol{T(n) = 2^n - 1}$.

For the remaining examples, I won't explicitly enumerate the steps in the solution.

- **Height-balanced trees: $H(n) = H(n-1) + H(n-2) + 1$, where $H(-1) = 0$ and $H(0) = 1$.** (Yes, we're starting at -1 instead of 0. So what?)

  We can rewrite the recurrence as $(\boldsymbol{E}^2 - \boldsymbol{E} - 1)H = 1$. The residue 1 is annihilated by $(\boldsymbol{E} - 1)$, so the compound operator $(\boldsymbol{E} - 1)(\boldsymbol{E}^2 - \boldsymbol{E} - 1)$ annihilates the recurrence. This operator factors into $(\boldsymbol{E} - 1)(\boldsymbol{E} - \phi)(\boldsymbol{E} - \hat{\phi})$, where $\phi = (1+\sqrt{5})/2$ and $\hat{\phi} = (1-\sqrt{5})/2$. Thus, we get the generic solution $H(n) = \alpha \cdot \phi^n + \beta + \gamma \cdot \hat{\phi}^n$, for some unknown constants α, β, γ that satisfy the following system of equations:

$$\begin{aligned} H(-1) = 0 &= \alpha\phi^{-1} + \beta + \gamma\hat{\phi}^{-1} = \alpha/\phi + \beta - \gamma/\hat{\phi} \\ H(0) = 1 &= \alpha\phi^0 + \beta + \gamma\hat{\phi}^0 \quad = \alpha + \beta + \gamma \\ H(1) = 2 &= \alpha\phi^1 + \beta + \gamma\hat{\phi}^1 \quad = \alpha\phi + \beta + \gamma\hat{\phi} \end{aligned}$$

  Solving this system (using Cramer's rule or Gaussian elimination), we find that $\alpha = (\sqrt{5}+2)/\sqrt{5}$, $\beta = -1$, and $\gamma = (\sqrt{5}-2)/\sqrt{5}$. We conclude that

$$\boldsymbol{H(n) = \frac{\sqrt{5}+2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - 1 + \frac{\sqrt{5}-2}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n}.$$

- $\boldsymbol{T(n) = 3T(n-1) - 8T(n-2) + 4T(n-3)}$**, where** $\boldsymbol{T(0) = 1}$**,** $\boldsymbol{T(1) = 0}$**, and** $\boldsymbol{T(2) = 0}$**.** This was our original example of a linear recurrence.

  We can rewrite the recurrence as $(E^3 - 3E^2 + 8E - 4)T = 0$, so we immediately have an annihilator $E^3 - 3E^2 + 8E - 4$. Using high-school algebra, we can factor the annihilator into $(\boldsymbol{E}-2)^2(\boldsymbol{E}-1)$, which implies the generic solution $T(n) = \alpha n 2^n + \beta 2^n + \gamma$. The constants α, β, and γ are determined by the base cases:

$$\begin{aligned} T(0) &= 1 = \alpha \cdot 0 \cdot 2^0 + \beta 2^0 + \gamma = \beta + \gamma \\ T(1) &= 0 = \alpha \cdot 1 \cdot 2^1 + \beta 2^1 + \gamma = 2\alpha + 2\beta + \gamma \\ T(2) &= 0 = \alpha \cdot 2 \cdot 2^2 + \beta 2^2 + \gamma = 8\alpha + 4\beta + \gamma \end{aligned}$$

  Solving this system of equations, we find that $\alpha = 1$, $\beta = -3$, and $\gamma = 4$, so $\boldsymbol{T(n) = (n-3)2^n + 4}$.

- $\boldsymbol{T(n) = T(n-1) + 2T(n-2) + 2^n - n^2}$

  We can rewrite the recurrence as $(\boldsymbol{E}^2 - \boldsymbol{E} - 2)T(n) = \boldsymbol{E}^2(2^n - n^2)$. Notice that we had to shift up the non-recursive parts of the recurrence when we expressed it in this form. The operator $(\boldsymbol{E}-2)(\boldsymbol{E}-1)^3$ annihilates the residue $2^n - n^2$, and therefore also annihilates the shifted residue $\boldsymbol{E}^2(2^n + n^2)$. Thus, the operator $(\boldsymbol{E}-2)(\boldsymbol{E}-1)^3(\boldsymbol{E}^2 - \boldsymbol{E} - 2)$ annihilates the entire recurrence. We can factor the quadratic factor into $(\boldsymbol{E}-2)(E+1)$, so the annihilator factors into $(\boldsymbol{E}-2)^2(\boldsymbol{E}-1)^3(\boldsymbol{E}+1)$. So the generic solution is $\boldsymbol{T(n) = \alpha n 2^n + \beta 2^n + \gamma n^2 + \delta n + \varepsilon + \eta(-1)^n}$. The coefficients α, β, γ, δ, ε, η satisfy a system of six equations determined by the first six function values $T(0)$ through $T(5)$. For almost[2] every set of base cases, we have $\alpha \neq 0$, which implies that $\boldsymbol{T(n) = \Theta(n2^n)}$.

For a more detailed explanation of the annihilator method, see George Lueker, Some techniques for solving recurrences, *ACM Computing Surveys* 12(4):419-436, 1980.

6 Transformations

Sometimes we encounter recurrences that don't fit the structures required for recursion trees or annihilators. In many of those cases, we can transform the recurrence into a more familiar form, by defining a new function in terms of the one we want to solve. There are many different kinds of transformations, but these three are probably the most useful:

- **Domain transformation:** Define a new function $S(n) = T(f(n))$ with a simpler recurrence, for some simple function f.

- **Range transformation:** Define a new function $S(n) = f(T(n))$ with a simpler recurrence, for some simple function f.

- **Difference transformation:** Simplify the recurrence for $T(n)$ by considering the difference function $\Delta T(n) = T(n) - T(n-1)$.

Here are some examples of these transformations in action.

[2]In fact, the only possible solutions with $\alpha = 0$ have the form $-2^{n-1} - n^2/2 - 5n/2 + \eta(-1)^n$ for some constant η.

- **Unsimplified Mergesort: $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)$**

  When n is a power of 2, we can simplify the mergesort recurrence to $T(n) = 2T(n/2) + \Theta(n)$, which has the solution $T(n) = \Theta(n \log n)$. Unfortunately, for other values values of n, this simplified recurrence is incorrect. When n is odd, then the recurrence calls for us to sort a fractional number of elements! Worse yet, if n is not a power of 2, we will *never* reach the base case $T(1) = 1$.

  So we really need to solve the original recurrence. We have no hope of getting an *exact* solution, even if we ignore the $\Theta(\,)$ in the recurrence; the floors and ceilings will eventually kill us. But we can derive a tight asymptotic solution using a domain transformation—we can rewrite the function $T(n)$ as a nested function $S(f(n))$, where $f(n)$ is a simple function and the function $S(\,)$ has an simpler recurrence.

  First let's overestimate the time bound, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

  $$T(n) \le 2T\big(\lceil n/2 \rceil\big) + n \le 2T(n/2 + 1) + n.$$

  Now we define a new function $S(n) = T(n + \alpha)$, where α is a unknown constant, chosen so that $S(n)$ satisfies the Master-Theorem-ready recurrence $S(n) \le 2S(n/2) + O(n)$. To figure out the correct value of α, we compare two versions of the recurrence for the function $T(n + \alpha)$:

  $$\begin{aligned} S(n) \le 2S(n/2) + O(n) \quad &\Longrightarrow \quad T(n+\alpha) \le 2T(n/2+\alpha) + O(n) \\ T(n) \le 2T(n/2+1) + n \quad &\Longrightarrow \quad T(n+\alpha) \le 2T((n+\alpha)/2+1) + n + \alpha \end{aligned}$$

  For these two recurrences to be equal, we need $n/2 + \alpha = (n + \alpha)/2 + 1$, which implies that $\alpha = 2$. The Master Theorem now tells us that $S(n) = O(n \log n)$, so

  $$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n).$$

  A similar argument implies the matching lower bound $T(n) = \Omega(n \log n)$. So $\boldsymbol{T(n) = \Theta(n \log n)}$ after all, just as though we had ignored the floors and ceilings from the beginning!

  Domain transformations are useful for removing floors, ceilings, and lower order terms from the arguments of any recurrence that otherwise looks like it ought to fit either the Master Theorem or the recursion tree method. But now that we know this, we don't need to bother grinding through the actual gory details!

- **Ham-Sandwich Trees: $T(n) = T(n/2) + T(n/4) + 1$**

  As we saw earlier, the recursion tree method only gives us the uselessly loose bounds $\sqrt{n} \ll T(n) \ll n$ for this recurrence, and the recurrence is in the wrong form for annihilators. The authors who discovered ham-sandwich trees (Yes, this is a real data structure!) solved this recurrence by guessing the solution and giving a complicated induction proof. We got a tight solution using the Akra-Bazzi method, but who can remember that?

  In fact, a simple domain transformation allows us to solve the recurrence in just a few lines. We define a new function $t(k) = T(2^k)$, which satisfies the simpler linear recurrence $t(k) = t(k-1) + t(k-2) + 1$. This recurrence should immediately remind you of Fibonacci numbers. Sure enough, the annihilator method implies the solution $t(k) = \Theta(\phi^k)$, where $\phi = (1+\sqrt{5})/2$ is the golden ratio. We conclude that

$$T(n) = t(\lg n) = \Theta(\phi^{\lg n}) = \boldsymbol{\Theta(n^{\lg \phi})} \approx \Theta(n^{0.69424}).$$

This is the same solution we obtained earlier using the Akra-Bazzi theorem.

Many other divide-and-conquer recurrences can be similarly transformed into linear recurrences and then solved with annihilators. Consider once more the simplified mergesort recurrence $T(n) = 2T(n/2) + n$. The function $t(k) = T(2^k)$ satisfies the recurrence $t(k) = 2t(k-1) + 2^k$. The annihilator method gives us the generic solution $t(k) = \Theta(k \cdot 2^k)$, which implies that $T(n) = t(\lg n) = \Theta(n \log n)$, just as we expected.

On the other hand, for some recurrences like $T(n) = T(n/3) + T(2n/3) + n$, the recursion tree method gives an easy solution, but there's no way to transform the recurrence into a form where we can apply the annihilator method directly.[3]

- **Random Binary Search Trees:** $\boldsymbol{T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1}$

  This looks like a divide-and-conquer recurrence, so we might be tempted to apply recursion trees, but what does it mean to have a quarter of a child? If we're not comfortable with weighted recursion trees (or the Akra-Bazzi theorem), we can instead apply the following range transformation. The function $U(n) = n \cdot T(n)$ satisfies the more palatable recurrence $U(n) = U(n/4) + U(3n/4) + n$. As we've already seen, recursion trees imply that $U(n) = \Theta(n \log n)$, which immediately implies that $\boldsymbol{T(n) = \Theta(\log n)}$.

- **Randomized Quicksort:** $\boldsymbol{T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + n}$

  This is our first example of a ***full history*** recurrence; each function value $T(n)$ is defined in terms of *all* previous function values $T(k)$ with $k < n$. Before we can apply any of our existing techniques, we need to convert this recurrence into an equivalent *limited history* form by shifting and subtracting away common terms. To make this step slightly easier, we first multiply both sides of the recurrence by n to get rid of the fractions.

$$\begin{aligned}
n \cdot T(n) &= 2\sum_{k=0}^{n-1} T(j) + n^2 && \text{[multiply both sides by } n\text{]}\\
(n-1)\cdot T(n-1) &= 2\sum_{k=0}^{n-2} T(j) + (n-1)^2 && \text{[shift]}\\
nT(n) - (n-1)T(n-1) &= 2T(n-1) + 2n - 1 && \text{[subtract]}\\
T(n) &= \frac{n+1}{n}T(n-1) + 2 - \frac{1}{n} && \text{[simplify]}
\end{aligned}$$

[3]However, we can still get a solution via functional transformations as follows. The function $t(k) = T((3/2)^k)$ satisfies the recurrence $t(n) = t(n-1) + t(n-\lambda) + (3/2)^k$, where $\lambda = \log_{3/2} 3 = 2.709511\ldots$. The ***characteristic function*** for this recurrence is $(r^\lambda - r^{\lambda-1} - 1)(r - 3/2)$, which has a double root at $r = 3/2$ and nowhere else. Thus, $t(k) = \Theta(k(3/2)^k)$, which implies that $T(n) = t(\log_{3/2} n) = \Theta(n \log n)$. This line of reasoning is the core of the Akra-Bazzi method.

We can solve this limited-history recurrence using another functional transformation. We define a new function $t(n) = T(n)/(n+1)$, which satisfies the simpler recurrence

$$t(n) = t(n-1) + \frac{2}{n+1} - \frac{1}{n(n+1)},$$

which we can easily unroll into a summation. If we only want an asymptotic solution, we can simplify the final recurrence to $t(n) = t(n-1) + \Theta(1/n)$, which unrolls into a very familiar summation:

$$t(n) = \sum_{i=1}^{n} \Theta(1/i) = \Theta(H_n) = \Theta(\log n).$$

Finally, substituting $T(n) = (n+1)t(n)$ gives us a solution to the original recurrence: $\boldsymbol{T(n) = \Theta(n \log n)}$.

Exercises

1. For each of the following recurrences, first ***guess*** an exact closed-form solution, and then prove your guess is correct. You are free to use any method you want to make your guess—unrolling the recurrence, writing out the first several values, induction proof template, recursion trees, annihilators, transformations, 'It looks like that other one', whatever—but please describe your method. All functions are from the non-negative integers to the reals. If it simplifies your solutions, express them in terms of Fibonacci numbers F_n, harmonic numbers H_n, binomial coefficients $\binom{n}{k}$, factorials $n!$, and/or the floor and ceiling functions $\lfloor x \rfloor$ and $\lceil x \rceil$.

   (a) $A(n) = A(n-1) + 1$, where $A(0) = 0$.

   (b) $B(n) = \begin{cases} 0 & \text{if } n < 5 \\ B(n-5) + 2 & \text{otherwise} \end{cases}$

   (c) $C(n) = C(n-1) + 2n - 1$, where $C(0) = 0$.

   (d) $D(n) = D(n-1) + \binom{n}{2}$, where $D(0) = 0$.

   (e) $E(n) = E(n-1) + 2^n$, where $E(0) = 0$.

   (f) $F(n) = 3 \cdot F(n-1)$, where $F(0) = 1$.

   (g) $G(n) = \frac{G(n-1)}{G(n-2)}$, where $G(0) = 1$ and $G(1) = 2$. *[Hint: This is easier than it looks.]*

   (h) $H(n) = H(n-1) + 1/n$, where $H(0) = 0$.

   (i) $I(n) = I(n-2) + 3/n$, where $I(0) = I(1) = 0$. *[Hint: Consider even and odd n separately.]*

   (j) $J(n) = J(n-1)^2$, where $J(0) = 2$.

   (k) $K(n) = K(\lfloor n/2 \rfloor) + 1$, where $K(0) = 0$.

   (l) $L(n) = L(n-1) + L(n-2)$, where $L(0) = 2$ and $L(1) = 1$.
   [Hint: Write the solution in terms of Fibonacci numbers.]

   (m) $M(n) = M(n-1) \cdot M(n-2)$, where $M(0) = 2$ and $M(1) = 1$.
   [Hint: Write the solution in terms of Fibonacci numbers.]

(n) $N(n) = 1 + \sum_{k=1}^{n} (N(k-1) + N(n-k))$, where $N(0) = 1$.

(p) $P(n) = \sum_{k=0}^{n-1} (k \cdot P(k-1))$, where $P(0) = 1$.

(q) $Q(n) = \frac{1}{2-Q(n-1)}$, where $Q(0) = 0$.

(r) $R(n) = \max_{1 \le k \le n} \{R(k-1) + R(n-k) + n\}$

(s) $S(n) = \max_{1 \le k \le n} \{S(k-1) + S(n-k) + 1\}$

(t) $T(n) = \min_{1 \le k \le n} \{T(k-1) + T(n-k) + n\}$

(u) $U(n) = \min_{1 \le k \le n} \{U(k-1) + U(n-k) + 1\}$

(v) $V(n) = \max_{n/3 \le k \le 2n/3} \{V(k-1) + V(n-k) + n\}$

2. Use recursion trees or the Akra-Bazzi theorem to solve each of the following recurrences.

(a) $A(n) = 2A(n/4) + \sqrt{n}$

(b) $B(n) = 2B(n/4) + n$

(c) $C(n) = 2C(n/4) + n^2$

(d) $D(n) = 3D(n/3) + \sqrt{n}$

(e) $E(n) = 3E(n/3) + n$

(f) $F(n) = 3F(n/3) + n^2$

(g) $G(n) = 4G(n/2) + \sqrt{n}$

(h) $H(n) = 4H(n/2) + n$

(i) $I(n) = 4I(n/2) + n^2$

(j) $J(n) = J(n/2) + J(n/3) + J(n/6) + n$

(k) $K(n) = K(n/2) + K(n/3) + K(n/6) + n^2$

(l) $L(n) = L(n/15) + L(n/10) + 2L(n/6) + \sqrt{n}$

*(m) $M(n) = 2M(n/3) + 2M(2n/3) + n$

(n) $N(n) = \sqrt{2n}\, N(\sqrt{2n}) + \sqrt{n}$

(p) $P(n) = \sqrt{2n}\, P(\sqrt{2n}) + n$

(q) $Q(n) = \sqrt{2n}\, Q(\sqrt{2n}) + n^2$

(r) $R(n) = R(n-3) + 8^n$ — Don't use annihilators!

(s) $S(n) = 2S(n-2) + 4^n$ — Don't use annihilators!

(t) $T(n) = 4T(n-1) + 2^n$ — Don't use annihilators!

3. Make up a bunch of linear recurrences and then solve them using annihilators.

4. Solve the following recurrences, using any tricks at your disposal.

   (a) $T(n) = \sum_{i=1}^{\lg n} T(n/2^i) + n$ *[Hint: Assume n is a power of 2.]*

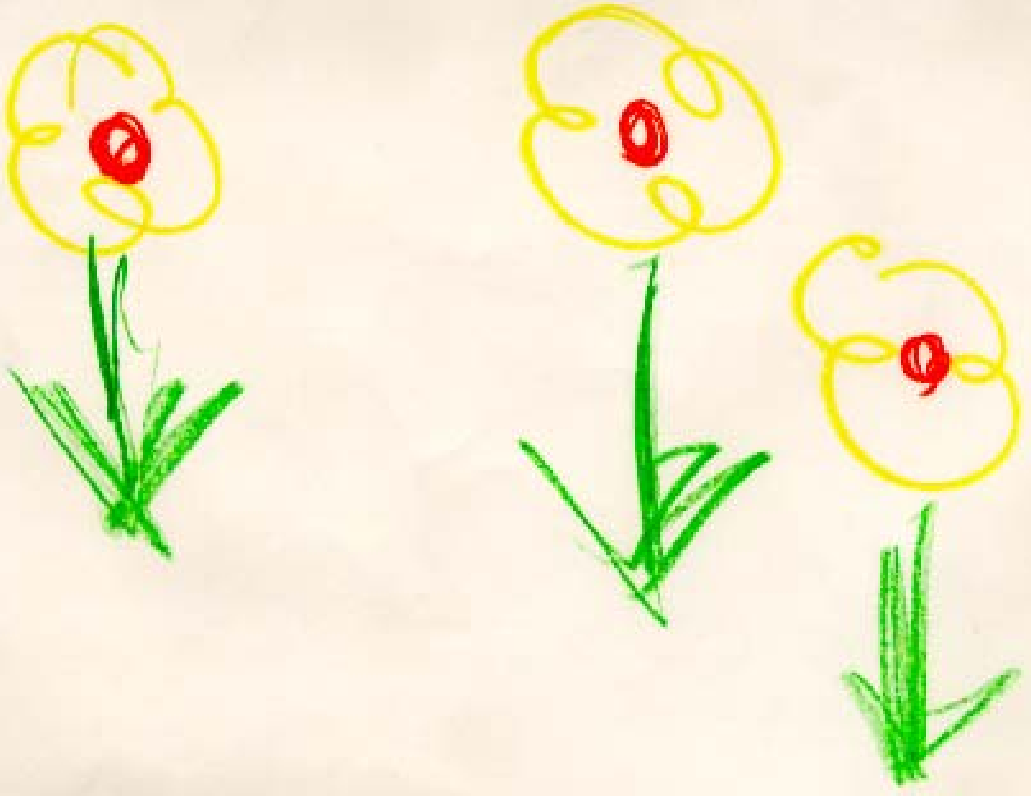
   (b) More to come...

CS 373

Name: Johnny

alias: pikachu

HW1



1

Trees have no cycles but must be connected.
Tourn'ments are cliques with their edges directed.
Hamilton circuits, Eulerian paths,
These are a few of my favorite graphs.

Matchings and bicliques and blossoms and bases,
Kempe chains, hypercubes, forests, and faces,
AKS networks that split into halves,
These are a few of my favorite graphs.

Short paths of co-authors leading to Erdos,
Large neural networks that translate from Kurdish,
Finite projective planes - they make me laugh.
These are a few of my favorite graphs.

Chorus: Propositions, corollaries,
Problems that are starred,
I simply remember my favorite graphs
And then they don't seem so hard.

If there's no K_5 or K_{3,3} minor,
Old Kuratowski says it'll be 'plinor'.
Four's enough colors if there are no gaffes!
These are a few of my favorite graphs.

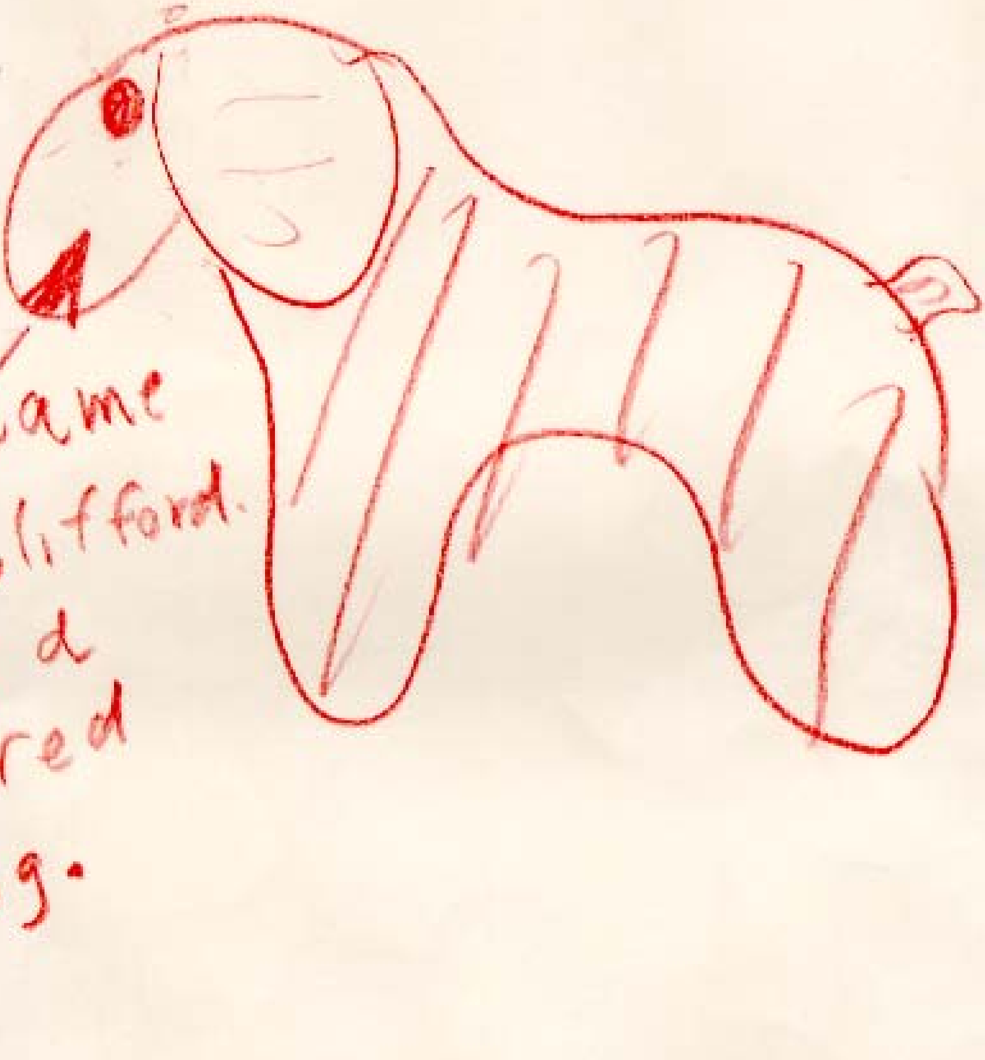
Quadrangles, thrackles, and triangulations,
Minor-closed families and sparsifications,
Voronoi diagrams found on giraffes,
These are a few of my favorite graphs.

Chorus: Propositions, corollaries,
When they're just too deep,
I simply remember my favorite graphs
And then I go right to sleep.

Look! I minimized the slop of my favourite graph song!

2.

Mamma always said:
2tupid is as stupid does.



My name
is Clifford.
I'm a
big red
dog.

3. This problem is the work of Satan. It is very scary. I was too afraid to do it. Sorry.

4. I betcha that:

∴

QED.

5. I am running this algorithm with MGD. Will report results...

much beer drinking

more analysis needed

...

nlgn is like a cheese log

Im hungry, call it

[illegible]

this is funny

beer.

CS 373: Combinatorial Algorithms, Spring 1999

http://www-courses.cs.uiuc.edu/ cs373

Homework 0 (due January 26, 1999 by the beginning of class)

Name:	
Net ID:	Alias:

Neatly print your name (first name first, with no comma), your network ID, and a short alias into the boxes above. Do not *sign* your name. Do not write your Social Security number. Staple this sheet of paper to the top of your homework. Grades will be listed on the course web site by alias, so your alias should not resemble your name (or your Net ID). If you do not give yourself an alias, you will be stuck with one we give you, no matter how much you hate it.

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

This homework tests your familiarity with the prerequisite material from CS 225 and CS 273 (and *their* prerequisites)—many of these problems appeared on homeworks and/or exams in those classes—primarily to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.**

Undergrad/.75U Grad/1U Grad Problems

▶1. **[173/273]**

(a) Prove that any positive integer can be written as the sum of distinct powers of 2. (For example: $42 = 2^5 + 2^3 + 2^1$, $25 = 2^4 + 2^3 + 2^0$, $17 = 2^4 + 2^0$.)

(b) Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. (For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.)

(c) Prove that *any* integer can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. (For example: $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.)

▶2. **[225/273]** Sort the following functions from asymptotically smallest to largest, indicating ties if there are any: n, $\lg n$, $\lg\lg^* n$, $\lg^* \lg n$, $\lg^* n$, $n \lg n$, $\lg(n \lg n)$, $n^{n/\lg n}$, $n^{\lg n}$, $(\lg n)^n$, $(\lg n)^{\lg n}$, $2^{\sqrt{\lg n \lg\lg n}}$, 2^n, $n^{\lg\lg n}$, $\sqrt[1000]{n}$, $(1 + \frac{1}{1000})^n$, $(1 - \frac{1}{1000})^n$, $\lg^{1000} n$, $\lg^{(1000)} n$, $\log_{1000} n$, $\lg^n 1000$, 1.

[To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2, n, $\binom{n}{2}$, n^3 could be sorted as follows: $n \ll n^2 \equiv \binom{n}{2} \ll n^3$.]

3. **[273/225]** Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable (nontrivial) base cases. Extra credit will be given for more exact solutions.

   ►(a) $A(n) = A(n/2) + n$

   (b) $B(n) = 2B(n/2) + n$

   ►(c) $C(n) = 3C(n/2) + n$

   (d) $D(n) = \max\limits_{n/3<k<2n/3} \big(D(k) + D(n-k) + n\big)$

   ►(e) $E(n) = \min\limits_{0<k<n} \big(E(k) + E(n-k) + 1\big)$

   (f) $F(n) = 4F(\lfloor n/2 \rfloor + 5) + n$

   ►(g) $G(n) = G(n-1) + 1/n$

   ⋆(h) $H(n) = H(n/2) + H(n/4) + H(n/6) + H(n/12) + n$ [Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.]

   ►⋆(i) $I(n) = 2I(n/2) + n/\lg n$

   ⋆(j) $J(n) = \dfrac{J(n-1)}{J(n-2)}$

►4. **[273]** Alice and Bob each have a fair n-sided die. Alice rolls her die once. Bob then repeatedly throws his die until the number he rolls is at least as big as the number Alice rolled. Each time Bob rolls, he pays Alice $\$1$. (For example, if Alice rolls a 5, and Bob rolls a 4, then a 3, then a 1, then a 5, the game ends and Alice gets $\$4$. If Alice rolls a 1, then no matter what Bob rolls, the game will end immediately, and Alice will get $\$1$.)

   Exactly how much money does Alice expect to win at this game? Prove that your answer is correct. (If you have to appeal to "intuition" or "common sense", your answer is probably wrong.)

►5. **[225]** George has a 26-node binary tree, with each node labeled by a unique letter of the alphabet. The preorder and postorder sequences of nodes are as follows:

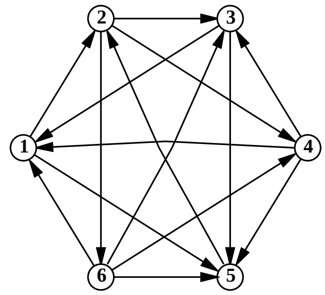
   preorder: `M N H C R S K W T G D X I Y A J P O E Z V B U L Q F`
   postorder: `C W T K S G R H D N A O E P J Y Z I B Q L F U V X M`

   Draw George's binary tree.

Only 1U Grad Problems

▷⋆1. **[225/273]** A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once.

   Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \to 4 \to 5 \to 2 \to 3 \to 1$.

Practice Problems

1. **[173/273]** Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m.

   (a) F_n is even if and only if n is divisible by 3.

   (b) $\sum_{i=0}^{n} F_i = F_{n+2} - 1$

   (c) $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$

   ★(d) If n is an integer multiple of m, then F_n is an integer multiple of F_m.

2. **[225/273]**

   (a) Prove that $2^{\lceil \lg n \rceil + \lfloor \lg n \rfloor}/n = \Theta(n)$.

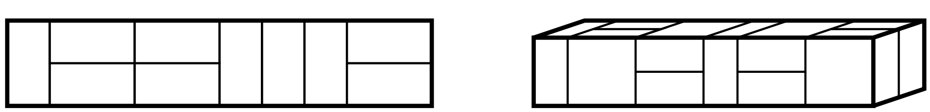
   (b) Is $2^{\lfloor \lg n \rfloor} = \Theta\big(2^{\lceil \lg n \rceil}\big)$? Justify your answer.

   (c) Is $2^{2^{\lfloor \lg \lg n \rfloor}} = \Theta\big(2^{2^{\lceil \lg \lg n \rceil}}\big)$? Justify your answer.

3. **[273]**

   (a) A *domino* is a 2×1 or 1×2 rectangle. How many different ways are there to completely fill a $2 \times n$ rectangle with n dominos?

   (b) A *slab* is a three-dimensional box with dimensions $1 \times 2 \times 2$, $2 \times 1 \times 2$, or $2 \times 2 \times 1$. How many different ways are there to fill a $2 \times 2 \times n$ box with n slabs? Set up a recurrence relation and give an *exact* closed-form solution.



A 2×10 rectangle filled with ten dominos, and a $2 \times 2 \times 10$ box filled with ten slabs.

4. **[273]** Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ..., 52 of clubs. (They're big cards.) Penn shuffles the deck until each each of the 52! possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the three of clubs.

(a) On average, how many cards does Penn give Teller?

(b) On average, what is the smallest-numbered card that Penn gives Teller?

*(c) On average, what is the largest-numbered card that Penn gives Teller?

Prove that your answers are correct. (If you have to appeal to "intuition" or "common sense", your answers are probably wrong.) [Hint: Solve for an n-card deck, and then set n to 52.]

5. **[273/225]** Prove that for any nonnegative parameters a and b, the following algorithms terminate and produce identical output.

```
SLOWEUCLID(a, b) :
  if b > a
      return SLOWEUCLID(b, a)
  else if b == 0
      return a
  else
      return SLOWEUCLID(a, b − a)
```

```
FASTEUCLID(a, b) :
  if b == 0
      return a
  else
      return FASTEUCLID(b, a mod b)
```

CS 373: Combinatorial Algorithms, Spring 1999

`http://www-courses.cs.uiuc.edu/~cs373`

Homework 1 (due February 9, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
2. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
3. Justify the correctness of your algorithm, including termination if that is not obvious.
4. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

▶1. Consider the following sorting algorithm:

```
StupidSort(A[0 .. n − 1]) :
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m = ⌈2n/3⌉
      StupidSort(A[0 .. m − 1])
      StupidSort(A[n − m .. n − 1])
      StupidSort(A[0 .. m − 1])
```

(a) Prove that StupidSort actually sorts its input.

(b) Would the algorithm still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.

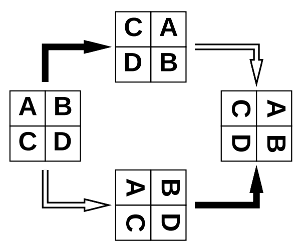
(c) State a recurrence (including the base case(s)) for the number of comparisons executed by StupidSort.

(d) Solve the recurrence, and prove that your solution is correct. [Hint: Ignore the ceiling.] Does the algorithm deserve its name?

*(e) Show that the number of *swaps* executed by STUPIDSORT is at most $\binom{n}{2}$.

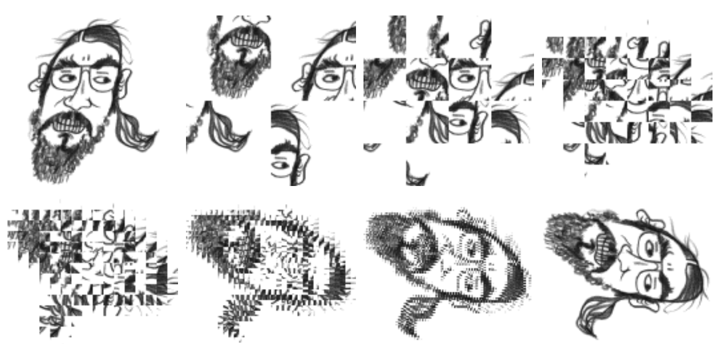
►2. Some graphics hardware includes support for an operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixelmap (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixelmap $90°$ clockwise. One way to do this is to split the pixelmap into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we can first recursively rotate the blocks and blit them into place afterwards.



Two algorithms for rotating a pixelmap.
Black arrows indicate blitting the blocks into place.
White arrows indicate recursively rotating the blocks.

The following sequence of pictures shows the first algorithm (blit then recurse) in action.



In the following questions, assume n is a power of two.

(a) Prove that both versions of the algorithm are correct.

(b) *Exactly* how many blits does the algorithm perform?

(c) What is the algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

(d) What if a $k \times k$ blit takes only $O(k)$ time?

►3. Dynamic Programming: The Company Party
A company is planning a party for its employees. The organizers of the party want it to be a fun party, and so have assigned a 'fun' rating to every employee. The employees are organized into a strict hierarchy, i.e. a tree rooted at the president. There is one restriction, though, on the guest list to the party: both an employee and their immediate supervisor (parent in the tree) cannot both attend the party (because that would be no fun at all). Give an algorithm that makes a guest list for the party that maximizes the sum of the 'fun' ratings of the guests.

►4. Dynamic Programming: Longest Increasing Subsequence (LIS)
Give an $O(n^2)$ algorithm to find the longest increasing subsequence of a sequence of numbers. Note: the elements of the subsequence need not be adjacent in the sequence. For example, the sequence $(1, 5, 3, 2, 4)$ has an LIS $(1, 3, 4)$.

►5. Nut/Bolt Median
You are given a set of n nuts and n bolts of different sizes. Each nut matches exactly one bolt (and vice versa, of course). The sizes of the nuts and bolts are so similar that you cannot compare two nuts or two bolts to see which is larger. You can, however, check whether a nut is too small, too large, or just right for a bolt (and vice versa, of course).

In this problem, your goal is to find the median bolt (*i.e.*, the $\lfloor n/2 \rfloor$th largest) as quickly as possible.

(a) Describe an efficient deterministic algorithm that finds the median bolt. How many nut-bolt comparisons does your algorithm perform in the worst case?

(b) Describe an efficient *randomized* algorithm that finds the median bolt.

  i. State a recurrence for the expected number of nut/bolt comparisons your algorithm performs.

  ii. What is the probability that your algorithm compares the ith largest bolt with the jth largest nut?

  iii. What is the expected number of nut-bolt comparisons made by your algorithm? [Hint: Use your answer to either of the previous two questions.]

Only 1U Grad Problems

▷1. You are at a political convention with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party a delegate belongs to. However, you can check whether any two delegates are in the *same* party or not by introducing them to each other. (Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.)

(a) Suppose a majority (more than half) of the delegates are from the same political party. Give an efficient algorithm that identifies a member of the majority party.

*(b) Suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Give an efficient algorithm that identifies a member of the plurality party.

*(c) Suppose you don't know how many parties there are, but you do know that one party has a plurality, and at least p people in the plurality party are present. Present a practical procedure to pick a person from the plurality party as parsimoniously as possible. (Please.)

$^\bigstar$(d) Finally, suppose you don't know how many parties are represented at the convention, and you don't know how big the plurality is. Give an efficient algorithm to identify a member of the plurality party. How is the running time of your algorithm affected by the number of parties (k)? By the size of the plurality (p)?

Practice Problems

1. Second Smallest
   Give an algorithm that finds the *second* smallest of n elements in at most $n + \lceil \lg n \rceil - 2$ comparisons. Hint: divide and conquer to find the smallest; where is the second smallest?

2. Linear in-place 0-1 sorting
   Suppose that you have an array of records whose keys to be sorted consist only of 0's and 1's. Give a simple, linear-time $O(n)$ algorithm to sort the array in place (using storage of no more than constant size in addition to that of the array).

3. Dynamic Programming: Coin Changing
   Consider the problem of making change for n cents using the least number of coins.

   (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

   (b) Suppose that the available coins have the values $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

   (c) Give a set of 4 coin values for which the greedy algorithm does not yield an optimal solution, show why.

   (d) Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of coin values.

   (e) Prove that, with only two coins a, b whose gcd is 1, the smallest value n for which change *can* be given for all values greater than or equal to n is $(a-1)(b-1)$.

   $^\bigstar$(f) For only three coins a, b, c whose gcd is 1, give an algorithm to determine the smallest value n for which change *can* be given for all values greater than n. (note: this problem is currently unsolved for $n > 4$.

4. Dynamic Programming: Paragraph Justification
Consider the problem of printing a paragraph neatly on a printer (with fixed width font). The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$. The line length is M (the maximum # of characters per line). We expect that the paragraph is left justified, that all first words on a line start at the leftmost position and that there is exactly one space between any two words on the same line. We want the uneven right ends of all the lines to be together as 'neat' as possible. Our criterion of neatness is that we wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of the lines. Note: if a printed line contains words i through j, then the number of spaces at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$.

   (a) Give a dynamic programming algorithm to do this.

   (b) Prove that if the neatness function is linear, a linear time greedy algorithm will give an optimum 'neatness'.

5. Comparison of Amortized Analysis Methods
A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

   (a) Aggregate method

   (b) Accounting method

   *(c) Potential method

CS 373: Combinatorial Algorithms, Spring 1999

http://www-courses.cs.uiuc.edu/~cs373

Homework 2 (due Thu. Feb. 18, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ►. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
2. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
3. Justify the correctness of your algorithm, including termination if that is not obvious.
4. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

►1. Faster Longest Increasing Subsequence (LIS)
Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. Hint: In the dynamic programming solution, you don't really have to look back at all previous items.

►2. SELECT(A, k)
Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.

(a) Show that a rotation in an augmented binary tree can be performed in constant time.

(b) Describe an algorithm SCAPEGOATSELECT(k) that selects the kth smallest item in an augmented scapegoat tree in $O(\log n)$ *worst-case* time. (The scapegoat trees presented in class were already augmented.)

(c) Describe an algorithm SPLAYSELECT(k) that selects the kth smallest item in an augmented splay tree in $O(\log n)$ *amortized* time.

(d) Describe an algorithm TREAPSELECT(k) that selects the kth smallest item in an augmented treap in $O(\log n)$ *expected* time.

►3. Scapegoat trees

(a) Prove that only one tree gets rebalanced at any insertion.

(b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree ($I(v) = max(0, |\hat{v}| - |\check{v}|)$, where $\hat{v}$ is the child of greater height and $\check{v}$ the child of lesser height, $|v|$ is the number of nodes in subtree v, and perfectly balanced means each subtree has as close to half the leaves as possible and is perfectly balanced itself.

*(c) Show that you can rebuild a fully balanced binary tree in $O(n)$ time using only $O(1)$ additional memory.

►4. Memory Management

Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than 1/4 full, we we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do not use the potential method (it makes it much more difficult).

Only 1U Grad Problems

▷1. Detecting overlap

(a) You are given a list of ranges represented by min and max (e.g. [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$-time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.

(b) You are given a list of rectangles represented by min and max x- and y- coordinates. Give an $O(n \log n)$-time algorithm that decides whether or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.

Practice Problems

1. Amortization

   (a) Modify the binary double-counter (see class notes Feb. 2) to support a new operation Sign, which determines whether the number being stored is positive, negative, or zero, in constant time. The amortized time to increment or decrement the counter should still be a constant.

   [Hint: Suppose p is the number of significant bits in P, and n is the number of significant bits in N. For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

   *(b) Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

*2. Amortization
Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of "fits", where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is *not* the same representation as on Homework 0.]

3. Rotations

   (a) Show that it is possible to transform any n-node binary search tree into any other n-node binary search tree using at most $2n - 2$ rotations.

   *(b) Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.

4. Fibonacci Heaps: SECONDMIN
We want to find the second smallest of a set efficiently.

   (a) Implement SECONDMIN by using a Fibonacci heap as a black box. Remember to justify its correctness and running time.

   *(b) Modify the Fibonacci Heap data structure to implement SECONDMIN in constant time.

5. Give an efficient implementation of the operation **Fib-Heap-Change-Key(**H, x, k**)**, which changes the key of a node x in a Fibonacci heap H to the value k. The changes you make to Fibonacci heap data structure to support your implementation should not affect the amortized running time of any other Fibonacci heap operations. Analyze the amortized running time of your implementation for cases in which k is greater than, less than, or equal to $key[x]$.

CS 373: Combinatorial Algorithms, Spring 1999

`http://www-courses.cs.uiuc.edu/~cs373`

Homework 2 (due Thu. Feb. 18, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ►. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
2. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
3. Justify the correctness of your algorithm, including termination if that is not obvious.
4. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

►1. Faster Longest Increasing Subsequence (LIS)
Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. Hint: In the dynamic programming solution, you don't really have to look back at all previous items.

►2. SELECT(A, k)
Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.

(a) Show that a rotation in an augmented binary tree can be performed in constant time.

(b) Describe an algorithm SCAPEGOATSELECT(k) that selects the kth smallest item in an augmented scapegoat tree in $O(\log n)$ *worst-case* time. (The scapegoat trees presented in class were already augmented.)

(c) Describe an algorithm SPLAYSELECT(k) that selects the kth smallest item in an augmented splay tree in $O(\log n)$ *amortized* time.

(d) Describe an algorithm TREAPSELECT(k) that selects the kth smallest item in an augmented treap in $O(\log n)$ *expected* time.

►3. Scapegoat trees

(a) Prove that only one tree gets rebalanced at any insertion.

(b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree ($I(v) = max(0, |\hat{v}| - |\check{v}|)$, where $\hat{v}$ is the child of greater height and $\check{v}$ the child of lesser height, $|v|$ is the number of nodes in subtree v, and perfectly balanced means each subtree has as close to half the leaves as possible and is perfectly balanced itself.

*(c) Show that you can rebuild a fully balanced binary tree in $O(n)$ time using only $O(1)$ additional memory.

►4. Memory Management

Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than 1/4 full, we we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do not use the potential method (it makes it much more difficult).

Only 1U Grad Problems

▷1. Detecting overlap

(a) You are given a list of ranges represented by min and max (e.g. [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$-time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.

(b) You are given a list of rectangles represented by min and max x- and y- coordinates. Give an $O(n \log n)$-time algorithm that decides whether or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.

Practice Problems

1. Amortization

   (a) Modify the binary double-counter (see class notes Feb. 2) to support a new operation Sign, which determines whether the number being stored is positive, negative, or zero, in constant time. The amortized time to increment or decrement the counter should still be a constant.

   [Hint: Suppose p is the number of significant bits in P, and n is the number of significant bits in N. For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

   *(b) Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

*2. Amortization
   Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of "fits", where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is *not* the same representation as on Homework 0.]

3. Rotations

   (a) Show that it is possible to transform any n-node binary search tree into any other n-node binary search tree using at most $2n - 2$ rotations.

   *(b) Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.

4. Fibonacci Heaps: SecondMin
   We want to find the second smallest of a set efficiently.

   (a) Implement SecondMin by using a Fibonacci heap as a black box. Remember to justify its correctness and running time.

   *(b) Modify the Fibonacci Heap data structure to implement SecondMin in constant time.

5. Give an efficient implementation of the operation **Fib-Heap-Change-Key(**H, x, k**)**, which changes the key of a node x in a Fibonacci heap H to the value k. The changes you make to Fibonacci heap data structure to support your implementation should not affect the amortized running time of any other Fibonacci heap operations. Analyze the amortized running time of your implementation for cases in which k is greater than, less than, or equal to $key[x]$.

CS 373: Combinatorial Algorithms, Spring 1999

`http://www-courses.cs.uiuc.edu/~cs373`

Homework 3 (due Thu. Mar. 11, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. (New!) If not already done, model the problem appropriately. Often the problem is stated in real world terms; give a more rigorous description of the problem. This will help you figure out what is assumed (what you know and what is arbitrary, what operations are and are not allowed), and find the tools needed to solve the problem.
2. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
3. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
4. Justify the correctness of your algorithm, including termination if that is not obvious.
5. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

▶1. Hashing

(a) (2 pts) Consider an open-address hash table with uniform hashing and a load factor $\alpha = 1/2$. What is the expected number of probes in an unsuccessful search? Successful search?

(b) (3 pts) Let the hash function for a table of size m be

$$h(x) = \lfloor Amx \rfloor \bmod m$$

where $A = \hat{\phi} = \frac{\sqrt{5}-1}{2}$. Show that this gives the best possible spread, i.e. if the x are hashed in order, $x + 1$ will be hashed in the largest remaining contiguous interval.

►2. (5 pts) Euler Tour:
Given an **undirected** graph $G = (V, E)$, give an algorithm that finds a cycle in the graph that visits every edge *exactly* once, or says that it can't be done.

►3. (5 pts) Makefiles:
In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called 'make' that only recompiles those files that were changed, *and* any intermediate files in the compiliation that depend on those changed. Design an algorithm to recompile only those necessary.

►4. (5 pts) Shortest Airplane Trip:
A person wants to fly from city A to city B in the shortest possible time. S/he turns to the traveling agent who knows all the departure and arrival times of all the flights on the planet. Give an algorithm that will allow the agent to choose an optimal route. Hint: rather than modify Dijkstra's algorithm, modify the data. The time is from departure to arrival at the destination, so it will include layover time (time waiting for a connecting flight).

►5. (9 pts, 3 each) Minimum Spanning Tree changes Suppose you have a graph G and an MST of that graph (i.e. the MST has already been constructed).

(a) Give an algorithm to update the MST when an edge is added to G.

(b) Give an algorithm to update the MST when an edge is deleted from G.

(c) Give an algorithm to update the MST when a vertex (and possibly edges to it) is added to G.

Only 1U Grad Problems

▷1. Nesting Envelopes
You are given an unlimited number of each of n different types of envelopes. The dimensions of envelope type i are $x_i \times y_i$. In nesting envelopes inside one another, you can place envelope A inside envelope B if and only if the dimensions A are *strictly smaller* than the dimensions of B. Design and analyze an algorithm to determine the largest number of envelopes that can be nested inside one another.

Practice Problems

1. The incidence matrix of an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that
$$b_{ij} = \begin{cases} 1 & (i,j) \in E, \\ 0 & (i,j) \notin E. \end{cases}$$
   (a) Describe what all the entries of the matrix product BB^T represent (B^T is the matrix transpose). Justify.
   (b) Describe what all the entries of the matrix product B^TB represent. Justify.
   ★(c) Let $C = BB^T - 2A$. Let C' be C with the first row and column removed. Show that $\det C'$ is the number of spanning trees. (A is the adjacency matrix of G, with zeroes on the diagonal).

2. $o(V^2)$ Adjacency Matrix Algorithms
   (a) Give an $O(V)$ algorithm to decide whether a directed graph contains a *sink* in an adjacency matrix representation. A sink is a vertex with in-degree $V - 1$.
   (b) An undirected graph is a scorpion if it has a vertex of degree 1 (the sting) connected to a vertex of degree two (the tail) connected to a vertex of degree $V - 2$ (the body) connected to the other $V - 3$ vertices (the feet). Some of the feet may be connected to other feet.

   Design an algorithm that decides whether a given adjacency matrix represents a scorpion by examining only $O(V)$ of the entries.
   (c) Show that it is impossible to decide whether G has at least one edge in $O(V)$ time.

3. Shortest Cycle:
   Given an **undirected** graph $G = (V, E)$, and a weight function $f : E \to \mathbf{R}$ on the **edges**, give an algorithm that finds (in time polynomial in V and E) a cycle of smallest weight in G.

4. Longest Simple Path:
   Let graph $G = (V, E)$, $|V| = n$. A *simple path* of G, is a path that does not contain the same vertex twice. Use dynamic programming to design an algorithm (not polynomial time) to find a simple path of maximum length in G. Hint: It can be done in $O(n^c 2^n)$ time, for some constant c.

5. Minimum Spanning Tree:
   Suppose all edge weights in a graph G are equal. Give an algorithm to compute an MST.

6. Transitive reduction:
   Give an algorithm to construct a *transitive reduction* of a directed graph G, i.e. a graph G^{TR} with the fewest edges (but with the same vertices) such that there is a path from a to b in G iff there is also such a path in G^{TR}.

7. (a) What is $5^{2^{29^{5^0}+23^{4^1}+17^{3^2}+11^{2^3}+5^{1^4}}} \bmod 6$?

(b) What is the capital of Nebraska? Hint: It is not Omaha. It is named after a famous president of the United States that was not George Washington. The distance from the Earth to the Moon averages roughly 384,000 km.

CS 373: Combinatorial Algorithms, Spring 1999

`http://www-courses.cs.uiuc.edu/~cs373`

Homework 4 (due Thu. Apr. 1, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ►. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: When a question asks you to "give/describe/present an algorithm", you need to do four things to receive full credit:

1. (New!) If not already done, model the problem appropriately. Often the problem is stated in real world terms; give a more rigorous description of the problem. This will help you figure out what is assumed (what you know and what is arbitrary, what operations are and are not allowed), and find the tools needed to solve the problem.
2. Design the most efficient algorithm possible. Significant partial credit will be given for less efficient algorithms, as long as they are still correct, well-presented, and correctly analyzed.
3. Describe your algorithm succinctly, using structured English/pseudocode. We don't want full-fledged compilable source code, but plain English exposition is usually not enough. Follow the examples given in the textbooks, lectures, homeworks, and handouts.
4. Justify the correctness of your algorithm, including termination if that is not obvious.
5. Analyze the time and space complexity of your algorithm.

Undergrad/.75U Grad/1U Grad Problems

►1. (5 pts total) Collinearity
Give an $O(n^2 \log n)$ algorithm to determine whether any three points of a set of n points are collinear. Assume two dimensions and *exact* arithmetic.

►2. (4 pts, 2 each) Convex Hull Recurrence
Consider the following generic recurrence for convex hull algorithms that divide and conquer:

$$T(n, h) = T(n_1, h_1) + T(n_2, h_2) + O(n)$$

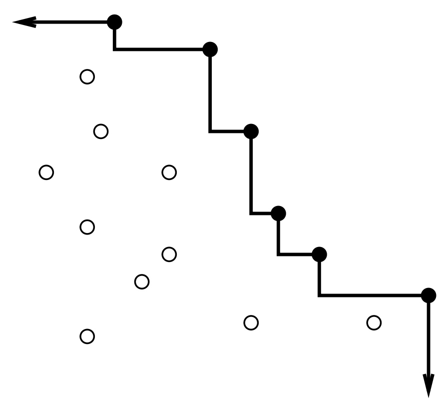
where $n \geq n_1 + n_2, h = h_1 + h_2$ and $n \geq h$. This means that the time to compute the convex hull is a function of both n, the number of input points, and h, the number of convex hull vertices. The splitting and merging parts of the divide-and-conquer algorithm take $O(n)$ time. When n is a constant, $T(n, h)$ is $O(1)$, but when h is a constant, $T(n, h)$ is $O(n)$. Prove that for both of the following restrictions, the solution to the recurrence is $O(n \log h)$:

(a) $h_1, h_2 < \frac{3}{4}h$

(b) $n_1, n_2 < \frac{3}{4}n$

►3. (5 pts) Circle Intersection
Give an $O(n \log n)$ algorithm to test whether any two circles in a set of size n intersect.

►4. (5 pts total) Staircases
You are given a set of points in the first quadrant. A *left-up* point of this set is defined to be a point that has no points both greater than it in both coordinates. The left-up subset of a set of points then forms a *staircase* (see figure).



(a) (3 pts) Give an $O(n \log n)$ algorithm to find the staircase of a set of points.

(b) (2 pts) Assume that points are chosen uniformly at random within a rectangle. What is the average number of points in a staircase? Justify. Hint: you will be able to give an exact answer rather than just asymptotics. You have seen the same analysis before.

Only 1U Grad Problems

▷1. (6 pts, 2 each) Ghostbusters and Ghosts
A group of n ghostbusters is battling n ghosts. Each ghostbuster can shoot a single energy beam at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits a ghost. The ghostbusters must all fire at the same time and no two energy beams may cross. The positions of the ghosts and ghostbusters is fixed in the plane (assume that no three points are collinear).

(a) Prove that for any configuration ghosts and ghostbusters there exists such a non-crossing matching.

(b) Show that there exists a line passing through one ghostbuster and one ghost such that the number of ghostbusters on one side of the line equals the number of ghosts on the same side. Give an efficient algorithm to find such a line.

(c) Give an efficient divide and conquer algorithm to pair ghostbusters and ghosts so that no two streams cross.

Practice Problems

1. Basic Computation (assume two dimensions and *exact* arithmetic)

   (a) Intersection: Extend the basic algorithm to determine if two line segments intersect by taking care of *all* degenerate cases.

   (b) Simplicity: Give an $O(n \log n)$ algorithm to determine whether an n-vertex polygon is simple.

   (c) Area: Give an algorithm to compute the area of a simple n-polygon (not necessarily convex) in $O(n)$ time.

   (d) Inside: Give an algorithm to determine whether a point is within a simple n-polygon (not necessarily convex) in $O(n)$ time.

2. External Diagonals and Mouths

   (a) A pair of polygon vertices defines an *external diagonal* if the line segment between them is completely outside the polygon. Show that every nonconvex polygon has at least one external diagonal.

   (b) Three consective polygon vertices p, q, r form a *mouth* if p and r define an external diagonal. Show that every nonconvex polygon has at least one mouth.

3. On-Line Convex Hull
   We are given the set of points one point at a time. After receiving each point, we must compute the convex hull of all those points so far. Give an algorithm to solve this problem in $O(n^2)$ (We could obviously use Graham's scan n times for an $O(n^2 \log n)$ algorithm). Hint: How do you maintain the convex hull?

4. Another On-Line Convex Hull Algorithm

   (a) Given an n-polygon and a point outside the polygon, give an algorithm to find a tangent.

   *(b) Suppose you have found both tangents. Give an algorithm to remove the points from the polygon that are within the angle formed by the tangents (as segments!) and the opposite side of the polygon.

   (c) Use the above to give an algorithm to compute the convex hull on-line in $O(n \log n)$

$^\bigstar$5. Order of the size of the convex hull
   The convex hull on $n \geq 3$ points can have anywhere from 3 to n points. The average case depends on the distribution.

   (a) Prove that if a set of points is chosen randomly within a given rectangle. then the average size of the convex hull is $O(\log n)$.

   (b) Prove that if a set of points is chosen randomly within a given circle. then the average size of the convex hull is $O(\sqrt{n})$.

CS 373: Combinatorial Algorithms, Spring 1999

http://www-courses.cs.uiuc.edu/~cs373

Homework 5 (due Thu. Apr. 22, 1999 by noon)

Name:	
Net ID:	Alias:

Everyone must do the problems marked ▶. Problems marked ▷ are for 1-unit grad students and others who want extra credit. (There's no such thing as "partial extra credit"!) Unmarked problems are extra practice problems for your benefit, which will not be graded. Think of them as potential exam questions.

Hard problems are marked with a star; the bigger the star, the harder the problem.

Note: You will be held accountable for the appropriate responses for answers (e.g. give models, proofs, analyses, etc)

Undergrad/.75U Grad/1U Grad Problems

▶1. (5 pts) Show how to find the occurrences of pattern P in text T by computing the prefix function of the string PT (the concatenation of P and T).

▶2. (10 pts total) Fibonacci strings and KMP matching
Fibonacci strings are defined as follows:

$$F_1 = \text{“b”}, \qquad F_2 = \text{“a”}, \qquad \text{and } F_n = F_{n-1}F_{n-2}, (n > 2)$$

where the recursive rule uses concatenation of strings, so F_2 is "ab", F_3 is "aba". Note that the length of F_n is the nth Fibonacci number.

(a) (2 pts) Prove that in any Fibonacci string there are no two b's adjacent and no three a's.

(b) (2 pts) Give the unoptimized and optimized 'prefix' (fail) function for F_7.

(c) (3 pts) Prove that, in searching for a Fibonacci string of length m using unoptimized KMP, it may shift up to $\lceil \log_\phi m \rceil$ times, where $\phi = (1 + \sqrt{5})/2$, is the golden ratio. (Hint: Another way of saying the above is that we are given the string F_n and we may have to shift n times. Find an example text T that gives this number of shifts).

(d) (3 pts) What happens here when you use the optimized prefix function? Explain.

▶3. (5 pts) Prove that finding the second smallest of n elements takes $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. Prove for both upper and lower bounds. Hint: find the (first) smallest using an elimination tournament.

▶4. (4 pts, 2 each) Lower Bounds on Adjacency Matrix Representations of Graphs

(a) Prove that the time to determine if an undirected graph has a cycle is $\Omega(V^2)$.

(b) Prove that the time to determine if there is a path between two nodes in an undirected graph is $\Omega(V^2)$.

Only 1U Grad Problems

$\triangleright$1. (5 pts) Prove that $\lceil 3n/2 \rceil - 2$ comparisons are necessary in the worst case to find both the minimum and maximum of n numbers. Hint: Consider how many are potentially either the min or max.

Practice Problems

1. String matching with wild-cards
   Suppose you have an alphabet for patterns that includes a 'gap' or wild-card character; any length string of any characters can match this additional character. For example if '$*$' is the wild-card, then the pattern 'foo*bar*nad' can be found in 'foofoowangbarnad'. Modify the computation of the prefix function to correctly match strings using KMP.

2. Prove that there is no comparison sort whose running time is linear for at least 1/2 of the $n!$ inputs of length n. What about at least $1/n$? What about at least $1/2^n$?.

3. Prove that $2n - 1$ comparisons are necessary in the worst case to merge two sorted lists containing n elements each.

4. Find asymptotic upper and lower bounds to $\lg(n!)$ without Stirling's approximation (Hint: use integration).

5. Given a sequence of n elements of n/k blocks (k elements per block) all elements in a block are less than those to the right in sequence, show that you cannot have the whole sequence sorted in better than $\Omega(n \lg k)$. Note that the entire sequence would be sorted if each of the n/k blocks were individually sorted in place. Also note that combining the lower bounds for each block is not adequate (that only gives an upper bound).

6. Some elementary reductions

   (a) Prove that if you can decide whether a graph G has a clique of size k (or less) then you can decide whether a graph G' has an independent set of size k (or more).

   (b) Prove that if you can decide whether one graph G_1 is a subgraph of another graph G_2 then you can decide whether a graph G has a clique of size k (or less).

7. There is no Proof but We are pretty Sure
   Justify (prove) the following logical rules of inference:

   (a) Classical - If $a \rightarrow b$ and a hold, then b holds.

   (b) Fuzzy - Prove: If $a \rightarrow b$ holds, and a holds with probability p, then b holds with probability less than p. Assume all probabilities are independent.

   (c) Give formulas for computing the probabilities of the fuzzy logical operators 'and', 'or', 'not', and 'implies', and fill out truth tables with the values T (true, $p = 1$), L (likely, $p = 0.9$), M (maybe, $p = 0.5$), N (not likely, $p = 0.1$), and F (false, $p = 0$).

(d) If you have a poly time (algorithmic) reduction from problem B to problem A (i.e. you can solve B using A with a poly time conversion), and it is very unlikely that A has better than lower bound $\Omega(2^n)$ algorithm, what can you say about problem A. Hint: a solution to A implies a solution to B.

CS 373: Combinatorial Algorithms, Spring 1999

Midterm 1 (February 23, 1999)

Name:	
Net ID:	Alias:

This is a closed-book, closed-notes exam!

If you brought anything with you besides writing instruments and your $8\frac{1}{2}'' \times 11''$ cheat sheet, please leave it at the front of the classroom.

- **Don't panic!**
- Print your name, netid, and alias in the boxes above, and print your name at the top of every page.
- **Answer four of the five questions on the exam.** Each question is worth 10 points. If you answer more than four questions, the one with the lowest score will be ignored. **1-unit graduate students must answer question #5.**
- Please write your answers on the front of the exam pages. Use the backs of the pages as scratch paper. Let us know if you need more paper.
- Read the entire exam before writing anything. Make sure you understand what the questions are asking. If you give a beautiful answer to the wrong question, you'll get no credit. If any question is unclear, please ask one of us for clarification.
- Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.
- Write *something* down for every problem. Don't panic and erase large chunks of work. Even if you think it's nonsense, it might be worth partial credit.
- **Don't panic!**

#	Score	Grader
1		
2		
3		
4		
5		

1. **Multiple Choice**

   Every question below has one of the following answers.

   (a) $\Theta(1)$ (b) $\Theta(\log n)$ (c) $\Theta(n)$ (d) $\Theta(n \log n)$ (e) $\Theta(n^2)$

   For each question, write the letter that corresponds to your answer. You do not need to justify your answer. Each correct answer earns you 1 point, but each incorrect answer *costs* you $\frac{1}{2}$ point. (You cannot score below zero.)

☐ What is $\sum_{i=1}^{n} i$?

☐ What is $\sum_{i=1}^{n} \frac{1}{i}$?

☐ What is the solution of the recurrence $T(n) = T(\sqrt{n}) + n$?

☐ What is the solution of the recurrence $T(n) = T(n-1) + \lg n$?

☐ What is the solution of the recurrence $T(n) = 2T\left(\left\lceil \frac{n+27}{2} \right\rceil\right) + 5n - 7\sqrt{\lg n} + \frac{1999}{n}$?

☐ The amortized time for inserting one item into an n-node splay tree is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty splay tree?

☐ The expected time for inserting one item into an n-node randomized treap is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty treap?

☐ What is the worst-case running time of randomized quicksort?

☐ How many bits are there in the binary representation of the nth Fibonacci number?

☐ What is the worst-case cost of merging two arbitrary splay trees with n items total into a single splay tree with n items.

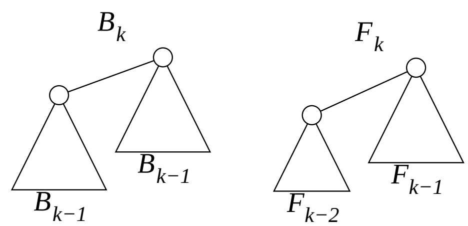
☐ Suppose you correctly identify three of the answers to this question as obviously wrong. If you pick one of the two remaining answers at random, what is your expected score for this problem?

2. (a) **[5 pt]** Recall that a binomial tree of order k, denoted B_k, is defined recursively as follows. B_0 is a single node. For any $k > 0$, B_k consists of two copies of B_{k-1} linked together.

   Prove that the degree of any node in a binomial tree is equal to its height.

   (b) **[5 pt]** Recall that a Fibonacci tree of order k, denoted F_k, is defined recursively as follows. F_1 and F_2 are both single nodes. For any $k > 2$, F_k consists of an F_{k-2} linked to an F_{k-1}.

   Prove that for any node v in a Fibonacci tree, $\mathrm{height}(v) = \lceil \mathrm{degree}(v)/2 \rceil$.



Recursive definitions of binomial trees and Fibonacci trees.

3. Consider the following randomized algorithm for computing the smallest element in an array.

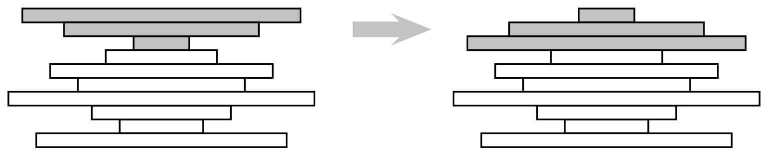
```
RANDOMMIN(A[1..n]):
  min ← ∞
  for i ← 1 to n in random order
      if A[i] < min
          min ← A[i]    (⋆)
      return min
```

(a) **[1 pt]** In the worst case, how many times does RANDOMMIN execute line $(\star)$?

(b) **[3 pt]** What is the probability that line $(\star)$ is executed during the nth iteration of the for loop?

(c) **[6 pt]** What is the exact expected number of executions of line $(\star)$? (A correct $\Theta()$ bound is worth 4 points.)

4. Suppose we have a stack of n pancakes of different sizes. We want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation we can perform is a *flip* — insert a spatula under the top k pancakes, for some k between 1 and n, and flip them all over.



Flipping the top three pancakes

   (a) **[3 pt]** Describe an algorithm to sort an arbitrary stack of n pancakes.

   (b) **[3 pt]** Prove that your algorithm is correct.

   (c) **[2 pt]** Exactly how many flips does your algorithm perform in the worst case? (A correct $\Theta()$ bound is worth one point.)

   (d) **[2 pt]** Suppose one side of each pancake is burned. Exactly how many flips do you need to sort the pancakes *and* have the burned side of every pancake on the bottom? (A correct $\Theta()$ bound is worth one point.)

5. You are given an array $A[1 \,..\, n]$ of integers. Describe and analyze an algorithm that finds the largest sum of of elements in a contiguous subarray $A[i \,..\, j]$.

   For example, if the array contains the numbers $(-6, 12, -7, 0, 14, -7, 5)$, then the largest sum is $19 = 12 - 7 + 0 + 14$.

   $$\boxed{-6} \underbrace{\boxed{12}\boxed{-7}\boxed{0}\boxed{14}}_{19} \boxed{-7}\boxed{5}$$

   To get full credit, your algorithm must run in $\Theta(n)$ time — there are at least three different ways to do this. An algorithm that runs in $\Theta(n^2)$ time is worth 7 points.

CS 373: Combinatorial Algorithms, Spring 1999

Midterm 2 (April 6, 1999)

Name:	
Net ID:	Alias:

This is a closed-book, closed-notes exam!

If you brought anything with you besides writing instruments and your $8\frac{1}{2}'' \times 11''$ cheat sheet, please leave it at the front of the classroom.

- **Don't panic!**
- Print your name, netid, and alias in the boxes above, and print your name at the top of every page.
- **Answer four of the five questions on the exam.** Each question is worth 10 points. If you answer more than four questions, the one with the lowest score will be ignored. **1-unit graduate students must answer question #5.**
- Please write your answers on the front of the exam pages. You can use the backs of the pages as scratch paper. Let us know if you need more paper.
- Read the entire exam before writing anything. Make sure you understand what the questions are asking. If you give a beautiful answer to the wrong question, you'll get no credit. If any question is unclear, please ask one of us for clarification.
- Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.
- Write *something* down for every problem. Don't panic and erase large chunks of work. Even if you think it's nonsense, it might be worth partial credit.
- **Don't panic!**

#	Score	Grader
1		
2		
3		
4		
5		

1. **Bipartite Graphs**

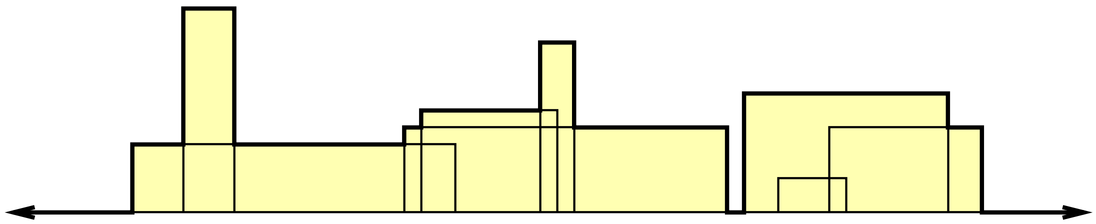
   A graph (V, E) is *bipartite* if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.

   (a) Prove that every tree is a bipartite graph.

   (b) Describe and analyze an efficient algorithm that determines whether a given connected, undirected graph is bipartite.

2. **Manhattan Skyline**

   The purpose of the following problem is to compute the outline of a projection of rectangular buildings. You are given the height, width, and left x-coordinate of n rectangles. The bottom of each rectangle is on the x-axis. Describe and analyze an efficient algorithm to compute the vertices of the "skyline".



A set of rectangles and its skyline.

3. **Least Cost Vertex Weighted Path**

   Suppose you want to drive from Champaign to Los Angeles via a network of roads connecting cities. You don't care how long it takes, how many cities you visit, or how much gas you use. All you care about is how much money you spend on food. Each city has a possibly different, but fixed, value for food.

   More formally, you are given a directed graph $G = (V, E)$ with nonnegative weights on the *vertices* $w\colon V \to \mathbb{R}^+$, a source vertex $s \in V$, and a target vertex $t \in V$. Describe and analyze an efficient algorithm to find a minimum-weight path from s to t. [Hint: Modify the graph.]

4. **Union-Find with Alternate Rule**

   In the UNION-FIND data structure described in CLR and in class, each set is represented by a rooted tree. The UNION algorithm, given two sets, decides which set is to be the parent of the other by comparing their ranks, where the rank of a set is an upper bound on the height of its tree.

   Instead of rank, we propose using the *weight* of the set, which is just the number of nodes in the set. Here's the modified UNION algorithm:

```
UNION(A, B):
  if weight(A) > weight(B)
      parent(B) ← A
      weight(A) ← weight(A) + weight(B)
  else
      parent(A) ← B
      weight(B) ← weight(A) + weight(B)
```

   Prove that if we use this method, then after any sequence of n MAKESETs, UNIONs, and FINDs (with path compression), the *height* of the tree representing any set is $O(\log n)$.

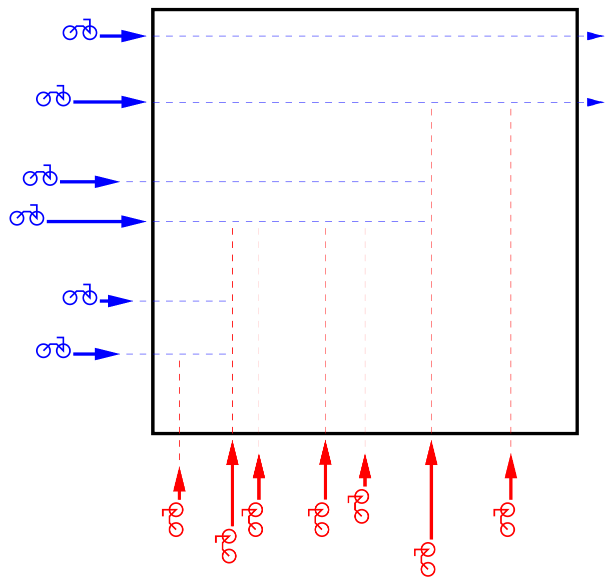
   [Hint: First prove it without path compression, and then argue that path compression doesn't matter (for this problem).]

5. **Motorcycle Collision**

   One gang, Hell's Ordinates, start west of the arena facing directly east; the other, The Vicious Abscissas of Death, start south of the arena facing due north. All the motorcycles start moving simultaneously at a prearranged signal. Each motorcycle moves at a fixed speed—no speeding up, slowing down, or turning is allowed. Each motorcycle leaves an oil trail behind it. If another motorcycle crosses that trail, it falls over and stops leaving a trail.

   More formally, we are given two sets H and V, each containing n motorcycles. Each motorcycle is represented by three numbers (s, x, y): its speed and the x- and y-coordinates of its initial location. Bikes in H move horizontally; bikes in V move vertically.

   Assume that the bikes are infinitely small points, that the bike trails are infinitely thin lie segments, that a bike crashes stops *exactly* when it hits a oil trail, and that no two bikes collide with each other.



Two sets of motorcycles and the oil trails they leave behind.

   (a) Solve the case $n = 1$. Given only two motorcycles moving perpendicular to each other, determine which one of them falls over and where in $O(1)$ time.

   (b) Describe an efficient algorithm to find the set of all points where motorcycles fall over.

5. **Motorcycle Collision (continued)**

   Incidentally, the movie *Tron* is being shown during Roger Ebert's Forgotten Film Festival at the Virginia Theater in Champaign on April 25. Get your tickets now!

CS 373: Combinatorial Algorithms, Spring 1999

Final Exam (May 7, 1999)

Name:	
Net ID:	Alias:

This is a closed-book, closed-notes exam!

If you brought anything with you besides writing instruments and your two $8\frac{1}{2}'' \times 11''$ cheat sheets, please leave it at the front of the classroom.

- Print your name, netid, and alias in the boxes above, and print your name at the top of every page.
- **Answer six of the seven questions on the exam.** Each question is worth 10 points. If you answer every question, the one with the lowest score will be ignored. **1-unit graduate students must answer question #7.**
- Please write your answers on the front of the exam pages. Use the backs of the pages as scratch paper. Let us know if you need more paper.
- Read the entire exam before writing anything. Make sure you understand what the questions are asking. If you give a beautiful answer to the wrong question, you'll get no credit. If any question is unclear, please ask one of us for clarification.
- Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.
- Write *something* down for every problem. Don't panic and erase large chunks of work. Even if you think it's nonsense, it might be worth partial credit.

#	Score	Grader
1		
2		
3		
4		
5		
6		
7		

1. **Short Answer**

sorting	induction	Master theorem	divide and conquer
randomized algorithm	amortization	brute force	hashing
binary search	depth-first search	splay tree	Fibonacci heap
convex hull	sweep line	minimum spanning tree	shortest paths
shortest path	adversary argument	NP-hard	reduction
string matching	evasive graph property	dynamic programming	H_n

Choose from the list above the best method for solving each of the following problems. We do *not* want complete solutions, just a short description of the proper solution technique! Each item is worth 1 point.

(a) Given a Champaign phone book, find your own phone number.

(b) Given a collection of n rectangles in the plane, determine whether any two intersect in $O(n \log n)$ time.

(c) Given an undirected graph G and an integer k, determine if G has a complete subgraph with k edges.

(d) Given an undirected graph G, determine if G has a triangle — a complete subgraph with three vertices.

(e) Prove that any n-vertex graph with minimum degree at least $n/2$ has a Hamiltonian cycle.

(f) Given a graph G and three distinguished vertices u, v, and w, determine whether G contains a path from u to v that passes through w.

(g) Given a graph G and two distinguished vertices u and v, determine whether G contains a path from u to v that passes through at most 17 edges.

(h) Solve the recurrence $T(n) = 5T(n/17) + O(n^{4/3})$.

(i) Solve the recurrence $T(n) = 1/n + T(n-1)$, where $T(0) = 0$.

(j) Given an array of n integers, find the integer that appears most frequently in the array.

(a) ____________________ (f) ____________________

(b) ____________________ (g) ____________________

(c) ____________________ (h) ____________________

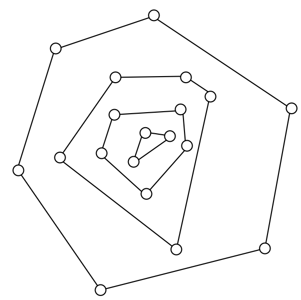
(d) ____________________ (i) ____________________

(e) ____________________ (j) ____________________

2. **Convex Layers**

   Given a set Q of points in the plane, define the *convex layers* of Q inductively as follows: The first convex layer of Q is just the convex hull of Q. For all $i > 1$, the ith convex layer is the convex hull of Q after the vertices of the first $i - 1$ layers have been removed.

   Give an $O(n^2)$-time algorithm to find all convex layers of a given set of n points. [Partial credit for a correct slower algorithm; extra credit for a correct faster algorithm.]



A set of points with four convex layers.

3. Suppose you are given an array of n numbers, sorted in increasing order.

   (a) **[3 pts]** Describe an $O(n)$-time algorithm for the following problem:

   Find two numbers from the list that add up to zero, or report that there is no such pair. In other words, find two numbers a and b such that $a + b = 0$.

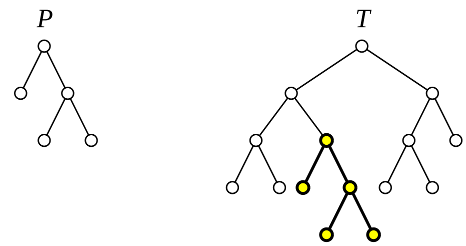
   (b) **[7 pts]** Describe an $O(n^2)$-time algorithm for the following problem:

   Find *three* numbers from the list that add up to zero, or report that there is no such triple. In other words, find three numbers a, b, and c, such that $a + b + c = 0$. [Hint: Use something similar to part (a) as a subroutine.]

4. **Pattern Matching**

   (a) **[4 pts]** A *cyclic rotation* of a string is obtained by chopping off a prefix and gluing it at the end of the string. For example, ALGORITHM is a cyclic shift of RITHMALGO. Describe and analyze an algorithm that determines whether one string $P[1 \mathinner{..} m]$ is a cyclic rotation of another string $T[1 \mathinner{..} n]$.

   (b) **[6 pts]** Describe and analyze an algorithm that decides, given any two binary trees P and T, whether P equals a subtree of T. [Hint: First transform both trees into strings.]



P occurs exactly once as a subtree of T.

5. **Two-stage Sorting**

   (a) **[1 pt]** Suppose we are given an array $A[1\,..\,n]$ of distinct integers. Describe an algorithm that splits A into n/k subarrays, each with k elements, such that the elements of each subarray $A[(i-1)k+1\,..\,ik]$ are sorted. Your algorithm should run in $O(n \log k)$ time.

   (b) **[2 pts]** Given an array $A[1\,..\,n]$ that is already split into n/k sorted subarrays as in part (a), describe an algorithm that sorts the entire array in $O(n \log(n/k))$ time.

   (c) **[3 pts]** Prove that your algorithm from part (a) is optimal.

   (d) **[4 pts]** Prove that your algorithm from part (b) is optimal.

4	14	7	3	1	20	11	9	5	13	12	19	10	16	17	2	8	6	18	15

$$\Downarrow (a)$$

1	4	3	7	14	‖	5	9	11	13	20	‖	10	12	16	17	19	‖	2	6	8	15	18

$$\Downarrow (b)$$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

6. **SAT Reduction**

   Suppose you are have a black box that magically solves SAT (the formula satisfiability problem) in constant time. That is, given a boolean formula of variables and logical operators ($\wedge, \vee, \neg$), the black box tells you, in constant time, whether or not the formula can be satisfied. Using this black box, design and analyze a **polynomial-time** algorithm that computes an assignment to the variables that satisfies the formula.

7. **Knapsack**

   You're hiking through the woods when you come upon a treasure chest filled with objects. Each object has a different size, and each object has a price tag on it, giving its value. There is no correlation between an object's size and its value. You want to take back as valuable a subset of the objects as possible (in one trip), but also making sure that you will be able to carry it in your knapsack which has a limited size.

   In other words, you have an integer capacity K and a target value V, and you want to decide whether there is a subset of the objects whose total size is *at most* K and whose total value is *at least* V.

   (a) **[5 pts]** Show that this problem is NP-hard. [Hint: Restate the problem more formally, then reduce from the NP-hard problem PARTITION: Given a set S of nonnegative integers, is there a partition of S into disjoint subsets A and B (where $A \cup B = S$) whose sums are equal, *i.e.*, $\sum_{a \in A} a = \sum_{b \in B} b$.]

   (b) **[5 pts]** Describe and analyze a dynamic programming algorithm to solve the knapsack problem in $O(nK)$ time. Prove your algorithm is correct.

CS 373: Combinatorial Algorithms, Fall 2000

Homework 0, due August 31, 2000 at the beginning of class

Name:	
Net ID:	Alias:

Neatly print your name (first name first, with no comma), your network ID, and a short alias into the boxes above. **Do not *sign* your name. Do *not* write your Social Security number.** Staple this sheet of paper to the top of your homework.

Grades will be listed on the course web site by alias give us, so your alias should not resemble your name or your Net ID. If you don't give yourself an alias, we'll give you one that you won't like.

Before you do anything else, read the Homework Instructions and FAQ on the CS 373 course web page (http://www-courses.cs.uiuc.edu/~cs373/hw/faq.html), and then check the box below. This web page gives instructions on how to write and submit homeworks—staple your solutions together in order, write your name and netID on every page, don't turn in source code, analyze everything, use good English and good logic, and so forth.

☐ I have read the CS 373 Homework Instructions and FAQ.

This homework tests your familiarity with the prerequisite material from CS 173, CS 225, and CS 273—many of these problems have appeared on homeworks or exams in those classes—primarily to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Parberry and Chapters 1–6 of CLR should be sufficient review, but you may want to consult other texts as well.

Required Problems

1. Sort the following 25 functions from asymptotically smallest to asymptotically largest, indicating ties if there are any:

$$\begin{array}{ccccc}
1 & n & n^2 & \lg n & \lg(n \lg n) \\
\lg^* n & \lg^* 2^n & 2^{\lg^* n} & \lg \lg^* n & \lg^* \lg n \\
n^{\lg n} & (\lg n)^n & (\lg n)^{\lg n} & n^{1/\lg n} & n^{\lg \lg n} \\
\log_{1000} n & \lg^{1000} n & \lg^{(1000)} n & \left(1 + \frac{1}{n}\right)^n & n^{1/1000}
\end{array}$$

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2, n, $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

2. (a) Prove that any positive integer can be written as the sum of distinct powers of 2. For example: $42 = 2^5 + 2^3 + 2^1$, $25 = 2^4 + 2^3 + 2^0$, $17 = 2^4 + 2^0$. [Hint: "Write the number in binary" is *not* a proof; it just restates the problem.]

   (b) Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.

   (c) Prove that *any* integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example: $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.

3. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. If no base cases are given, assume something reasonable but nontrivial. Extra credit will be given for more exact solutions.

   (a) $A(n) = 3A(n/2) + n$

   (b) $B(n) = \max_{n/3<k<2n/3} \big(B(k) + B(n-k) + n\big)$

   (c) $C(n) = 4C(\lfloor n/2 \rfloor + 5) + n^2$

   *(d) $D(n) = 2D(n/2) + n/\lg n$

   *(e) $E(n) = \dfrac{E(n-1)}{E(n-2)}$, where $E(1) = 1$ and $E(2) = 2$.

4. Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, $\ldots$, 52 of clubs. (They're big cards.) Penn shuffles the deck until each each of the $52!$ possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the three of clubs.

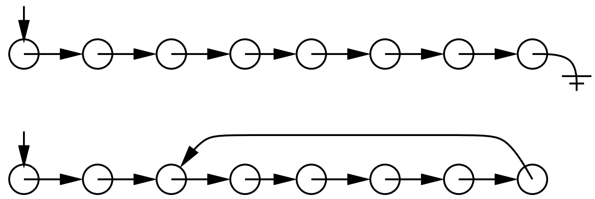
   (a) On average, how many cards does Penn give Teller?

   (b) On average, what is the smallest-numbered card that Penn gives Teller?

   *(c) On average, what is the largest-numbered card that Penn gives Teller?

   [Hint: Solve for an n-card deck, and then set $n = 52$.] Prove that your answers are correct. If you have to appeal to "intuition" or "common sense", your answers are probably wrong!

5. Suppose you have a pointer to the head of singly linked list. Normally, each node in the list only has a pointer to the next element, and the last node's pointer is NULL. Unfortunately, your list might have been corrupted by a bug in somebody else's code[1], so that the last node has a pointer back to some other node in the list instead.

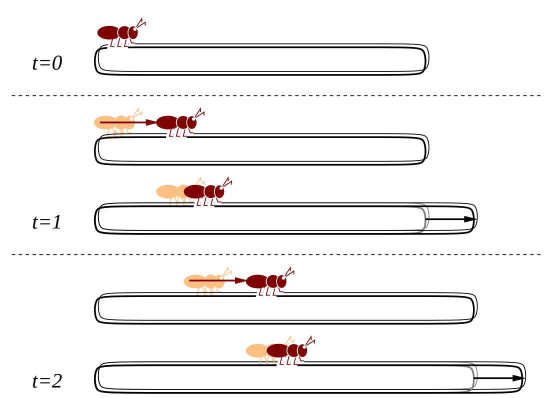


Top: A standard singly-linked list. Bottom: A corrupted singly linked list.

Describe an algorithm[2] that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the list, and use $O(1)$ extra space (not counting the list itself).

6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

An ant is walking along a rubber band, starting at the left end. Once every second, the ant walks one inch to the right, and then you make the rubber band one inch longer by pulling on the right end. The rubber band stretches uniformly, so stretching the rubber band also pulls the ant to the right. The initial length of the rubber band is n inches, so after t seconds, the rubber band is $n + t$ inches long.



Every second, the ant walks an inch, and then the rubber band is stretched an inch longer.

(a) How far has the ant moved after t seconds, as a function of n and t? Set up a recurrence and (for full credit) give an *exact* closed-form solution. [Hint: What *fraction* of the rubber band's length has the ant walked?]

*(b) How long does it take the ant to get to the right end of the rubber band? For full credit, give an answer of the form $f(n) + \Theta(1)$ for some explicit function $f(n)$.

[1]After all, *your* code is always completely 100% bug-free. Isn't that right, Mr. Gates?

[2]Since you've read the Homework Instructions, you know what the phrase "describe an algorithm" means. Right?

Practice Problems

These remaining practice problems are entirely for your benefit. Don't turn in solutions—we'll just throw them out—but feel free to ask us about these questions during office hours and review sessions. Think of these as potential exam questions (hint, hint).

1. Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m.

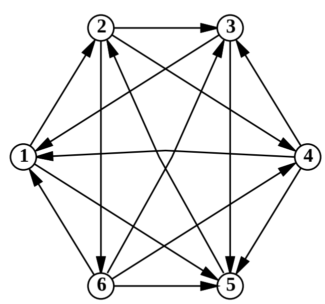
   (a) F_n is even if and only if n is divisible by 3.

   (b) $\sum_{i=0}^{n} F_i = F_{n+2} - 1$

   (c) $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$

   ★(d) If n is an integer multiple of m, then F_n is an integer multiple of F_m.

2. A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once. Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \to 4 \to 5 \to 2 \to 3 \to 1$.

3. (a) Prove the following identity by induction:

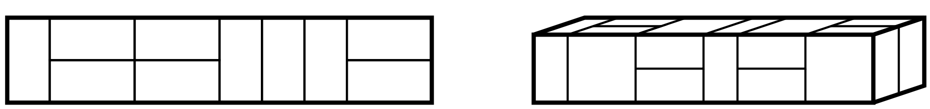
$$\binom{2n}{n} = \sum_{k=0}^{n} \binom{n}{k} \binom{n}{n-k}.$$

   (b) Give a non-inductive combinatorial proof of the same identity, by showing that the two sides of the equation count exactly the same thing in two different ways. There is a correct one-sentence proof.

4. (a) Prove that $2^{\lceil \lg n \rceil + \lfloor \lg n \rfloor}/n = \Theta(n)$.
   (b) Is $2^{\lfloor \lg n \rfloor} = \Theta\big(2^{\lceil \lg n \rceil}\big)$? Justify your answer.
   (c) Is $2^{2^{\lfloor \lg \lg n \rfloor}} = \Theta\big(2^{2^{\lceil \lg \lg n \rceil}}\big)$? Justify your answer.
   (d) Prove that if $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$. Justify your answer.
   (e) Prove that $f(n) = O(g(n))$ does *not* imply that $\log(f(n)) = O(\log(g(n)))$?.
   *(f) Prove that $\log^k n = o(n^{1/k})$ for any positive integer k.

5. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. If no base cases are given, assume something reasonable (but nontrivial). Extra credit will be given for more exact solutions.
   (a) $A(n) = A(n/2) + n$
   (b) $B(n) = 2B(n/2) + n$
   (c) $C(n) = \min\limits_{0<k<n} \big(C(k) + C(n-k) + 1\big)$, where $C(1) = 1$.
   (d) $D(n) = D(n-1) + 1/n$
   *(e) $E(n) = 8E(n-1) - 15E(n-2) + 1$
   *(f) $F(n) = (n-1)(F(n-1) + F(n-2))$, where $F(0) = F(1) = 1$
   $\bigstar$(g) $G(n) = G(n/2) + G(n/4) + G(n/6) + G(n/12) + n$ [Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.]

6. (a) A *domino* is a 2×1 or 1×2 rectangle. How many different ways are there to completely fill a $2 \times n$ rectangle with n dominos? Set up a recurrence relation and give an *exact* closed-form solution.
   (b) A *slab* is a three-dimensional box with dimensions $1 \times 2 \times 2$, $2 \times 1 \times 2$, or $2 \times 2 \times 1$. How many different ways are there to fill a $2 \times 2 \times n$ box with n slabs? Set up a recurrence relation and give an *exact* closed-form solution.



A 2×10 rectangle filled with ten dominos, and a $2 \times 2 \times 10$ box filled with ten slabs.

7. Professor George O'Jungle has a favorite 26-node binary tree, whose nodes are labeled by letters of the alphabet. The preorder and postorder sequences of nodes are as follows:

   preorder: `M N H C R S K W T G D X I Y A J P O E Z V B U L Q F`
   postorder: `C W T K S G R H D N A O E P J Y Z I B Q L F U V X M`

   Draw Professor O'Jungle's binary tree, and give the inorder sequence of nodes.

8. Alice and Bob each have a fair n-sided die. Alice rolls her die once. Bob then repeatedly throws his die until he rolls a number at least as big as the number Alice rolled. Each time Bob rolls, he pays Alice \$1. (For example, if Alice rolls a 5, and Bob rolls a 4, then a 3, then a 1, then a 5, the game ends and Alice gets \$4. If Alice rolls a 1, then no matter what Bob rolls, the game will end immediately, and Alice will get \$1.)

   Exactly how much money does Alice expect to win at this game? Prove that your answer is correct. If you have to appeal to "intuition" or "common sense", your answer is probably wrong!

9. Prove that for any nonnegative parameters a and b, the following algorithms terminate and produce identical output.

```
SlowEuclid(a, b):
  if b > a
      return SlowEuclid(b, a)
  else if b = 0
      return a
  else
      return SlowEuclid(b, a − b)
```

```
FastEuclid(a, b):
  if b = 0
      return a
  else
      return FastEuclid(b, a mod b)
```

CS 373: Combinatorial Algorithms, Fall 2000

Homework 1 (due September 12, 2000 at midnight)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the ith word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of whitespace between any two words on the same line.

   Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j, then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^{j} p_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

2. Consider the following sorting algorithm:

```
STUPIDSORT(A[0 .. n − 1]) :
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m ← ⌈2n/3⌉
      STUPIDSORT(A[0 .. m − 1])
      STUPIDSORT(A[n − m .. n − 1])
      STUPIDSORT(A[0 .. m − 1])
```

   (a) Prove that StupidSort actually sorts its input.

   (b) Would the algorithm still sort correctly if we replaced the line $m \leftarrow \lceil 2n/3 \rceil$ with $m \leftarrow \lfloor 2n/3 \rfloor$? Justify your answer.

   (c) State a recurrence (including the base case(s)) for the number of comparisons executed by StupidSort.

   (d) Solve the recurrence, and prove that your solution is correct. [Hint: Ignore the ceiling.] Does the algorithm deserve its name?

   *(e) Show that the number of *swaps* executed by StupidSort is at most $\binom{n}{2}$.

3. The following randomized algorithm selects the rth smallest element in an unsorted array $A[1 .. n]$. For example, to find the smallest element, you would call RandomSelect$(A, 1)$; to find the median element, you would call RandomSelect$(A, \lfloor n/2 \rfloor)$. Recall from lecture that Partition splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n - 1$ comparisons altogether, and returns the new index of the pivot element.

```
RANDOMSELECT(A[1 .. n], r) :
  p ← RANDOM(1, p)
  k ← PARTITION(A[1 .. n], p)
  if r < k
      return RANDOMSELECT(A[1 .. k − 1], r)
  else if r > k
      return RANDOMSELECT(A[k + 1 .. n], r − k)
  else
      return A[k]
```

   (a) State a recurrence for the expected running time of RandomSelect, as a function of n and r.

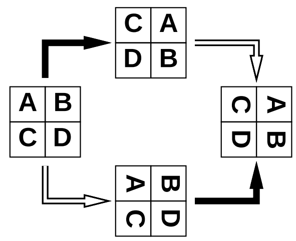
   (b) What is the *exact* probability that RandomSelect compares the ith smallest and jth smallest elements in the input array? The correct answer is a simple function of i, j, and r. [Hint: Check your answer by trying a few small examples.]

   *(c) What is the expected running time of RandomSelect, as a function of n and r? You can use either the recurrence from part (a) or the probabilities from part (b). For extra credit, give the *exact* expected number of comparisons.

   (d) What is the expected number of times that RandomSelect calls itself recursively?

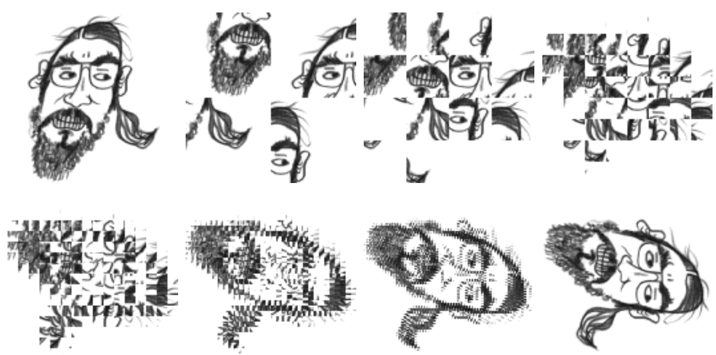
4. Some graphics hardware includes support for an operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixelmap (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

   Suppose we want to rotate an $n \times n$ pixelmap $90°$ clockwise. One way to do this is to split the pixelmap into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we can first recursively rotate the blocks and blit them into place afterwards.



Two algorithms for rotating a pixelmap.
Black arrows indicate blitting the blocks into place.
White arrows indicate recursively rotating the blocks.

The following sequence of pictures shows the first algorithm (blit then recurse) in action.



In the following questions, assume n is a power of two.

(a) Prove that both versions of the algorithm are correct. [Hint: If you exploit all the available symmetries, your proof will only be a half of a page long.]

(b) *Exactly* how many blits does the algorithm perform?

(c) What is the algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

(d) What if a $k \times k$ blit takes only $O(k)$ time?

5. The traditional Devonian/Cornish drinking song "The Barley Mow" has the following pseudolyrics[1], where $container[i]$ is the name of a container that holds 2^i ounces of beer.[2]

```
BarleyMow(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for i ← 1 to n
      "We'll drink it out of the container[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The container[j],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

(a) Suppose each container name $container[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

(b) If you want to sing this song for $n > 20$, you'll have to make up your own container names, and to avoid repetition, these names will get progressively longer as n increases[3]. Suppose $container[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

(c) Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each $container[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BarleyMow(n)? (Give an *exact* answer, not just an asymptotic bound.)

[1]Pseudolyrics are to lyrics as pseudocode is to code.

[2]One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.

[3]"We'll drink it out of the hemisemidemiyottapint, boys!"

6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

   A company is planning a party for its employees. The employees in the company are organized into a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party have assigned a real number to each employee measuring how 'fun' the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the 'fun' ratings of the guests.

Practice Problems

1. Give an $O(n^2)$ algorithm to find the longest increasing subsequence of a sequence of numbers. The elements of the subsequence need not be adjacent in the sequence. For example, the sequence $\langle 1, 5, 3, 2, 4 \rangle$ has longest increasing subsequence $\langle 1, 3, 4 \rangle$.

2. You are at a political convention with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party a delegate belongs to. However, you can check whether any two delegates are in the *same* party or not by introducing them to each other. (Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.)

   (a) Suppose a majority (more than half) of the delegates are from the same political party. Give an efficient algorithm that identifies a member of the majority party.

   (b) Suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick a person from the plurality party as parsimoniously as possible. (Please.)

3. Give an algorithm that finds the *second* smallest of n elements in at most $n + \lceil \lg n \rceil - 2$ comparisons. [Hint: divide and conquer to find the smallest; where is the second smallest?]

4. Suppose that you have an array of records whose keys to be sorted consist only of 0's and 1's. Give a simple, linear-time $O(n)$ algorithm to sort the array in place (using storage of no more than constant size in addition to that of the array).

5. Consider the problem of making change for n cents using the least number of coins.

   (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

   (b) Suppose that the available coins have the values $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the obvious greedy algorithm always yields an optimal solution.

   (c) Give a set of 4 coin values for which the greedy algorithm does not yield an optimal solution.

   (d) Describe a dynamic programming algorithm that yields an optimal solution for an arbitrary set of coin values.

   (e) Suppose we have only two types of coins whose values a and b are relatively prime. Prove that any value of greater than $(a-1)(b-1)$ can be made with these two coins.

   ★(f) For only three coins a, b, c whose greatest common divisor is 1, give an algorithm to determine the smallest value n such that change *can* be given for all values greater than n. [Note: this problem is currently unsolved for more than four coins!]

6. Suppose you have a subroutine that can find the median of a set of n items (*i.e.*, the $\lfloor n/2 \rfloor$ smallest) in $O(n)$ time. Give an algorithm to find the kth biggest element (for arbitrary k) in $O(n)$ time.

7. You're walking along the beach and you stub your toe on something in the sand. You dig around it and find that it is a treasure chest full of gold bricks of different (integral) weight. Your knapsack can only carry up to weight n before it breaks apart. You want to put as much in it as possible without going over, but you *cannot* break the gold bricks up.

   (a) Suppose that the gold bricks have the weights $1, 2, 4, 8, \ldots, 2^k$, $k \geq 1$. Describe and prove correct a greedy algorithm that fills the knapsack as much as possible without going over.

   (b) Give a set of 3 weight values for which the greedy algorithm does *not* yield an optimal solution and show why.

   (c) Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of gold brick values.

CS 373: Combinatorial Algorithms, Fall 2000

Homework 2 (due September 28, 2000 at midnight)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Faster Longest Increasing Subsequence (15 pts)
   Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. *[Hint: In the dynamic programming solution, you don't really have to look back at all previous items. There was a practice problem on HW 1 that asked for an $O(n^2)$ algorithm for this. If you are having difficulty, look at the HW 1 solutions.]*

2. SELECT(A, k) (10 pts)
   Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.

   (a) Show that a rotation in an augmented binary tree can be performed in constant time.

   (b) Describe an algorithm SCAPEGOATSELECT(k) that selects the kth smallest item in an augmented scapegoat tree in $O(\log n)$ *worst-case* time.

   (c) Describe an algorithm SPLAYSELECT(k) that selects the kth smallest item in an augmented splay tree in $O(\log n)$ *amortized* time.

(d) Describe an algorithm TREAPSELECT(k) that selects the kth smallest item in an augmented treap in $O(\log n)$ *expected* time.

[Hint: The answers for (b), (c), and (d) are almost exactly the same!]

3. Scapegoat trees (15 pts)

   (a) Prove that only one subtree gets rebalanced in a scapegoat tree insertion.

   (b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree. (Recall that $I(v) = \max\{0, |T| - |s| - 1\}$, where T is the child of greater height and s the child of lesser height, and $|v|$ is the number of nodes in subtree v. A perfectly balanced tree has two perfectly balanced subtrees, each with as close to half the nodes as possible.)

   *(c) Show that you can rebuild a fully balanced binary tree from an unbalanced tree in $O(n)$ time using only $O(\log n)$ additional memory. For 5 extra credit points, use only $O(1)$ additional memory.

4. Memory Management (10 pts)
   Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than 1/4 full, we we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method—it makes the problem much too hard!

5. Fibonacci Heaps: SECONDMIN (10 pts)

   (a) Implement SECONDMIN by using a Fibonacci heap as a black box. Remember to justify its correctness and running time.

   *(b) Modify the Fibonacci Heap data structure to implement the SECONDMIN operation in constant time, without degrading the performance of any other Fibonacci heap operation.

Practice Problems

1. Amortization

   (a) Modify the binary double-counter (see class notes Sept 12) to support a new operation SIGN, which determines whether the number being stored is positive, negative, or zero, in constant time. The amortized time to increment or decrement the counter should still be a constant.

   [Hint: Suppose p is the number of significant bits in P, and n is the number of significant bits in N. For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

   *(b) Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

*2. Amortization
Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of 'fits', where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. *[Hint: Most numbers can be represented by more than one fit string. This is not the same representation as on Homework 0!]*

3. Rotations

   (a) Show that it is possible to transform any n-node binary search tree into any other n-node binary search tree using at most $2n - 2$ rotations.

   *(b) Use fewer than $2n - 2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n - 6$ rotations, and there are pairs of trees that are $2n - 10$ rotations apart. These are the best bounds known.

4. Give an efficient implementation of the operation CHANGEKEY(x, k), which changes the key of a node x in a Fibonacci heap to the value k. The changes you make to Fibonacci heap data structure to support your implementation should not affect the amortized running time of any other Fibonacci heap operations. Analyze the amortized running time of your implementation for cases in which k is greater than, less than, or equal to $key[x]$.

5. Detecting overlap

   (a) You are given a list of ranges represented by min and max (*e.g.*, [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$-time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.

(b) You are given a list of rectangles represented by min and max x- and y-coordinates. Give an $O(n \log n)$-time algorithm that decides whet her or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). [Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.]

6. Comparison of Amortized Analysis Methods
A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

(a) Aggregate method

(b) Accounting method

*(c) Potential method

CS 373: Combinatorial Algorithms, Fall 2000

Homework 3 (due October 17, 2000 at midnight)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Suppose you have to design a dictionary that holds 2048 items.

   (a) How many probes are used for an unsuccessful search if the dictionary is implemented as a sorted array? Assume the use of Binary Search.

   (b) How large a hashtable do you need if your goal is to have 2 as the expected number of probes for an unsuccessful search?

   (c) How much more space is needed by the hashtable compared to the sorted array? Assume that each pointer in a linked list takes 1 word of storage.

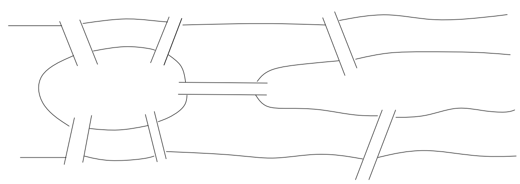
2. In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called 'make' that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of

the other files which are listed. Thus a source file must be recompiled if a file on which it depends is changed.

Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design an algorithm to recompile only those necessary. Don't worry about the details of parsing a Makefile.

3. A person wants to fly from city A to city B in the shortest possible time. She turns to the traveling agent who knows all the departure and arrival times of all the flights on the planet. Give an algorithm that will allow the agent to choose a route with the minimum total travel time—initial takeoff to final landing, including layovers. *[Hint: Modify the data and call a shortest-path algorithm.]*

4. During the eighteenth century the city of Königsberg in East Prussia was divided into four sections by the Pregel river. Seven bridges connected these regions, as shown below. It was said that residents spent their Sunday walks trying to find a way to walk about the city so as to cross each bridge exactly once and then return to their starting point.



   (a) Show how the residents of the city could accomplish such a walk or prove no such walk exists.
   (b) Given any undirected graph $G = (V, E)$, give an algorithm that finds a cycle in the graph that visits every edge exactly once, or says that it can't be done.

5. Suppose you have a graph G and an MST of that graph (i.e. the MST has already been constructed).
   (a) Give an algorithm to update the MST when an edge is added to G.
   (b) Give an algorithm to update the MST when an edge is deleted from G.
   (c) Give an algorithm to update the MST when a vertex (and possibly edges to it) is added to G.

6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

   You are given an unlimited number of each of n different types of envelopes. The dimensions of envelope type i are $x_i \times y_i$. In nesting envelopes inside one another, you can place envelope A inside envelope B if and only if the dimensions A are *strictly smaller* than the dimensions of B. Design and analyze an algorithm to determine the largest number of envelopes that can be nested inside one another.

Practice Problems

★1. Let the hash function for a table of size m be

$$h(x) = \lfloor Amx \rfloor \bmod m$$

where $A = \hat{\phi} = \frac{\sqrt{5}-1}{2}$. Show that this gives the best possible spread, i.e. if the x are hashed in order, $x+1$ will be hashed in the largest remaining contiguous interval.

2. The incidence matrix of an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \big[(i,j) \in E\big] = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{if } (i,j) \notin E. \end{cases}$$

   (a) Describe what all the entries of the matrix product BB^T represent (B^T is the matrix transpose).

   (b) Describe what all the entries of the matrix product $B^T B$ represent.

   ★(c) Let $C = BB^T - 2A$, where A is the adjacency matrix of G, with zeroes on the diagonal. Let C' be C with the first row and column removed. Show that $\det C'$ is the number of spanning trees.

3. (a) Give an $O(V)$ algorithm to decide whether a directed graph contains a *sink* in an adjacency matrix representation. A sink is a vertex with in-degree $V-1$.

   (b) An undirected graph is a scorpion if it has a vertex of degree 1 (the sting) connected to a vertex of degree two (the tail) connected to a vertex of degree $V-2$ (the body) connected to the other $V-3$ vertices (the feet). Some of the feet may be connected to other feet.

   Design an algorithm that decides whether a given adjacency matrix represents a scorpion by examining only $O(V)$ of the entries.

   (c) Show that it is impossible to decide whether G has at least one edge in $O(V)$ time.

4. Given an *undirected* graph $G = (V, E)$, and a weight function $f : E \to \mathbb{R}$ on the *edges*, give an algorithm that finds (in time polynomial in V and E) a cycle of smallest weight in G.

5. Let $G = (V, E)$ be a graph with n vertices. A *simple path* of G, is a path that does not contain the same vertex twice. Use dynamic programming to design an algorithm (not polynomial time) to find a simple path of maximum length in G. Hint: It can be done in $O(n^c 2^n)$ time, for some constant c.

6. Suppose all edge weights in a graph G are equal. Give an algorithm to compute a minimum spanning tree of G.

7. Give an algorithm to construct a *transitive reduction* of a directed graph G, i.e. a graph G^{TR} with the fewest edges (but with the same vertices) such that there is a path from a to b in G iff there is also such a path in G^{TR}.

8. (a) What is $5^{2^{29^{5^0}+23^{4^1}+17^{3^2}+11^{2^3}+5^{1^4}}} \bmod 6$?

   (b) What is the capital of Nebraska? Hint: It is not Omaha. It is named after a famous president of the United States that was not George Washington. The distance from the Earth to the Moon averages roughly 384,000 km.

CS 373: Combinatorial Algorithms, Fall 2000
Homework 4 (due October 26, 2000 at midnight)

Name:		
Net ID:	Alias:	U $^3/_4$ 1

Name:		
Net ID:	Alias:	U $^3/_4$ 1

Name:		
Net ID:	Alias:	U $^3/_4$ 1

Homeworks may be done in teams of up to three people. Each team turns in just one solution, and every memeber of a team gets the same grad. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3/_4$, or 1, respectively. Staple this sheet to the top of your homework.

RequiredProblems

1. (10 points) A certain algorithms professor once claimed that the height of an n-node Fibonacci heap is of height $O(\log n)$. Disprove his claim by showing that for a positive integer n, a sequence of Fibonacci heap operations that creates a Fibonacci heap consisting of just one tree that is a (downward) linear chain of n nodes.

2. (20 points) *Fibonacci strings* are defined as follows:

$$\begin{aligned} F_1 &= \mathsf{b} \\ F_2 &= \mathsf{a} \\ F_n &= F_{n-1}F_{n-2} \quad \text{for all } n > 2 \end{aligned}$$

where the recursive rule uses concatenation of strings, so $F_3 = \mathsf{ab}$, $F_4 = \mathsf{aba}$, and so on. Note that the length of F_n is the nth Fibonacci number.

(a) Prove that in any Fibonacci string there are no two b's adjacent and no three a's.

(b) Give the unoptimized and optimized failure function for F_7.

(c) Prove that, in searching for the Fibonacci string F_k, the unoptimized KMP algorithm may shift $\lceil k/2 \rceil$ times on the same text character. In other words, prove that there is a chain of failure links $j \to fail[j] \to fail[fail[j]] \to \dots$ of length $\lceil k/2 \rceil$, and find an example text T that would cause KMP to traverse this entire chain on the same position in the text.

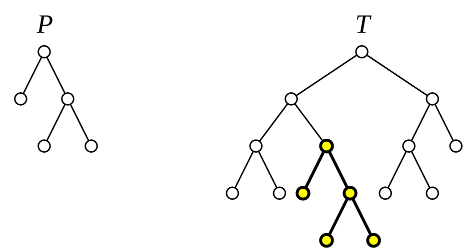
(d) What happens here when you use the optimized prefix function? Explain.

3. (10 points) Show how to extend the Rabin-Karp fingerprinting method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. The pattern may be shifted horizontally and vertically, but it may not be rotated.

4. (10 points)

(a) A *cyclic rotation* of a string is obtained by chopping off a prefix and gluing it at the end of the string. For example, ALGORITHM is a cyclic shift of RITHMALGO. Describe and analyze an algorithm that determines whether one string $P[1..m]$ is a cyclic rotation of another string $T[1..n]$.

(b) Describe and analyze an algorithm that decides, given any two binary trees P and T, whether P equals a subtree of T. We want an algorithm that compares the *shapes* of the trees. There is no data stored in the nodes, just pointers to the left and right children. *[Hint: First transform both trees into strings.]*



P occurs exactly once as a subtree of T.

5. (10 points) *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

Refer to the notes for lecture 11 for this problem. The GENERICSSSP algorithm described in class can be implemented using a stack for the 'bag'. Prove that the resulting algorithm can be forced to perform in $\Omega(2^n)$ relaxation steps. To do this, you need to describe, for any positive integer n, a specific weighted directed n-vertex graph that forces this exponential behavior. The easiest way to describe such a family of graphs is using an *algorithm*!

Practice Problems

1. String matching with wild-cards
   Suppose you have an alphabet for patterns that includes a 'gap' or wild-card character; any length string of any characters can match this additional character. For example if '*' is the wild-card, then the pattern foo*bar*nad can be found in foofoowangbarnad. Modify the computation of the prefix function to correctly match strings using KMP.

2. Prove that there is no comparison sort whose running time is linear for at least 1/2 of the $n!$ inputs of length n. What about at least $1/n$? What about at least $1/2^n$?.

3. Prove that $2n - 1$ comparisons are necessary in the worst case to merge two sorted lists containing n elements each.

4. Find asymptotic upper and lower bounds to $\lg(n!)$ without Stirling's approximation (Hint: use integration).

5. Given a sequence of n elements of n/k blocks (k elements per block) all elements in a block are less than those to the right in sequence, show that you cannot have the whole sequence sorted in better than $\Omega(n \lg k)$. Note that the entire sequence would be sorted if each of the n/k blocks were individually sorted in place. Also note that combining the lower bounds for each block is not adequate (that only gives an upper bound).

6. Show how to find the occurrences of pattern P in text T by computing the prefix function of the string PT (the concatenation of P and T).

7. Lower Bounds on Adjacency Matrix Representations of Graphs

   (a) Prove that the time to determine if an undirected graph has a cycle is $\Omega(V^2)$.

   (b) Prove that the time to determine if there is a path between two nodes in an undirected graph is $\Omega(V^2)$.

CS 373: Combinatorial Algorithms, Fall 2000
Homework 1 (due November 16, 2000 at midnight)

Name:		
Net ID:	Alias:	U $^{3}\!/_{4}$ 1

Name:		
Net ID:	Alias:	U $^{3}\!/_{4}$ 1

Name:		
Net ID:	Alias:	U $^{3}\!/_{4}$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^{3}\!/_{4}$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Give an $O(n^2 \log n)$ algorithm to determine whether any three points of a set of n points are collinear. Assume two dimensions and *exact* arithmetic.

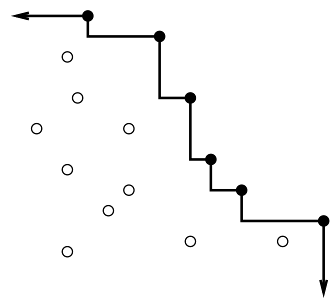
2. We are given an array of n bits, and we want to determine if it contains two consecutive 1 bits. Obviously, we can check every bit, but is this always necessary?

   (a) (4 pts) Show that when $n \bmod 3 = 0$ or 2, we must examine every bit in the array. that is, give an adversary strategy that forces any algorithm to examine every bit when $n = 2, 3, 5, 6, 8, 9, \ldots$.

   (b) (4 pts) Show that when $n = 3k + 1$, we only have to examine $n - 1$ bits. That is, describe an algorithm that finds two consecutive 1s or correctly reports that there are none after examining at most $n - 1$ bits, when $n = 1, 4, 7, 10, \ldots$.

   (c) (2 pts) How many n-bit strings are there with two consecutive ones? For which n is this number even or odd?

3. You are given a set of points in the plane. A point is *maximal* if there is no other point both both above and to the right. The subset of maximal points of points then forms a *staircase*.



The staircase of a set of points. Maximal points are black.

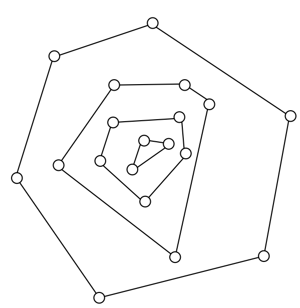
   (a) (0 pts) Prove that maximal points are *not* necessarily on the convex hull.

   (b) (6 pts) Give an $O(n \log n)$ algorithm to find the maximal points.

   (c) (4 pts) Assume that points are chosen uniformly at random within a rectangle. What is the average number of maximal points? Justify. Hint: you will be able to give an exact answer rather than just asymptotics. You have seen the same analysis before.

4. Given a set Q of points in the plane, define the *convex layers* of Q inductively as follows: The first convex layer of Q is just the convex hull of Q. For all $i > 1$, the ith convex layer is the convex hull of Q after the vertices of the first $i - 1$ layers have been removed.

   Give an $O(n^2)$-time algorithm to find all convex layers of a given set of n points.



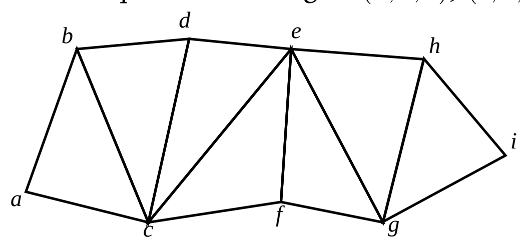
A set of points with four convex layers.

5. Prove that finding the second smallest of n elements takes $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. Prove for both upper and lower bounds. Hint: find the (first) smallest using an elimination tournament.

6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

   Almost all computer graphics systems, at some level, represent objects as collections of triangles. In order to minimize storage space and rendering time, many systems allow objects to be stored as a set of *triangle strips*. A triangle strip is a sequence of vertices $\langle v_1, v_2, \ldots, v_k \rangle$, where each contiguous triple of vertices v_i, v_{i+1}, v_{i+2} represents a triangle. As the rendering system reads the sequence of vertices and draws the triangles, it keeps the two most recent vertices in a cache.

   Some systems allow triangle strips to contain *swaps*: special flags indicating that the order of the two cached vertices should be reversed. For example, the triangle strip $\langle a, b, c, d, \mathsf{swap}, e, f, \mathsf{swap}, g, h, i \rangle$ represents the sequence of triangles (a, b, c), (b, c, d), (d, c, e), (c, e, f), (f, e, g).



   Two triangle strips are *disjoint* if they share no triangles (although they may share vertices). The *length* of a triangle strip is the length of its vertex sequence, including swaps; for example, the example strip above has length 11. A *pure* triangle strip is one with no swaps. The adjacency graph of a triangle strip is a simple path. If the strip is pure, this path alternates between left and right turns.

   Suppose you are given a set S of interior-disjoint triangles whose adjacency graph is a tree. (In other words, S is a triangulation of a simple polygon.) Describe a linear-time algorithm to decompose S into a set of disjoint triangle strips of minimum total length.

Practice Problems

1. Consider the following generic recurrence for convex hull algorithms that divide and conquer:

$$T(n, h) = T(n_1, h_1) + T(n_2, h_2) + O(n)$$

   where $n \geq n_1 + n_2, h = h_1 + h_2$ and $n \geq h$. This means that the time to compute the convex hull is a function of both n, the number of input points, and h, the number of convex hull vertices. The splitting and merging parts of the divide-and-conquer algorithm take $O(n)$ time. When n is a constant, $T(n, h) = O(1)$, but when h is a constant, $T(n, h) = O(n)$. Prove that for both of the following restrictions, the solution to the recurrence is $O(n \log h)$:

   (a) $h_1, h_2 < \frac{3}{4}h$

   (b) $n_1, n_2 < \frac{3}{4}n$

2. Circle Intersection
   Give an $O(n \log n)$ algorithm to test whether any two circles in a set of size n intersect.

3. Basic polygon computations (assume *exact* arithmetic)

   (a) Intersection: Extend the basic algorithm to determine if two line segments intersect by taking care of *all* degenerate cases.

   (b) Simplicity: Give an $O(n \log n)$ algorithm to determine whether an n-vertex polygon is simple.

   (c) Area: Give an algorithm to compute the area of a simple n-polygon (not necessarily convex) in $O(n)$ time.

   (d) Inside: Give an algorithm to determine whether a point is within a simple n-polygon (not necessarily convex) in $O(n)$ time.

4. We are given the set of points one point at a time. After receiving each point, we must compute the convex hull of all those points so far. Give an algorithm to solve this problem in $O(n^2)$ total time. (We could obviously use Graham's scan n times for an $O(n^2 \log n)$-time algorithm). Hint: How do you maintain the convex hull?

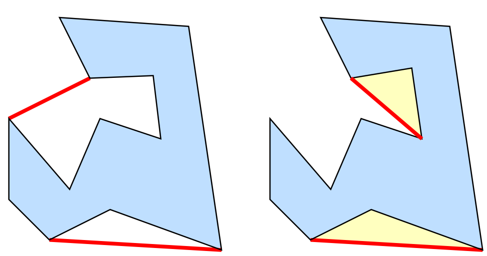
5. *(a) Given an n-polygon and a point outside the polygon, give an algorithm to find a tangent.

   (b) Suppose you have found both tangents. Give an algorithm to remove the points from the polygon that are within the angle formed by the tangents (as segments!) and the opposite side of the polygon.

   (c) Use the above to give an algorithm to compute the convex hull on-line in $O(n \log n)$

6. (a) A pair of polygon vertices defines an *external diagonal* if the line segment between them is completely outside the polygon. Show that every nonconvex polygon has at least one external diagonal.

   (b) Three consective polygon vertices p, q, r form a *mouth* if p and r define an external diagonal. Show that every nonconvex polygon has at least one mouth.



Two external diagonals Two mouths

7. A group of n ghostbusters is battling n ghosts. Each ghostbuster can shoot a single energy beam at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The ghostbusters all fire at the same time and no two energy beams may cross. The positions of the ghosts and ghostbusters are fixed points in the plane.

   (a) Prove that for any configuration of ghosts and ghostbusters, there is such a non-crossing matching. (Assume that no three points are collinear.)

(b) Show that there is a line passing through one ghostbuster and one ghost such that the number of ghostbusters on one side of the line equals the number of ghosts on the same side. Give an efficient algorithm to find such a line.

(c) Give an efficient divide and conquer algorithm to pair ghostbusters and ghosts so that no two streams cross.

CS 373: Combinatorial Algorithms, Fall 2000
Homework 6 (due December 7, 2000 at midnight)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. (a) Prove that P $\subseteq$ co-NP.

   (b) Show that if NP $\neq$ co-NP, then *no* NP-complete problem is a member of co-NP.

2. 2SAT is a special case of the formula satisfiability problem, where the input formula is in conjunctive normal form and every clause has at most *two* literals. Prove that 2SAT is in P.

3. Describe an algorithm that solves the following problem, called 3SUM, as quickly as possible: Given a set of n numbers, does it contain three elements whose sum is zero? For example, your algorithm should answer TRUE for the set $\{-5, -17, 7, -4, 3, -2, 4\}$, since $-5+7+(-2) = 0$, and FALSE for the set $\{-6, 7, -4, -13, -2, 5, 13\}$.

4. (a) Show that the problem of deciding whether one undirected graph is a subgraph of another is NP-complete.
   (b) Show that the problem of deciding whether an unweighted undirected graph has a path of length greater than k is NP-complete.

5. (a) Consider the following problem: Given a set of axis-aligned rectangles in the plane, decide whether any point in the plane is covered by k or more rectangles. Now also consider the CLIQUE problem. Describe and analyze a reduction of one problem to the other.
   (b) Finding the largest clique in an arbitrary graph is NP-hard. What does this fact imply about the complexity of finding a point that lies inside the largest number of rectangles?

6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

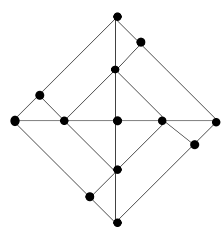
   PARTITION is the problem of deciding, given a set $S = \{s_1, s_2, \ldots, s_n\}$ of numbers, whether there is a subset T containing half the 'weight' of S, *i.e.*, such that $\sum T = \frac{1}{2} \sum S$. SUBSETSUM is the problem of deciding, given a set $S = \{s_1, s_2, \ldots, s_n\}$ of numbers and a target sum t, whether there is a subset $T \subseteq S$ such that $\sum T = t$. Give two reductions between these two problems, one in each direction.

Practice Problems

1. What is the *exact* worst case number of comparisons needed to find the median of 5 numbers? For 6 numbers?

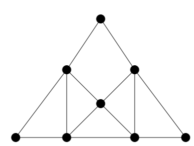
2. The ExactCoverByThrees problem is defined as follows: given a finite set X and a collection C of 3-element subsets of X, does C contain an *exact cover* for X, that is, a subcollection $C' \subseteq C$ where every element of X occurs in exactly one member of C'? Given that ExactCoverByThrees is NP-complete, show that the similar problem ExactCoverByFours is also NP-complete.

3. Using 3Color and the 'gadget' below, prove that the problem of deciding whether a planar graph can be 3-colored is NP-complete. *[Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]*



Crossing gadget for Planar3Color.

4. Using the previous result, and the 'gadget' below, prove that the problem of deciding whether a planar graph with no vertex of degree greater than four can be 3-colored is NP-complete. *[Hint: Show that you can replace any vertex with degree greater than 4 with a collection of gadgets connected in such a way that no degree is greater than four.]*



Degree gadget for Degree4Planar3Color

5. Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

6. (a) Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian. Give a polynomial time algorithm algorithm for finding a hamiltonian cycle in an undirected bipartite graph or establishing that it does not exist.
   (b) Show that the hamiltonian-path problem can be solved in polynomial time on directed acyclic graphs.
   (c) Explain why the results in previous questions do not contradict the fact that both HamiltonianCycle and HamiltonianPath are NP-complete problems.

7. Consider the following pairs of problems:

(a) MIN SPANNING TREE and MAX SPANNING TREE

(b) SHORTEST PATH and LONGEST PATH

(c) TRAVELING SALESMAN and VACATION TOUR (the longest tour is sought).

(d) MIN CUT and MAX CUT (between s and t)

(e) EDGE COVER and VERTEX COVER

(f) TRANSITIVE REDUCTION and MIN EQUIVALENT DIGRAPH

(All of these seem dual or opposites, except the last, which are just two versions of minimal representation of a graph.) Which of these pairs are polytime equivalent and which are not?

★8. Consider the problem of deciding whether one graph is isomorphic to another.

(a) Give a brute force algorithm to decide this.

(b) Give a dynamic programming algorithm to decide this.

(c) Give an efficient probabilistic algorithm to decide this.

★(d) Either prove that this problem is NP-complete, give a poly time algorithm for it, or prove that neither case occurs.

*9. Prove that PRIMALITY (Given n, is n prime?) is in NP $\cap$ co-NP. Showing that PRIMALITY is in co-NP is easy. (What's a certificate for showing that a number is composite?) For NP, consider a certificate involving primitive roots and recursively their primitive roots. Show that this tree of primitive roots can be checked to be correct and used to show that n is prime, and that this check takes polynomial time.

10. How much wood would a woodchuck chuck if a woodchuck could chuck wood?

CS 373: Combinatorial Algorithms, Fall 2000

Midterm 1 — October 3, 2000

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

This is a closed-book, closed-notes exam!

If you brought anything with you besides writing instruments and your $8\frac{1}{2}'' \times 11''$ cheat sheet, please leave it at the front of the classroom.

- Print your name, netid, and alias in the boxes above. Circle U if you are an undergrad, $^3\!/_4$ if you are a 3/4-unit grad student, or 1 if you are a 1-unit grad student. Print your name at the top of every page (in case the staple falls out!).
- **Answer four of the five questions on the exam.** Each question is worth 10 points. If you answer more than four questions, the one with the lowest score will be ignored. **1-unit graduate students must answer question 5.**
- Please write your final answers on the front of the exam pages. Use the backs of the pages as scratch paper. Let us know if you need more paper.
- Unless we specifically say otherwise, proofs are not required. However, they may help us give you partial credit.
- Read the entire exam before writing anything. Make sure you understand what the questions are asking. If you give a beautiful answer to the wrong question, you'll get no credit. If any question is unclear, please ask one of us for clarification.
- Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.
- Write something down for every problem. Don't panic and erase large chunks of work. Even if you think it's absolute nonsense, it might be worth partial credit.
- Relax. Breathe. Kick some ass.

#	Score	Grader
1		
2		
3		
4		
5		
Total		

1. **Multiple Choice**

   Every question below has one of the following answers.

   (a) $\Theta(1)$ (b) $\Theta(\log n)$ (c) $\Theta(n)$ (d) $\Theta(n \log n)$ (e) $\Theta(n^2)$

   For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point, but each incorrect answer *costs* you $\frac{1}{2}$ point. You cannot score below zero.

☐ What is $\sum_{i=1}^{n} \log i$?

☐ What is $\sum_{i=1}^{n} \frac{n}{i}$?

☐ How many digits do you need to write 2^n in decimal?

☐ What is the solution of the recurrence $T(n) = 25T(n/5) + n$?

☐ What is the solution of the recurrence $T(n) = T(n-1) + \frac{1}{2^n}$?

☐ What is the solution of the recurrence $T(n) = 3T\left(\left\lceil \frac{n+51}{3} \right\rceil\right) + 17n - \sqrt[7]{\lg \lg n} - 2^{2^{\log^* n}} + \pi$?

☐ What is the worst-case running time of randomized quicksort?

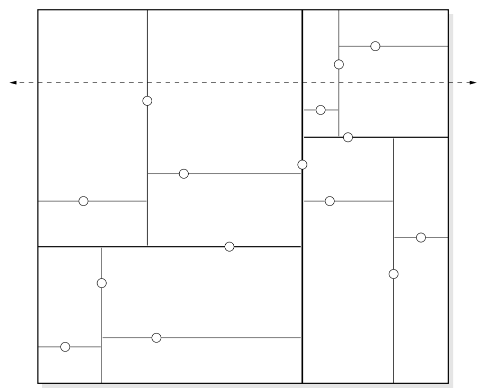
☐ The expected time for inserting one item into an n-node randomized treap is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty treap?

☐ The amortized time for inserting one item into an n-node scapegoat tree is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty scapegoat tree?

☐ In the worst case, how many nodes can be in the root list of a Fibonacci heap storing n keys, immediately after a DECREASEKEY operation?

☐ Every morning, an Amtrak train leaves Chicago for Champaign, 200 miles away. The train can accelerate or decelerate at 10 miles per hour per second, and it has a maximum speed of 60 miles an hour. Every 50 miles, the train must stop for five minutes while a school bus crosses the tracks. Every hour, the conductor stops the train for a union-mandated 10-minute coffee break. How long does it take the train to reach Champaign?

2. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the rectangle as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line passes through some point inside the box (*not* on the boundary) and partitions the rest of the interior points as evenly as possible. If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



A kd-tree for 15 points. The dashed line crosses four cells.

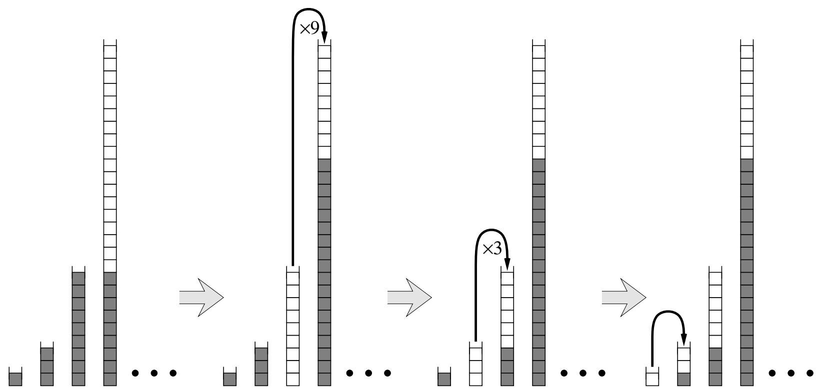
   (a) *[2 points]* How many cells are there, as a function of n? Prove your answer is correct.

   (b) *[8 points]* In the worst case, exactly how many cells can a horizontal line cross, as a function of n? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[For full credit, you must give an exact answer. A tight asymptotic bound (with proof) is worth 5 points. A correct recurrence is worth 3 points.]*

   (c) *[5 points extra credit]* In the worst case, how many cells can a *diagonal* line cross?

Incidentally, 'kd-tree' originally meant 'k-dimensional tree'—for example, the specific data structure described here used to be called a '2d-tree'—but current usage ignores this etymology. The phrase 'd-dimensional kd-tree' is now considered perfectly standard, even though it's just as redundant as 'ATM machine', 'PIN number', 'HIV virus', 'PDF format', 'Mt. Fujiyama', 'Sahara Desert', 'The La Brea Tar Pits', or 'and etc.' On the other hand, 'BASIC code' is *not* redundant; 'Beginner's All-Purpose Instruction Code' is a backronym. Hey, aren't you supposed to be taking a test?

3. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i, we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2}, and so on. Moving a single element from one stack to the next takes $O(1)$ time.



Making room for one new element in a multistack.

(a) *[1 point]* In the worst case, how long does it take to push one more element onto a multistack containing n elements?

(b) *[9 points]* Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack. You can use any method you like.

4. After graduating with a computer science degree, you find yourself working for a software company that publishes a word processor. The program stores a document containing n characters, grouped into p paragraphs. Your manager asks you to implement a 'Sort Paragraphs' command that rearranges the paragraphs into alphabetical order.

   Design and analyze and efficient paragraph-sorting algorithm, using the following pair of routines as black boxes.

   - COMPAREPARAGRAPHS(i, j) compares the ith and jth paragraphs, and returns i or j depending on which paragraph should come first in the final sorted output. (Don't worry about ties.) This function runs in $O(1)$ time, since almost any two paragraphs can be compared by looking at just their first few characters!
   - MOVEPARAGRAPH(i, j) 'cuts' out the ith paragraph and 'pastes' it back in as the jth paragraph. This function runs in $O(n_i)$ time, where n_i is the number of characters in the ith paragraph. (So in particular, $n_1 + n_2 + \cdots + n_p = n$.)

     Here is an example of MOVEPARAGRAPH$(7, 2)$:

Congress shall make no law respecting...		Congress shall make no law respecting...
No soldier shall, in time of peace...		**A well regulated militia, being...**
The right of the people to be secure...		No soldier shall, in time of peace...
No person shall be held to answer for...		The right of the people to be secure...
In all criminal prosecutions, the...	$\Longrightarrow$	No person shall be held to answer for...
In suits at common law, where the...		In all criminal prosecutions, the...
A well regulated militia, being...		In suits at common law, where the...
Excessive bail shall not be required...		Excessive bail shall not be required...
The enumeration in the Constitution...		The enumeration in the Constitution...
The powers not delegated to the...		The powers not delegated to the...

   [Hint: For full credit, your algorithm should run in $o(n \log n)$ time when $p = o(n)$.]

5. *[1-unit grad students must answer this question.]*

   Describe and analyze an algorithm to randomly shuffle an array of n items, so that each of the $n!$ possible permutations is equally likely. Assume that you have a function RANDOM(i, j) that returns a random integer from the set $\{i, i+1, \ldots, j\}$ in constant time.

   [Hint: As a sanity check, you might want to confirm that for $n = 3$, all six permutations have probability $1/6$. For full credit, your algorithm must run in $\Theta(n)$ time. A correct algorithm that runs in $\Theta(n \log n)$ time is worth 7 points.]

From: "Josh Pepper" <jwpepper@uiuc.edu>
To: "Chris Neihengen" <neihenge@uiuc.edu>
Subject: FW: proof
Date: Fri, 29 Sep 2000 09:34:56 -0500

thought you might like this.

Problem: To prove that computer science 373 is indeed the work of Satan.

Proof: First, let us assume that everything in "Helping Yourself with Numerology", by Helyn Hitchcock, is true.

Second, let us apply divide and conquer to this problem. There are main parts:

1. The name of the course: "Combinatorial Algorithms"
2. The most important individual in the course, the "Recursion Fairy"
3. The number of this course: 373.

We examine these sequentially.

The name of the course. "Combinatorial Algorithms" can actually be expressed as a single integer - 23 - since it has 23 letters. The most important individual, the Recursion Fairy, can also be expressed as a single integer - 14 - since it has 14 letters. In other words:

    COMBINATORIAL ALGORITHMS = 23
    RECURSION FAIRY = 14

As a side note, a much shorter proof has already been published showing that the Recursion Fairy is Lucifer, and that any class involving the Fairy is from Lucifer, however, that proofs numerological significance is slight.

Now we can move on to an analysis of the number of course, which holds great meaning. The first assumtion we make is that the number of the course, 373, is not actually a base 10 number. We can prove this inductively by making a reasonable guess for the actual base, then finding a new way to express the nature of the course, and if the answer

confirms what we assumed, then we're right. That's the way induction works.

What is a reasonable guess for the base of the course? The answer is trivial, since the basest of all beings is the Recursion Fairy, the base is 14. So a true base 10 representation of 373 (base 14) is 689. So we see:

    373 (base 14) = 689 (base 10)

Now since the nature of the course has absolutely nothing to do with combinatorial algorithms (instead having much to do with the work of the devil), we can subtract from the above result everything having to do with combinatorial algorithms just by subtracting 23. Here we see that:

    689 - 23 = 666

QED.

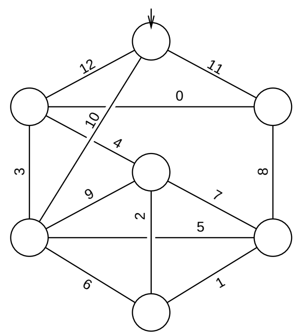
1. Using any method you like, compute the following subgraphs for the weighted graph below. Each subproblem is worth 3 points. Each incorrect edge costs you 1 point, but you cannot get a negative score for any subproblem.

   (a) a depth-first search tree, starting at the top vertex;

   (b) a breadth-first search tree, starting at the top vertex;

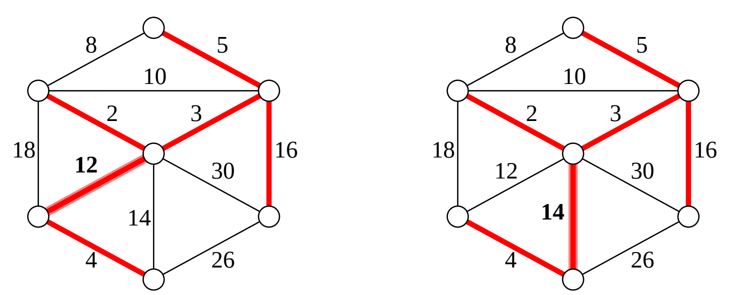
   (c) a shortest path tree, starting at the top vertex;

   (d) the minimum spanning tree.



2. Suppose you are given a weighted undirected graph G (represented as an adjacency list) and its minimum spanning tree T (which you already know how to compute). Describe and analyze and algorithm to find the *second-minimum spanning tree* of G, *i.e.*, the spanning tree of G with smallest total weight except for T.

   The minimum spanning tree and the second-minimum spanning tree differ by exactly one edge. But *which* edge is different, and *how* is it different? That's what your algorithm has to figure out!



The minimum spanning tree and the second-minimum spanning tree of a graph.

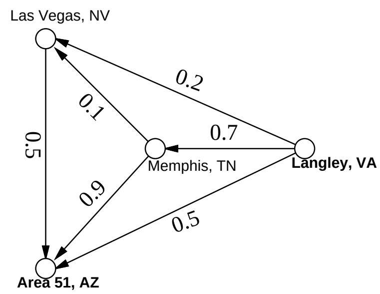
3. (a) *[4 pts]* Prove that a connected acyclic graph with V vertices has exactly $V-1$ edges. ("It's a tree!" is not a proof.)

   (b) *[4 pts]* Describe and analyze an algorithm that determines whether a given graph is a tree, where the graph is represented by an adjacency list.

   (c) *[2 pts]* What is the running time of your algorithm from part (b) if the graph is represented by an adjacency matrix?

4. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

   More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.



   With the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted![1]

5. *[1-unit grad students must answer this question.]*

   Many string matching applications allow the following *wild card* characters in the pattern.

   - The wild card `?` represents an arbitrary single character. For example, the pattern `s?r?ng` matches the strings `string`, `sprung`, and `sarong`.
   - The wild card `*` represents an arbitrary string of zero or more characters. For example, the pattern `te*st*` matches the strings `test`, `tensest`, and `technostructuralism`.

   Both wild cards can occur in a single pattern. For example, the pattern `f*a??` matches the strings `face`, `football`, and `flippityfloppitydingdongdang`. On the other hand, neither wild card can occur in the text.

   Describe how to modify the Knuth-Morris-Pratt algorithm to support patterns with these wild cards, and analyze the modified algorithm. Your algorithm should find the first substring in the text that matches the pattern. An algorithm that supports only one of the two wild cards is worth 5 points.

[1]That's how they got Elvis, you know.

1. **True, False, or Maybe**

   Indicate whether each of the following statments is always true, sometimes true, always false, or unknown. Some of these questions are deliberately tricky, so read them carefully. Each correct choice is worth $+1$, and each incorrect choice is worth -1. **Guessing will hurt you!**

   (a) Suppose SMARTALGORITHM runs in $\Theta(n^2)$ time and DUMBALGORITHM runs in $\Theta(2^n)$ time for all inputs of size n. (Thus, for each algorithm, the best-case and worst-case running times are the same.) SMARTALGORITHM is faster than DUMBALGORITHM.

   ☐ True ☐ False ☐ Sometimes ☐ Nobody Knows

   (b) QUICKSORT runs in $O(n^6)$ time.

   ☐ True ☐ False ☐ Sometimes ☐ Nobody Knows

   (c) $\lfloor \log_2 n \rfloor \geq \lceil \log_2 n \rceil$

   ☐ True ☐ False ☐ Sometimes ☐ Nobody Knows

   (d) The recurrence $F(n) = n + 2\sqrt{n} \cdot F(\sqrt{n})$ has the solution $F(n) = \Theta(n \log n)$.

   ☐ True ☐ False ☐ Sometimes ☐ Nobody Knows

   (e) A Fibonacci heap with n nodes has depth $\Omega(\log n)$.

   ☐ True ☐ False ☐ Sometimes ☐ Nobody Knows

   (f) Suppose a graph G is represented by an adjacency matrix. It is possible to determine whether G is an independent set without looking at every entry of the adjacency matrix.

   ☐ True ☐ False ☐ Sometimes ☐ Nobody Knows

   (g) NP $\neq$ co-NP

   ☐ True ☐ False ☐ Sometimes ☐ Nobody Knows

   (h) Finding the smallest clique in a graph is NP-hard.

   ☐ True ☐ False ☐ Sometimes ☐ Nobody Knows

   (i) A polynomial-time reduction from X to 3SAT proves that X is NP-hard.

   ☐ True ☐ False ☐ Sometimes ☐ Nobody Knows

   (j) The correct answer for exactly three of these questions is “False”.

   ☐ True ☐ False

2. **Convex Hull**

   Suppose you are given the convex hull of a set of n points, and one additional point (x, y). The convex hull is represented by an array of vertices in counterclockwise order, starting from the leftmost vertex. Describe how to test in $O(\log n)$ time whether or not the additional point (x, y) is inside the convex hull.

3. **Finding the Largest Block**

   In your new job, you are working with screen images. These are represented using two dimensional arrays where each element is a 1 or a 0, indicating whether that position of the screen is illuminated. Design and analyze an efficient algorithm to find the largest rectangular block of ones in such an array. For example, the largest rectangular block of ones in the array shown below is in rows 2–4 and columns 2–3. *[Hint: Use dynamic programming.]*

1	0	1	0	0
1	1	1	0	1
0	1	1	1	1
1	1	1	0	0

4. **The Hogwarts Sorting Hat**

   Every year, upon their arrival at Hogwarts School of Witchcraft and Wizardry, new students are sorted into one of four houses (Gryffindor, Hufflepuff, Ravenclaw, or Slytherin) by the Hogwarts Sorting Hat. The student puts the Hat on their head, and the Hat tells the student which house they will join. This year, a failed experiment by Fred and George Weasley filled almost all of Hogwarts with sticky brown goo, mere moments before the annual Sorting. As a result, the Sorting had to take place in the basement hallways, where there was so little room to move that the students had to stand in a long line.

   After everyone learned what house they were in, the students tried to group together by house, but there was too little room in the hallway for more than one student to move at a time. Fortunately, the Sorting Hat took CS 373 many years ago, so it knew how to group the students as quickly as possible. What method did the Sorting Hat use?

   More formally, you are given an array of n items, where each item has one of four possible values, possibly with a pointer to some additional data. Design and analyze an algorithm that rearranges the items into four clusters in $O(n)$ time using only $O(1)$ extra space.

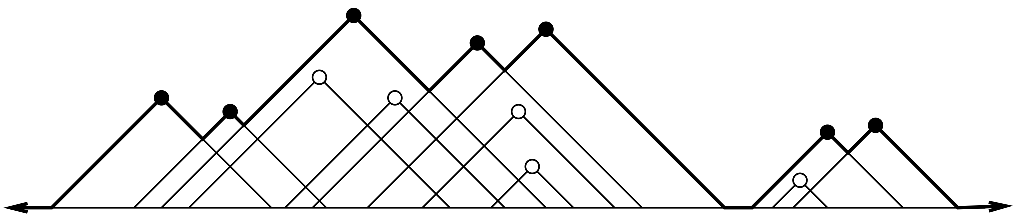
G	H	R	R	G	G	R	G	H	H	R	S	R	R	H	G	S	H	G	G
Harry	Ann	Bob	Tina	Chad	Bill	Lisa	Ekta	Bart	Jim	John	Jeff	Liz	Mary	Dawn	Nick	Kim	Fox	Dana	Mel

$$\Downarrow$$

G	G	G	G	G	G	G	H	H	H	H	H	R	R	R	R	R	R	S	S
Harry	Ekta	Bill	Chad	Nick	Mel	Dana	Fox	Ann	Jim	Dawn	Bart	Lisa	Tina	John	Bob	Liz	Mary	Kim	Jeff

5. **The Egyptian Skyline**

Suppose you are given a set of n pyramids in the plane. Each pyramid is a isosceles triangle with two $45°$ edges and a horizontal edge on the x-axis. Each pyramid is represented by the x- and y-coordinates of its topmost point. Your task is to compute the "skyline" formed by these pyramids (the dark line shown below).



The skyline formed by these 12 pyramids has 16 vertices.

(a) Describe and analyze an algorithm that determines which pyramids are visible on the skyline. These are the pyramids with black points in the figure above; the pyramids with white points are not visible. *[Hint: You've seen this problem before.]*

(b) One you know which pyramids are visible, how would you compute the *shape* of the skyline? Describe and analyze an algorithm to compute the left-to-right sequence of skyline vertices, including the vertices between the pyramids and on the ground.

6. **DNF-SAT**

A boolean formula is in *disjunctive normal form* (DNF) if it consists of clauses of conjunctions (ANDs) joined together by disjunctions (ORs). For example, the formula

$$(\overline{a} \wedge b \wedge \overline{c}) \vee (b \wedge c) \vee (a \wedge \overline{b} \wedge \overline{c})$$

is in disjunctive normal form. DNF-SAT is the problem that asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

(a) Show that DNF-SAT is in P.

(b) What is wrong with the following argument that P=NP?

> Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,
>
> $$(a \vee b \vee \overline{c}) \wedge (\overline{a} \vee \overline{b}) \iff (a \wedge \overline{b}) \vee (b \wedge \overline{a}) \vee (\overline{c} \wedge \overline{a}) \vee (\overline{c} \wedge \overline{b})$$
>
> Now we can use the answer to part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time! Since 3SAT is NP-hard, we must conclude that P=NP.

7. **Magic 3-Coloring** *[1-unit graduate students must answer this question.]*

The recursion fairy's distant cousin, the reduction genie, shows up one day with a magical gift for you—a box that determines in constant time whether or not a graph is 3-colorable. (A graph is 3-colorable if you can color each of the vertices red, green, or blue, so that every edge has do different colors.) The magic box does not tell you *how* to color the graph, just whether or not it can be done. Devise and analyze an algorithm to 3-color any graph **in polynomial time** using this magic box.

CS 373: Combinatorial Algorithms, Spring 2001

Homework 0, due January 23, 2001 at the beginning of class

Name:	
Net ID:	Alias:

Neatly print your name (first name first, with no comma), your network ID, and a short alias into the boxes above. **Do not sign your name. Do not write your Social Security number.** Staple this sheet of paper to the top of your homework.

Grades will be listed on the course web site by alias give us, so your alias should not resemble your name or your Net ID. If you don't give yourself an alias, we'll give you one that you won't like.

This homework tests your familiarity with the prerequisite material from CS 173, CS 225, and CS 273—many of these problems have appeared on homeworks or exams in those classes—primarily to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Parberry and Chapters 1–6 of CLR should be sufficient review, but you may want to consult other texts as well.

Before you do anything else, read the Homework Instructions and FAQ on the CS 373 course web page (http://www-courses.cs.uiuc.edu/~cs373/hw/faq.html), and then check the box below. This web page gives instructions on how to write and submit homeworks—staple your solutions together in order, write your name and netID on every page, don't turn in source code, analyze everything, use good English and good logic, and so forth.

☐ I have read the CS 373 Homework Instructions and FAQ.

Required Problems

1. (a) Prove that any positive integer can be written as the sum of distinct powers of 2. For example: $42 = 2^5 + 2^3 + 2^1$, $25 = 2^4 + 2^3 + 2^0$, $17 = 2^4 + 2^0$. *[Hint: 'Write the number in binary' is* not *a proof; it just restates the problem.]*

   (b) Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.

   (c) Prove that *any* integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example: $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.

2. Sort the following 20 functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice.

$$\begin{array}{ccccc} 1 & n & n^2 & \lg n & \lg^* n \\ 2^{2^{\lg\lg n+1}} & \lg^* 2^n & 2^{\lg^* n} & \lfloor \lg(n!) \rfloor & \lfloor \lg n \rfloor ! \\ n^{\lg n} & (\lg n)^n & (\lg n)^{\lg n} & n^{1/\lg n} & n^{\lg\lg n} \\ \log_{1000} n & \lg^{1000} n & \lg^{(1000)} n & \left(1+\frac{1}{1000}\right)^n & n^{1/1000} \end{array}$$

To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2, n, $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

3. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied. Extra credit will be given for more exact solutions.

   (a) $A(n) = 5A(n/3) + n \log n$

   (b) $B(n) = \min\limits_{0<k<n} \big(B(k) + B(n-k) + 1\big).$

   (c) $C(n) = 4C(\lfloor n/2 \rfloor + 5) + n^2$

   (d) $D(n) = D(n-1) + 1/n$

   *(e) $E(n) = n + 2\sqrt{n} \cdot E(\sqrt{n})$

4. This problem asks you to simplify some recursively defined boolean formulas as much as possible. In each case, prove that your answer is correct. Each proof can be just a few sentences long, but it must be a *proof.*

   (a) Suppose $\alpha_0 = p$, $\alpha_1 = q$, and $\alpha_n = (\alpha_{n-2} \wedge \alpha_{n-1})$ for all $n \geq 2$. Simplify α_n as much as possible. *[Hint: What is α_5?]*

   (b) Suppose $\beta_0 = p$, $\beta_1 = q$, and $\beta_n = (\beta_{n-2} \Leftrightarrow \beta_{n-1})$ for all $n \geq 2$. Simplify β_n as much as possible. *[Hint: What is β_5?]*

   (c) Suppose $\gamma_0 = p$, $\gamma_1 = q$, and $\gamma_n = (\gamma_{n-2} \Rightarrow \gamma_{n-1})$ for all $n \geq 2$. Simplify γ_n as much as possible. *[Hint: What is γ_5?]*

   (d) Suppose $\delta_0 = p$, $\delta_1 = q$, and $\delta_n = (\delta_{n-2} \bowtie \delta_{n-1})$ for all $n \geq 2$, where $\bowtie$ is some boolean function with two arguments. Find a boolean function $\bowtie$ such that $\delta_n = \delta_m$ if and only if $n - m$ is a multiple of 4. *[Hint: There is only one such function.]*

5. Every year, upon their arrival at Hogwarts School of Witchcraft and Wizardry, new students are sorted into one of four houses (Gryffindor, Hufflepuff, Ravenclaw, or Slytherin) by the Hogwarts Sorting Hat. The student puts the Hat on their head, and the Hat tells the student which house they will join. This year, a failed experiment by Fred and George Weasley filled almost all of Hogwarts with sticky brown goo, mere moments before the annual Sorting. As a result, the Sorting had to take place in the basement hallways, where there was so little room to move that the students had to stand in a long line.

   After everyone learned what house they were in, the students tried to group together by house, but there was too little room in the hallway for more than one student to move at a time. Fortunately, the Sorting Hat took CS 373 many years ago, so it knew how to group the students as quickly as possible. What method did the Sorting Hat use?

   More formally, you are given an array of n items, where each item has one of four possible values, possibly with a pointer to some additional data. Describe an algorithm[1] that rearranges the items into four clusters in $O(n)$ time using only $O(1)$ extra space.

G	H	R	R	G	G	R	G	H	H	R	S	R	R	H	G	S	H	G	G
Harry	Ann	Bob	Tina	Chad	Bill	Lisa	Ekta	Bart	Jim	John	Jeff	Liz	Mary	Dawn	Nick	Kim	Fox	Dana	Mel

$$\Downarrow$$

G	G	G	G	G	G	G	H	H	H	H	H	R	R	R	R	R	R	S	S
Harry	Ekta	Bill	Chad	Nick	Mel	Dana	Fox	Ann	Jim	Dawn	Bart	Lisa	Tina	John	Bob	Liz	Mary	Kim	Jeff

6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

   Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, . . . , 52 of clubs. (They're big cards.) Penn shuffles the deck until each each of the 52! possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the three of clubs.

   (a) On average, how many cards does Penn give Teller?

   (b) On average, what is the smallest-numbered card that Penn gives Teller?

   *(c) On average, what is the largest-numbered card that Penn gives Teller?

   [Hint: Solve for an n-card deck and then set $n = 52$.] In each case, give *exact* answers and prove that they are correct. If you have to appeal to "intuition" or "common sense", your answers are probably wrong!

[1] Since you've read the Homework Instructions, you know what the phrase 'describe an algorithm' means. Right?

Practice Problems

The remaining problems are entirely for your benefit; similar questions will appear in every homework. Don't turn in solutions—we'll just throw them out—but feel free to ask us about practice questions during office hours and review sessions. Think of them as potential exam questions (hint, hint). We'll post solutions to *some* of the practice problems after the homeworks are due.

1. Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m.

   (a) F_n is even if and only if n is divisible by 3.

   (b) $\sum_{i=0}^{n} F_i = F_{n+2} - 1$

   (c) $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$

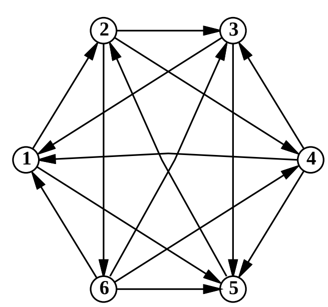
   ★(d) If n is an integer multiple of m, then F_n is an integer multiple of F_m.

2. (a) Prove the following identity by induction:

$$\binom{2n}{n} = \sum_{k=0}^{n} \binom{n}{k}\binom{n}{n-k}.$$

   (b) Give a non-inductive combinatorial proof of the same identity, by showing that the two sides of the equation count exactly the same thing in two different ways. There is a correct one-sentence proof.

3. A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once. Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \to 4 \to 5 \to 2 \to 3 \to 1$.

4. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway just for practice. Assume reasonable but nontrivial base cases if none are supplied. Extra credit will be given for more exact solutions.

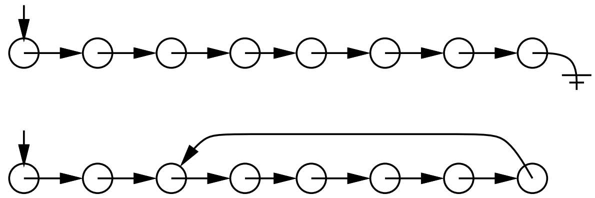
   (a) $A(n) = A(n/2) + n$
   (b) $B(n) = 2B(n/2) + n$
   ★(c) $C(n) = n + \frac{1}{2}\big(C(n-1) + C(3n/4)\big)$
   (d) $D(n) = \max\limits_{n/3<k<2n/3} \big(D(k) + D(n-k) + n\big)$
   ⋆(e) $E(n) = 2E(n/2) + n/\lg n$
   ⋆(f) $F(n) = \dfrac{F(n-1)}{F(n-2)}$, where $F(1) = 1$ and $F(2) = 2$.
   ⋆(g) $G(n) = G(n/2) + G(n/4) + G(n/6) + G(n/12) + n$ [Hint: $\frac{1}{2} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} = 1$.]
   ⋆(h) $H(n) = n + \sqrt{n} \cdot H(\sqrt{n})$
   ⋆(i) $I(n) = (n-1)(I(n-1) + I(n-2))$, where $F(0) = F(1) = 1$
   ⋆(j) $J(n) = 8J(n-1) - 15J(n-2) + 1$

5. (a) Prove that $2^{\lceil \lg n \rceil + \lfloor \lg n \rfloor} = \Theta(n^2)$.
   (b) Prove or disprove: $2^{\lfloor \lg n \rfloor} = \Theta\big(2^{\lceil \lg n \rceil}\big)$.
   (c) Prove or disprove: $2^{2^{\lfloor \lg \lg n \rfloor}} = \Theta\big(2^{2^{\lceil \lg \lg n \rceil}}\big)$.
   (d) Prove or disprove: If $f(n) = O(g(n))$, then $\log(f(n)) = O(\log(g(n)))$.
   (e) Prove or disprove: If $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$.
   ⋆(f) Prove that $\log^k n = o(n^{1/k})$ for any positive integer k.

6. Evaluate the following summations; simplify your answers as much as possible. Significant partial credit will be given for answers in the form $\Theta(f(n))$ for some recognizable function $f(n)$.

   (a) $\displaystyle\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=j}^{i} \frac{1}{i}$
   ⋆(b) $\displaystyle\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=j}^{i} \frac{1}{j}$
   (c) $\displaystyle\sum_{i=1}^{n} \sum_{j=1}^{i} \sum_{k=j}^{i} \frac{1}{k}$

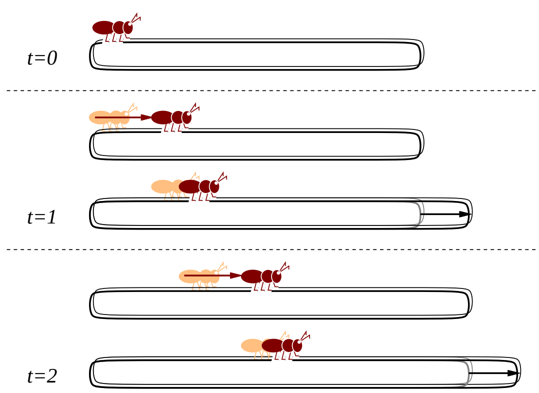
7. Suppose you have a pointer to the head of singly linked list. Normally, each node in the list only has a pointer to the next element, and the last node's pointer is NULL. Unfortunately, your list might have been corrupted by a bug in somebody else's code[2], so that the last node has a pointer back to some other node in the list instead.



Top: A standard linked list. Bottom: A corrupted linked list.

Describe an algorithm that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the list, and use $O(1)$ extra space (not counting the list itself).

*8. An ant is walking along a rubber band, starting at the left end. Once every second, the ant walks one inch to the right, and then you make the rubber band one inch longer by pulling on the right end. The rubber band stretches uniformly, so stretching the rubber band also pulls the ant to the right. The initial length of the rubber band is n inches, so after t seconds, the rubber band is $n + t$ inches long.



Every second, the ant walks an inch, and then the rubber band is stretched an inch longer.

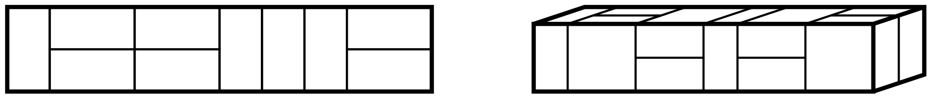
(a) How far has the ant moved after t seconds, as a function of n and t? Set up a recurrence and (for full credit) give an *exact* closed-form solution. [Hint: What *fraction* of the rubber band's length has the ant walked?]

(b) How long does it take the ant to get to the right end of the rubber band? For full credit, give an answer of the form $f(n) + \Theta(1)$ for some explicit function $f(n)$.

9. (a) A *domino* is a 2×1 or 1×2 rectangle. How many different ways are there to completely fill a $2 \times n$ rectangle with n dominos? Set up a recurrence relation and give an *exact* closed-form solution.

[2]After all, *your* code is always completely 100% bug-free. Isn't that right, Mr. Gates?

(b) A *slab* is a three-dimensional box with dimensions $1 \times 2 \times 2$, $2 \times 1 \times 2$, or $2 \times 2 \times 1$. How many different ways are there to fill a $2 \times 2 \times n$ box with n slabs? Set up a recurrence relation and give an *exact* closed-form solution.



A 2×10 rectangle filled with ten dominos, and a $2 \times 2 \times 10$ box filled with ten slabs.

10. Professor George O'Jungle has a favorite 26-node binary tree, whose nodes are labeled by letters of the alphabet. The preorder and postorder sequences of nodes are as follows:

    preorder: `M N H C R S K W T G D X I Y A J P O E Z V B U L Q F`
    postorder: `C W T K S G R H D N A O E P J Y Z I B Q L F U V X M`

    Draw Professor O'Jungle's binary tree, and give the inorder sequence of nodes.

11. Alice and Bob each have a fair n-sided die. Alice rolls her die once. Bob then repeatedly throws his die until he rolls a number at least as big as the number Alice rolled. Each time Bob rolls, he pays Alice \$1. (For example, if Alice rolls a 5, and Bob rolls a 4, then a 3, then a 1, then a 5, the game ends and Alice gets \$4. If Alice rolls a 1, then no matter what Bob rolls, the game will end immediately, and Alice will get \$1.)

    Exactly how much money does Alice expect to win at this game? Prove that your answer is correct. If you have to appeal to 'intuition' or 'common sense', your answer is probably wrong!

12. Prove that for any nonnegative parameters a and b, the following algorithms terminate and produce identical output.

```
SlowEuclid(a, b):
  if b > a
      return SlowEuclid(b, a)
  else if b = 0
      return a
  else
      return SlowEuclid(b, a − b)
```

```
FastEuclid(a, b):
  if b = 0
      return a
  else
      return FastEuclid(b, a mod b)
```

CS 373: Combinatorial Algorithms, Spring 2001

Homework 1 (due Thursday, February 1, 2001 at 11:59:59 p.m.)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \cdots < c[n]$. (In the U.S., for example, $n = 6$ and the values are $1, 5, 10, 25, 50$ and 100 cents.) Your beloved and belevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.

   (a) In the United States, there is a simple greedy algorithm that always results in the smallest number of coins: subtract the largest coin and recursively give change for the remainder. El Generalissimo does not approve of American capitalist greed. Show that there is a set of coin values for which the greedy algorithm does *not* always give the smallest possible of coins.

   (b) Describe and analyze a dynamic programming algorithm to determine, given a target amount A and a sorted array $c[1\,..\,n]$ of coin values, the smallest number of coins needed to make A cents in change. You can assume that $c[1] = 1$, so that it is possible to make change for any amount A.

2. Consider the following sorting algorithm:

```
StupidSort(A[0 .. n−1]):
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m ← ⌈2n/3⌉
      StupidSort(A[0 .. m−1])
      StupidSort(A[n−m .. n−1])
      StupidSort(A[0 .. m−1])
```

   (a) Prove that StupidSort actually sorts its input.

   (b) Would the algorithm still sort correctly if we replaced the line $m \leftarrow \lceil 2n/3 \rceil$ with $m \leftarrow \lfloor 2n/3 \rfloor$? Justify your answer.

   (c) State a recurrence (including the base case(s)) for the number of comparisons executed by StupidSort.

   (d) Solve the recurrence, and prove that your solution is correct. [Hint: Ignore the ceiling.] Does the algorithm deserve its name?

   *(e) Show that the number of *swaps* executed by StupidSort is at most $\binom{n}{2}$.

3. The following randomized algorithm selects the rth smallest element in an unsorted array $A[1 .. n]$. For example, to find the smallest element, you would call RandomSelect$(A, 1)$; to find the median element, you would call RandomSelect$(A, \lfloor n/2 \rfloor)$. Recall from lecture that Partition splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n-1$ comparisons altogether, and returns the new index of the pivot element.

```
RandomSelect(A[1 .. n], r):
  p ← Random(1, n)
  k ← Partition(A[1 .. n], p)

  if r < k
      return RandomSelect(A[1 .. k−1], r)
  else if r > k
      return RandomSelect(A[k+1 .. n], r−k)
  else
      return A[k]
```

   (a) State a recurrence for the expected running time of RandomSelect, as a function of n and r.

   (b) What is the *exact* probability that RandomSelect compares the ith smallest and jth smallest elements in the input array? The correct answer is a simple function of i, j, and r. [Hint: Check your answer by trying a few small examples.]

   (c) Show that for any n and r, the expected running time of RandomSelect is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b). For extra credit, find the exact expected number of comparisons, as a function of n and r.

   (d) What is the expected number of times that RandomSelect calls itself recursively?

4. What excitement! The Champaign Spinners and the Urbana Dreamweavers have advanced to meet each other in the World Series of Basketweaving! The World Champions will be decided by a best-of- $2n-1$ series of head-to-head weaving matches, and the first to win n matches will take home the coveted Golden Basket (for example, a best-of-7 series requiring four match wins, but we will keep the generalized case). We know that for any given match there is a constant probability p that Champaign will win, and a subsequent probability $q = 1 - p$ that Urbana will win.

   Let $P(i, j)$ be the probability that Champaign will win the series given that they still need i more victories, whereas Urbana needs j more victories for the championship. $P(0, j) = 1$, $1 \leq j \leq n$, because Champaign needs no more victories to win. $P(i, 0) = 0$, $1 \leq i \leq n$, as Champaign cannot possibly win if Urbana already has. $P(0, 0)$ is meaningless. Champaign wins any particular match with probability p and loses with probability q, so

$$P(i, j) = p \cdot P(i-1, j) + q \cdot P(i, j-1)$$

   for any $i \geq 1$ and $j \geq 1$.

   Create and analyze an $O(n^2)$-time dynamic programming algorithm that takes the parameters n, p and q and returns the probability that Champaign will win the series (that is, calculate $P(n, n)$).

5. The traditional Devonian/Cornish drinking song "The Barley Mow" has the following pseudolyrics[1], where *container*$[i]$ is the name of a container that holds 2^i ounces of beer.[2]

```
BarleyMow(n):
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"

    "We'll drink it out of the jolly brown bowl,"
    "Here's a health to the barley-mow!"
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"

    for i ← 1 to n
        "We'll drink it out of the container[i], boys,"
        "Here's a health to the barley-mow!"
        for j ← i downto 1
            "The container[j],"
        "And the jolly brown bowl!"
        "Here's a health to the barley-mow!"
        "Here's a health to the barley-mow, my brave boys,"
        "Here's a health to the barley-mow!"
```

   (a) Suppose each container name *container*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

   (b) If you want to sing this song for $n > 20$, you'll have to make up your own container names, and to avoid repetition, these names will get progressively longer as n increases[3]. Suppose *container*$[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

   (c) Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *container*$[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BarleyMow(n)? (Give an *exact* answer, not just an asymptotic bound.)

[1]Pseudolyrics are to lyrics as pseudocode is to code.

[2]One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.

[3]"We'll drink it out of the hemisemidemiyottapint, boys!"

6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

   Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the ith word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of whitespace between any two words on the same line.

   Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j, then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^{j} p_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

Practice Problems

1. Give an $O(n^2)$ algorithm to find the longest increasing subsequence of a sequence of numbers. The elements of the subsequence need not be adjacent in the sequence. For example, the sequence $\langle 1, 5, 3, 2, 4 \rangle$ has longest increasing subsequence $\langle 1, 3, 4 \rangle$.

2. You are at a political convention with n delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party a delegate belongs to. However, you can check whether any two delegates are in the *same* party or not by introducing them to each other. (Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.)

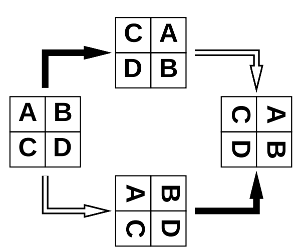
   (a) Suppose a majority (more than half) of the delegates are from the same political party. Give an efficient algorithm that identifies a member of the majority party.

   (b) Suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick a person from the plurality party as parsimoniously as possible. (Please.)

3. Give an algorithm that finds the *second* smallest of n elements in at most $n + \lceil \lg n \rceil - 2$ comparisons. [Hint: divide and conquer to find the smallest; where is the second smallest?]

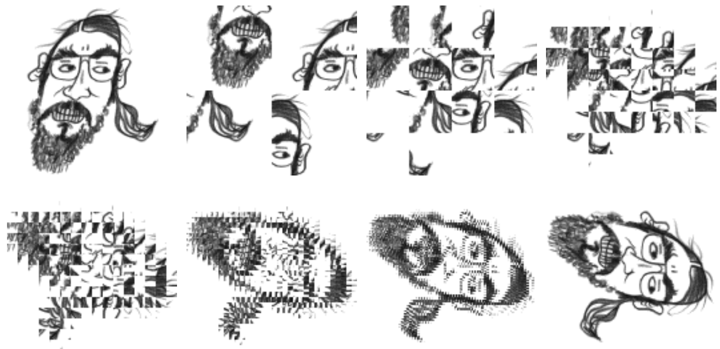
4. Some graphics hardware includes support for an operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixelmap (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

   Suppose we want to rotate an $n \times n$ pixelmap 90° clockwise. One way to do this is to split the pixelmap into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we can first recursively rotate the blocks and blit them into place afterwards.



Two algorithms for rotating a pixelmap.
Black arrows indicate blitting the blocks into place.
White arrows indicate recursively rotating the blocks.

The following sequence of pictures shows the first algorithm (blit then recurse) in action.



In the following questions, assume n is a power of two.

(a) Prove that both versions of the algorithm are correct. [Hint: If you exploit all the available symmetries, your proof will only be a half of a page long.]

(b) *Exactly* how many blits does the algorithm perform?

(c) What is the algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

(d) What if a $k \times k$ blit takes only $O(k)$ time?

5. A company is planning a party for its employees. The employees in the company are organized into a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party have assigned a real number to each employee measuring how 'fun' the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the 'fun' ratings of the guests.

6. Suppose you have a subroutine that can find the median of a set of n items (*i.e.*, the $\lfloor n/2 \rfloor$ smallest) in $O(n)$ time. Give an algorithm to find the kth biggest element (for arbitrary k) in $O(n)$ time.

7. You're walking along the beach and you stub your toe on something in the sand. You dig around it and find that it is a treasure chest full of gold bricks of different (integral) weight. Your knapsack can only carry up to weight n before it breaks apart. You want to put as much in it as possible without going over, but you *cannot* break the gold bricks up.

   (a) Suppose that the gold bricks have the weights $1, 2, 4, 8, \ldots, 2^k$, $k \geq 1$. Describe and prove correct a greedy algorithm that fills the knapsack as much as possible without going over.

   (b) Give a set of 3 weight values for which the greedy algorithm does *not* yield an optimal solution and show why.

   (c) Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of gold brick values.

CS 373: Combinatorial Algorithms, Spring 2001

Homework 2 (due Thu. Feb. 15, 2001 at 11:59 PM)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Suppose we are given two sorted arrays $A[1\,..\,n]$ and $B[1\,..\,n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B. (For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, our algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicates. For full credit, your algorithm should run in $\Theta(\log n)$ time. *[Hint: First try to solve the special case $k = n$.]*

2. Say that a binary search tree is *augmented* if every node v also stores $|v|$, the size of its subtree.

   (a) Show that a rotation in an augmented binary tree can be performed in constant time.

   (b) Describe an algorithm ScapegoatSelect(k) that selects the kth smallest item in an augmented scapegoat tree in $O(\log n)$ *worst-case* time.

   (c) Describe an algorithm SplaySelect(k) that selects the kth smallest item in an augmented splay tree in $O(\log n)$ *amortized* time.

(d) Describe an algorithm TREAPSELECT(k) that selects the kth smallest item in an augmented treap in $O(\log n)$ *expected* time.

3. (a) Prove that only one subtree gets rebalanced in a scapegoat tree insertion.

   (b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree. (Recall that $I(v) = \max\{0, |T| - |s| - 1\}$, where T is the child of greater height and s the child of lesser height, and $|v|$ is the number of nodes in subtree v. A perfectly balanced tree has two perfectly balanced subtrees, each with as close to half the nodes as possible.)

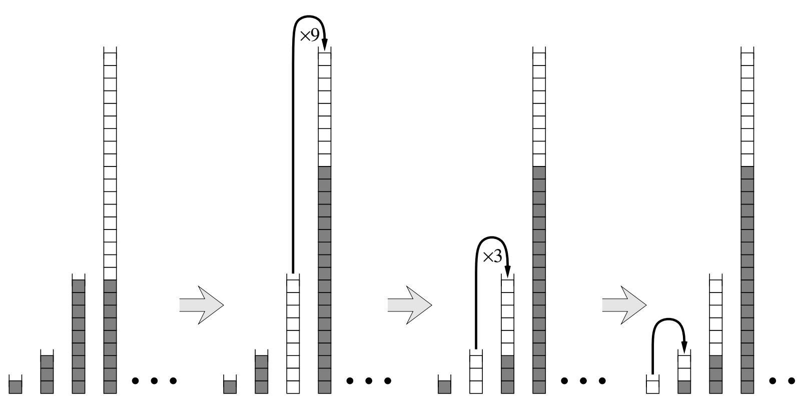
   *(c) Show that you can rebuild a fully balanced binary tree from an unbalanced tree in $O(n)$ time using only $O(\log n)$ additional memory.

4. Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than 1/4 full, we we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method (it makes it much more difficult).

5. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i, we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2}, and so on. Moving a single element from one stack to the next takes $O(1)$ time.



Making room for one new element in a multistack.

   (a) *[1 point]* In the worst case, how long does it take to push one more element onto a multistack containing n elements?

   (b) *[9 points]* Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack. You can use any method you like.

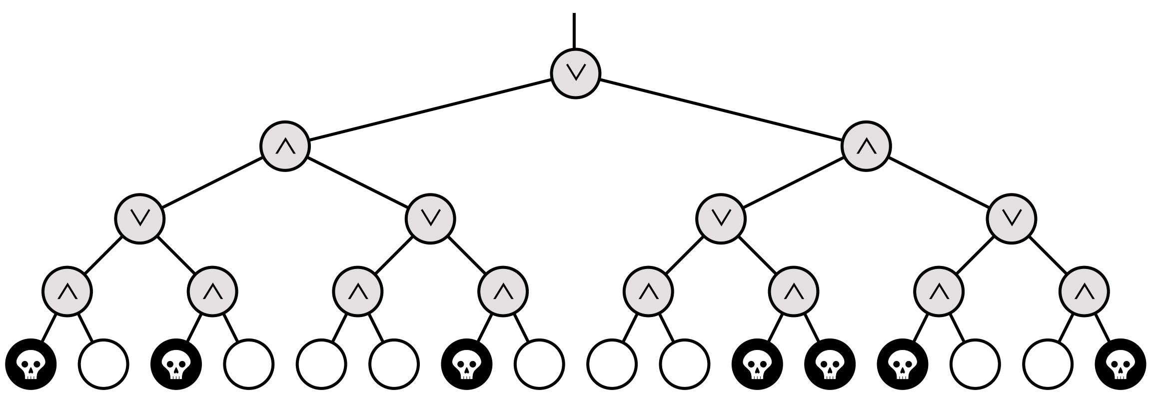
6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

   Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.

   You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are *and* gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

   (a) (2 pts) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

   (b) (8 pts) Unfortunately, Death won't let you even look at every node in the tree. Describe a *randomized* algorithm that determines whether you can win in $\Theta(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*



Practice Problems

1. (a) Show that it is possible to transform any n-node binary search tree into any other n-node binary search tree using at most $2n-2$ rotations.

   *(b) Use fewer than $2n-2$ rotations. Nobody knows how few rotations are required in the worst case. There is an algorithm that can transform any tree to any other in at most $2n-6$ rotations, and there are pairs of trees that are $2n-10$ rotations apart. These are the best bounds known.

2. Faster Longest Increasing Subsequence(LIS)
   Give an $O(n \log n)$ algorithm to find the longest increasing subsequence of a sequence of numbers. [Hint: In the dynamic programming solution, you don't really have to look back at all previous items. There was a practice problem on HW 1 that asked for an $O(n^2)$ algorithm for this. If you are having difficulty, look at the solution provided in the HW 1 solutions.]

3. Amortization

   (a) Modify the binary double-counter (see class notes Sept 12) to support a new operation Sign, which determines whether the number being stored is positive, negative, or zero, in constant time. The amortized time to increment or decrement the counter should still be a constant.

   [Hint: Suppose p is the number of significant bits in P, and n is the number of significant bits in N. For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Assume you can update p and n in $O(1)$ time.]

   *(b) Do the same but now you can't assume that p and n can be updated in $O(1)$ time.

*4. Amortization
   Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of 'fits', where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string. This is *not* the same representation as on Homework 0.]

5. Detecting overlap

   (a) You are given a list of ranges represented by min and max (*e.g.*, [1,3], [4,5], [4,9], [6,8], [7,10]). Give an $O(n \log n)$-time algorithm that decides whether or not a set of ranges contains a pair that overlaps. You need not report all intersections. If a range completely covers another, they are overlapping, even if the boundaries do not intersect.

   (b) You are given a list of rectangles represented by min and max x- and y-coordinates. Give an $O(n \log n)$-time algorithm that decides whet her or not a set of rectangles contains a pair that overlaps (with the same qualifications as above). [Hint: sweep a vertical line from left to right, performing some processing whenever an end-point is encountered. Use a balanced search tree to maintain any extra info you might need.]

6. Comparison of Amortized Analysis Methods
   A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

   Determine the amortized cost per operation using the following methods of analysis:

   (a) Aggregate method

   (b) Accounting method

   *(c) Potential method

CS 373: Combinatorial Algorithms, Spring 2001
Homework 3 (due Thursday, March 8, 2001 at 11:59.99 p.m.)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Hashing:
   A hash table of size m is used to store n items with $n \leq m/2$. Open addressing is used for collision resolution.

   (a) Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability that the i^{th} insertion requires strictly more than k probes is at most 2^{-k}.

   (b) Show that for $i = 1, 2, \ldots, n$, the probability that the i^{th} insertion requires more than $2 \lg n$ probes is at most $1/n^2$.

   Let the random variable X_i denote the number of probes required by the i^{th} insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

   (c) Show that $\Pr\{X > 2 \lg n\} \leq 1/n$.

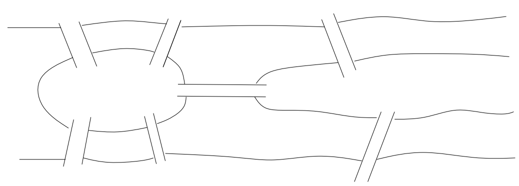
   (d) Show that the expected length of the longest probe sequence is $E[X] = O(\lg n)$.

2. Reliable Network:
Suppose you are given a graph of a computer network $G = (V, E)$ and a function $r(u, v)$ that gives a reliability value for every edge $(u, v) \in E$ such that $0 \leq r(u, v) \leq 1$. The reliability value gives the probability that the network connection corresponding to that edge will *not* fail. Describe and analyze an algorithm to find the most reliable path from a given source vertex s to a given target vertex t.

3. Aerophobia:
After graduating you find a job with Aerophobes-R′-Us, the leading traveling agency for aerophobic people. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying so the trip should be as short as possible.

   In other words, a person wants to fly from city A to city B in the shortest possible time. S/he turns to the traveling agent who knows all the departure and arrival times of all the flights on the planet. Give an algorithm that will allow the agent to choose an optimal route to minimize the total time in transit. Hint: rather than modify Dijkstra's algorithm, modify the data. The total transit time is from departure to arrival at the destination, so it will include layover time (time waiting for a connecting flight).

4. The Seven Bridges of Königsberg:
During the eighteenth century the city of Königsberg in East Prussia was divided into four sections by the Pregel river. Seven bridges connected these regions, as shown below. It was said that residents spent their Sunday walks trying to find a way to walk about the city so as to cross each bridge exactly once and then return to their starting point.



   (a) Show how the residents of the city could accomplish such a walk or prove no such walk exists.

   (b) Given any undirected graph $G = (V, E)$, give an algorithm that finds a cycle in the graph that visits every edge exactly once, or says that it can't be done.

5. Minimum Spanning Tree changes:
Suppose you have a graph G and an MST of that graph (i.e. the MST has already been constructed).

   (a) Give an algorithm to update the MST when an edge is added to G.

   (b) Give an algorithm to update the MST when an edge is deleted from G.

   (c) Give an algorithm to update the MST when a vertex (and possibly edges to it) is added to G.

6. Nesting Envelopes
[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.] You are given an unlimited number of each of n different types of envelopes. The dimensions of envelope type i are $x_i \times y_i$. In nesting envelopes inside one another, you can place envelope A inside envelope B if and only if the dimensions A are *strictly smaller* than the dimensions of B. Design and analyze an algorithm to determine the largest number of envelopes that can be nested inside one another.

Practice Problems

1. Makefiles:
In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called 'make' that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of the other files which are listed. Thus a source file must be recompiled if a file on which it depends is changed.

   Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design an algorithm to recompile only those necessary. DO NOT worry about the details of parsing a Makefile.

*2. Let the hash function for a table of size m be

$$h(x) = \lfloor Amx \rfloor \bmod m$$

where $A = \hat{\phi} = \frac{\sqrt{5}-1}{2}$. Show that this gives the best possible spread, i.e. if the x are hashed in order, $x + 1$ will be hashed in the largest remaining contiguous interval.

3. The incidence matrix of an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} 1 & (i, j) \in E, \\ 0 & (i, j) \notin E. \end{cases}$$

   (a) Describe what all the entries of the matrix product BB^T represent (B^T is the matrix transpose). Justify.

   (b) Describe what all the entries of the matrix product $B^T B$ represent. Justify.

   *(c) Let $C = BB^T - 2A$. Let C' be C with the first row and column removed. Show that $\det C'$ is the number of spanning trees. (A is the adjacency matrix of G, with zeroes on the diagonal).

4. $o(V^2)$ Adjacency Matrix Algorithms

   (a) Give an $O(V)$ algorithm to decide whether a directed graph contains a *sink* in an adjacency matrix representation. A sink is a vertex with in-degree $V - 1$.

(b) An undirected graph is a scorpion if it has a vertex of degree 1 (the sting) connected to a vertex of degree two (the tail) connected to a vertex of degree $V-2$ (the body) connected to the other $V-3$ vertices (the feet). Some of the feet may be connected to other feet.

Design an algorithm that decides whether a given adjacency matrix represents a scorpion by examining only $O(V)$ of the entries.

(c) Show that it is impossible to decide whether G has at least one edge in $O(V)$ time.

5. Shortest Cycle:
   Given an **undirected** graph $G = (V, E)$, and a weight function $f : E \to \mathbf{R}$ on the **edges**, give an algorithm that finds (in time polynomial in V and E) a cycle of smallest weight in G.

6. Longest Simple Path:
   Let graph $G = (V, E)$, $|V| = n$. A *simple path* of G, is a path that does not contain the same vertex twice. Use dynamic programming to design an algorithm (not polynomial time) to find a simple path of maximum length in G. Hint: It can be done in $O(n^c 2^n)$ time, for some constant c.

7. Minimum Spanning Tree:
   Suppose all edge weights in a graph G are equal. Give an algorithm to compute an MST.

8. Transitive reduction:
   Give an algorithm to construct a *transitive reduction* of a directed graph G, i.e. a graph G^{TR} with the fewest edges (but with the same vertices) such that there is a path from a to b in G iff there is also such a path in G^{TR}.

9. (a) What is $5^{2^{29^{5^0}+23^{4^1}+17^{3^2}+11^{2^3}+5^{1^4}}} \bmod 6$?

   (b) What is the capital of Nebraska? Hint: It is not Omaha. It is named after a famous president of the United States that was not George Washington. The distance from the Earth to the Moon averages roughly 384,000 km.

CS 373: Combinatorial Algorithms, Spring 2001

`http://www-courses.cs.uiuc.edu/~cs373`

Homework 4 (due Thu. March 29, 2001 at 11:59:59 pm)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

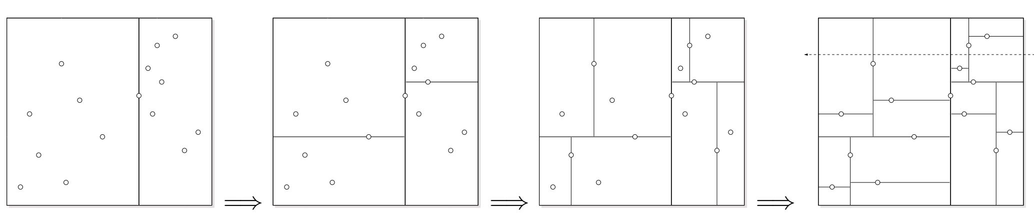
Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

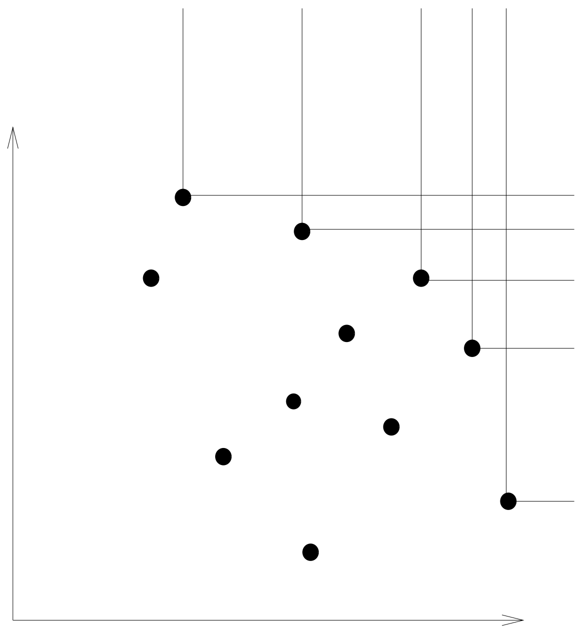
Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the rectangle as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point inside the box (*not* on the boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



Successive divisions of a kd-tree for 15 points. The dashed line crosses four cells.



An example staircase as in problem 3.

(a) How many cells are there, as a function of n? Prove your answer is correct.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k.

(c) Suppose we have n points stored in a kd-tree. Describe an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) in $O(\sqrt{n})$ time.

*(d) *[Optional: 5 pts extra credit]* Find an algorithm that counts the number of points that lie inside a rectangle R and show that it takes $O(\sqrt{n})$ time. You may assume that the sides of the rectangle are parallel to the sides of the box.

2. Circle Intersection *[This problem is worth 20 points]*
Describe an algorithm to decide, given n circles in the plane, whether any two of them intersect, in $O(n \log n)$ time. Each circle is specified by three numbers: its radius and the x- and y-coordinates of its center.

We only care about intersections between circle boundaries; concentric circles do not intersect. What general position assumptions does your algorithm require? *[Hint: Modify an algorithm for detecting line segment intersections, but describe your modifications very carefully! There are at least two very different solutions.]*

3. Staircases
You are given a set of points in the first quadrant. A *left-up* point of this set is defined to be a point that has no points both greater than it in both coordinates. The left-up subset of a set of points then forms a *staircase* (see figure).

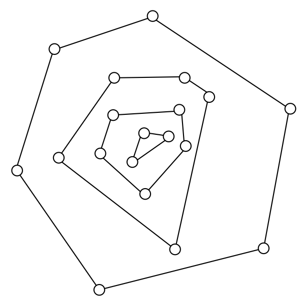
(a) Prove that left-up points do not necessarily lie on the convex hull.

(b) Give an $O(n \log n)$ algorithm to find the staircase of a set of points.

(c) Assume that points are chosen uniformly at random within a rectangle. What is the average number of points in a staircase? Justify. Hint: you will be able to give an exact answer rather than just asymptotics. You have seen the same analysis before.

4. Convex Layers
Given a set Q of points in the plane, define the *convex layers* of Q inductively as follows: The first convex layer of Q is just the convex hull of Q. For all $i > 1$, the ith convex layer is the convex hull of Q after the vertices of the first $i-1$ layers have been removed.

Give an $O(n^2)$-time algorithm to find all convex layers of a given set of n points.



A set of points with four convex layers.

5. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]* Solve the travelling salesman problem for points in convex position (ie, the vertices of a convex polygon). Finding the shortest cycle that visits every point is easy – it's just the convex hull. Finding the shortest path that visits evey point is a little harder, because the path can cross through the interior.

   (a) Show that the optimal path cannot be one that crosses itself.
   (b) Describe an $O(n^2)$ time dynamic programming algorithm to solve the problem.

Practice Problems

1. Basic Computation (assume two dimensions and *exact* arithmetic)

   (a) Intersection: Extend the basic algorithm to determine if two line segments intersect by taking care of *all* degenerate cases.

   (b) Simplicity: Give an $O(n \log n)$ algorithm to determine whether an n-vertex polygon is simple.

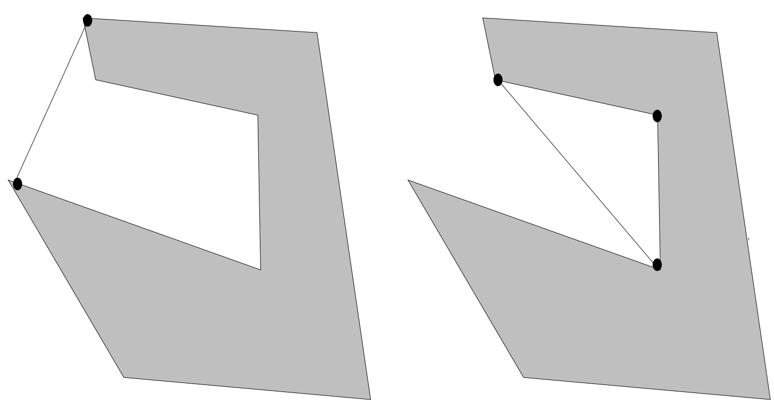
   (c) Area: Give an algorithm to compute the area of a simple n-polygon (not necessarily convex) in $O(n)$ time.

   (d) Inside: Give an algorithm to determine whether a point is within a simple n-polygon (not necessarily convex) in $O(n)$ time.

2. External Diagonals and Mouths

   (a) A pair of polygon vertices defines an *external diagonal* if the line segment between them is completely outside the polygon. Show that every nonconvex polygon has at least one external diagonal.

   (b) Three consective polygon vertices p, q, r form a *mouth* if p and r define an external diagonal. Show that every nonconvex polygon has at least one mouth.



An external diagonal A mouth

3. On-Line Convex Hull
   We are given the set of points one point at a time. After receiving each point, we must compute the convex hull of all those points so far. Give an algorithm to solve this problem in $O(n^2)$ (We could obviously use Graham's scan n times for an $O(n^2 \log n)$ algorithm). Hint: How do you maintain the convex hull?

4. Another On-Line Convex Hull Algorithm

   (a) Given an n-polygon and a point outside the polygon, give an algorithm to find a tangent.

   *(b) Suppose you have found both tangents. Give an algorithm to remove the points from the polygon that are within the angle formed by the tangents (as segments!) and the opposite side of the polygon.

(c) Use the above to give an algorithm to compute the convex hull on-line in $O(n \log n)$

5. Order of the size of the convex hull
   The convex hull on $n \geq 3$ points can have anywhere from 3 to n points. The average case depends on the distribution.

   (a) Prove that if a set of points is chosen randomly within a given rectangle then the average size of the convex hull is $O(\log n)$.

   *(b) Prove that if a set of points is chosen randomly within a given circle then the average size of the convex hull is $O(n^{1/3})$.

6. Ghostbusters and Ghosts
   A group of n ghostbusters is battling n ghosts. Each ghostbuster can shoot a single energy beam at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits a ghost. The ghostbusters must all fire at the same time and no two energy beams may cross (it would be bad). The positions of the ghosts and ghostbusters is fixed in the plane (assume that no three points are collinear).

   (a) Prove that for any configuration of ghosts and ghostbusters there exists such a non-crossing matching.

   (b) Show that there exists a line passing through one ghostbuster and one ghost such that the number of ghostbusters on one side of the line equals the number of ghosts on the same side. Give an efficient algorithm to find such a line.

   (c) Give an efficient divide and conquer algorithm to pair ghostbusters and ghosts so that no two streams cross.

CS 373: Combinatorial Algorithms, Spring 2001

http://www-courses.cs.uiuc.edu/~cs373

Homework 5 (due Tue. Apr. 17, 2001 at 11:59 pm)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

RequiredProblems

1. Prove that finding the second smallest of n elements takes EXACTLY $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. Prove for both upper and lower bounds. Hint: find the (first) smallest using an elimination tournament.

2. *Fibonacci strings* are defined as follows:

$$F_1 = \text{“b”}, \qquad F_2 = \text{“a”}, \qquad \text{and } F_n = F_{n-1}F_{n-2}, (n > 2)$$

where the recursive rule uses concatenation of strings, so F_3 is “ab”, F_4 is “aba”. Note that the length of F_n is the nth Fibonacci number.

   (a) Prove that in any Fibonacci string there are no two b's adjacent and no three a's.

   (b) Give the unoptimized and optimized ‘prefix’ (fail) function for F_7.

   (c) Prove that, in searching for the Fibonacci string F_k, the unoptimized KMP algorithm can shift $\lceil k/2 \rceil$ times in a row trying to match the last character of the pattern. In other words, prove that there is a chain of failure links $m \to fail[m] \to fail[fail[m]] \to \ldots$ of length $\lceil k/2 \rceil$, and find an example text T that would cause KMP to traverse this entire chain at a single text position.

(d) Prove that the unoptimized KMP algorithm can shift $k - 2$ times in a row at the same text position when searching for F_k. Again, you need to find an example text T that would cause KMP to traverse this entire chain on the same text character.

(e) How do the failure chains in parts (c) and (d) change if we use the optimized failure function instead?

3. Two-stage sorting

(a) Suppose we are given an array $A[1..n]$ of distinct integers. Describe an algorithm that splits A into n/k subarrays, each with k elements, such that the elements of each subarray $A[(i-1)k+1..ik]$ are sorted. Your algorithm should run in $O(n \log k)$ time.

(b) Given an array $A[1..n]$ that is already split into n/k sorted subarrays as in part (a), describe an algorithm that sorts the entire array in $O(n \log(n/k))$ time.

(c) Prove that your algorithm from part (a) is optimal.

(d) Prove that your alogrithm from part (b) is optimal.

4	14	7	3	1	20	11	9	5	13	12	19	10	16	17	2	8	6	18	15

$\Downarrow (a)$

1	3	4	7	14	5	9	11	13	20	10	12	16	17	19	2	6	8	15	18

$\Downarrow (b)$

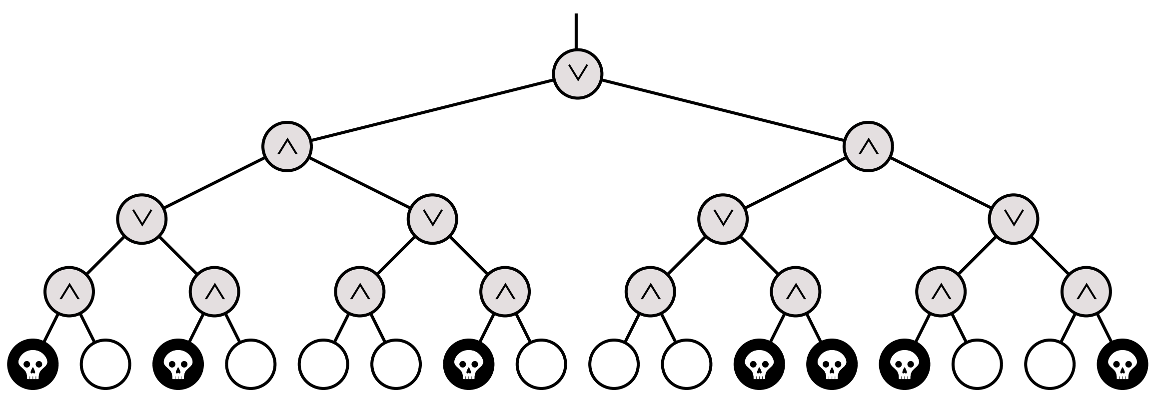
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

4. Show how to extend the Rabin-Karp fingerprinting method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. (The pattern may be shifted horizontally and vertically, but it may not be rotated.)

5. Death knocks on your door once more on a warm spring day. He remembers that you are an algorithms student and that you soundly defeated him last time and are now living out your immortality. Death is in a bit of a quandry. He has been losing a lot and doesn't know why. He wants you to prove a lower bound on your deterministic algorithm so that he can reap more souls. If you have forgotten, the game goes like this: It is a complete binary tree with 4^n leaves, each colored black or white. There is a toke at the root of the tree. To play the game, you and Death took turns movin the token from its current node to one of its children. The game ends after $2n$ moves, when the token lands on a leaf. If the final leaf is black, the player dies; if it's white, you will live forever. You move first, so Death gets the last turn.

   You decided whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are *and* gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should've challenge Death to a game of Twister instead.

   Prove that *any* deterministic algorithm must examine *every* leaf of the tree in the worst case. Since there are 4^n leaves, this implies that any deterministic algorithm must take $\Omega(4^n)$ time in the worst case. Use an adversary argument, or in other words, assume Death cheats.



6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

   Lower Bounds on Adjacency Matrix Representations of Graphs

   (a) Prove that the time to determine if an undirected graph has a cycle is $\Omega(V^2)$.

   (b) Prove that the time to determine if there is a path between two nodes in an undirected graph is $\Omega(V^2)$.

Practice Problems

1. String matching with wild-cards
   Suppose you have an alphabet for patterns that includes a 'gap' or wild-card character; any length string of any characters can match this additional character. For example if '$*$' is the wild-card, then the pattern 'foo*bar*nad' can be found in 'foofoowangbarnad'. Modify the computation of the prefix function to correctly match strings using KMP.

2. Prove that there is no comparison sort whose running time is linear for at least $1/2$ of the $n!$ inputs of length n. What about at least $1/n$? What about at least $1/2^n$?.

3. Prove that $2n - 1$ comparisons are necessary in the worst case to merge two sorted lists containing n elements each.

4. Find asymptotic upper and lower bounds to $\lg(n!)$ without Stirling's approximation (Hint: use integration).

5. Given a sequence of n elements of n/k blocks (k elements per block) all elements in a block are less than those to the right in sequence, show that you cannot have the whole sequence sorted in better than $\Omega(n \lg k)$. Note that the entire sequence would be sorted if each of the n/k blocks were individually sorted in place. Also note that combining the lower bounds for each block is not adequate (that only gives an upper bound).

6. Show how to find the occurrences of pattern P in text T by computing the prefix function of the string PT (the concatenation of P and T).

CS 373: Combinatorial Algorithms, Spring 2001

http://www-courses.cs.uiuc.edu/~cs373

Homework 6 (due Tue. May 1, 2001 at 11:59.99 p.m.)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since 1-unit graduate students are required to solve problems that are worth extra credit for other students, **1-unit grad students may not be on the same team as 3/4-unit grad students or undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Note: You will be held accountable for the appropriate responses for answers (e.g. give models, proofs, analyses, etc). For NP-complete problems you should prove everything rigorously, i.e. for showing that it is in NP, give a description of a certificate and a poly time algorithm to verify it, and for showing NP-hardness, you must show that your reduction is polytime (by similarly proving something about the algorithm that does the transformation) and proving both directions of the 'if and only if' (a solution of one is a solution of the other) of the many-one reduction.

Required Problems

1. Complexity

   (a) Prove that P $\subseteq$ co-NP.

   (b) Show that if NP $\neq$ co-NP, then *every* NP-complete problem is *not* a member of co-NP.

2. 2-CNF-SAT
   Prove that deciding satisfiability when all clauses have at most 2 literals is in P.

3. Graph Problems

(a) SUBGRAPH-ISOMORPHISM
Show that the problem of deciding whether one graph is a subgraph of another is NP-complete.

(b) LONGEST-PATH
Show that the problem of deciding whether an unweighted undirected graph has a path of length greater than k is NP-complete.

4. PARTITION, SUBSET-SUM
PARTITION is the problem of deciding, given a set of numbers, whether there exists a subset whose sum equals the sum of the complement, i.e. given $S = s_1, s_2 \ldots, s_n$, does there exist a subset S' such that $\sum_{s \in S'} s = \sum_{t \in S - S'} t$. SUBSET-SUM is the problem of deciding, given a set of numbers and a target sum, whether there exists a subset whose sum equals the target, i.e. given $S = s_1, s_2 \ldots, s_n$ and k, does there exist a subset S' such that $\sum_{s \in S'} s = k$. Give two reduction, one in both directions.

5. BIN-PACKING Consider the bin-packing problem: given a finite set U of n items and the positive integer size $s(u)$ of each item $u \in U$, can U be partitioned into k disjoint sets $U_1, \ldots, U_k$ such that the sum of the sizes of the items in each set does not exceed B? Show that the bin-packing problem is NP-Complete. [Hint: Use the result from the previous problem.]

6. 3SUM
[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]
Describe an algorithm that solves the following problem as quickly as possible: Given a set of n numbers, does it contain three elements whose sum is zero? For example, your algorithm should answer TRUE for the set $\{-5, -17, 7, -4, 3, -2, 4\}$, since $-5+7+(-2) = 0$, and FALSE for the set $\{-6, 7, -4, -13, -2, 5, 13\}$.

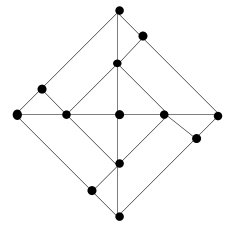


Figure 1. Gadget for PLANAR-3-COLOR.

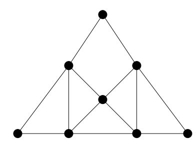


Figure 2. Gadget for DEGREE-4-PLANAR-3-COLOR.

Practice Problems

1. Consider finding the median of 5 numbers by using only comparisons. What is the exact worst case number of comparisons needed to find the median. Justify (exhibit a set that cannot be done in one less comparisons). Do the same for 6 numbers.

2. EXACT-COVER-BY-4-SETS
   The EXACT-COVER-BY-3-SETS problem is defines as the following: given a finite set X with $|X| = 3q$ and a collection C of 3-element subsets of X, does C contain an *exact cover* for X, that is, a subcollection $C' \subseteq C$ such that every element of X occurs in exactly one member of C'?

   Given that EXACT-COVER-BY-3-SETS is NP-complete, show that EXACT-COVER-BY-4-SETS is also NP-complete.

3. PLANAR-3-COLOR
   Using 3-COLOR, and the 'gadget' in figure 3, prove that the problem of deciding whether a planar graph can be 3-colored is NP-complete. Hint: show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.

4. DEGREE-4-PLANAR-3-COLOR
   Using the previous result, and the 'gadget' in figure 4, prove that the problem of deciding whether a planar graph with no vertex of degree greater than four can be 3-colored is NP-complete. Hint: show that you can replace any vertex with degree greater than 4 with a collection of gadgets connected in such a way that no degree is greater than four.

5. Poly time subroutines can lead to exponential algorithms
   Show that an algorithm that makes at most a constant number of calls to polynomial-time subroutines runs in polynomial time, but that a polynomial number of calls to polynomial-time subroutines may result in an exponential-time algorithm.

6. (a) Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian. Give a polynomial time algorithm algorithm for finding a **hamiltonian cycle** in an undirected bipartite graph or establishing that it does not exist.
   (b) Show that the **hamiltonian-path** problem can be solved in polynomial time on directed acyclic graphs by giving an efficient algorithm for the problem.
   (c) Explain why the results in previous questions do not contradict the facts that both HAM-CYCLE and HAM-PATH are NP-complete problems.

7. Consider the following pairs of problems:
   (a) MIN SPANNING TREE and MAX SPANNING TREE
   (b) SHORTEST PATH and LONGEST PATH
   (c) TRAVELING SALESMAN PROBLEM and VACATION TOUR PROBLEM (the longest tour is sought).
   (d) MIN CUT and MAX CUT (between s and t)
   (e) EDGE COVER and VERTEX COVER
   (f) TRANSITIVE REDUCTION and MIN EQUIVALENT DIGRAPH

   (all of these seem dual or opposites, except the last, which are just two versions of minimal representation of a graph).

   Which of these pairs are polytime equivalent and which are not? Why?

★8. GRAPH-ISOMORPHISM
   Consider the problem of deciding whether one graph is isomorphic to another.
   (a) Give a brute force algorithm to decide this.
   (b) Give a dynamic programming algorithm to decide this.
   (c) Give an efficient probabilistic algorithm to decide this.
   (d) Either prove that this problem is NP-complete, give a poly time algorithm for it, or prove that neither case occurs.

9. Prove that PRIMALITY (Given n, is n prime?) is in NP $\cap$ co-NP. Hint: co-NP is easy (what's a certificate for showing that a number is composite?). For NP, consider a certificate involving primitive roots and recursively their primitive roots. Show that knowing this tree of primitive roots can be checked to be correct and used to show that n is prime, and that this check takes poly time.

10. How much wood would a woodchuck chuck if a woodchuck could chuck wood?

Write your answers in the separate answer booklet.

1. **Multiple Choice:** Each question below has one of the following answers.

   (a) $\Theta(1)$ (b) $\Theta(\log n)$ (c) $\Theta(n)$ (d) $\Theta(n \log n)$ (e) $\Theta(n^2)$

   For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point, but each incorrect answer *costs* you $\frac{1}{2}$ point. You cannot score below zero.

   (a) What is $\sum_{i=1}^{n} H_i$?

   (b) What is $\sum_{i=1}^{\lg n} 2^i$?

   (c) How many digits do you need to write $n!$ in decimal?

   (d) What is the solution of the recurrence $T(n) = 16T(n/4) + n$?

   (e) What is the solution of the recurrence $T(n) = T(n-2) + \lg n$?

   (f) What is the solution of the recurrence $T(n) = 4T\left(\left\lceil \frac{n+51}{4} \right\rceil - \sqrt{n}\right) + 17n - 2^{8\log^*(n^2)} + 6$?

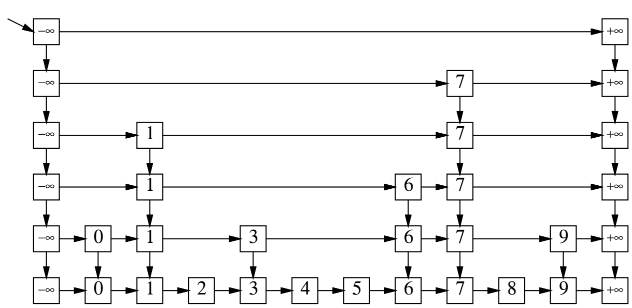
   (g) What is the worst-case running time of randomized quicksort?

   (h) The expected time for inserting one item into a treap is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty treap?

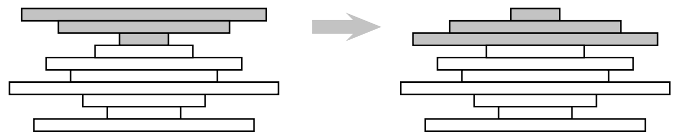
   (i) The amortized time for inserting one item into an n-node splay tree is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty splay tree?

   (j) In the worst case, how long does it take to solve the traveling salesman problem for $10,000,000,000,000,000$ cities?

2. What is the *exact* expected number of nodes in a skip list storing n keys, *not* counting the sentinel nodes at the beginning and end of each level? Justify your answer. A correct $\Theta()$ bound (with justification) is worth 5 points.



3. Suppose we have a stack of n pancakes of all different sizes. We want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation we can perform is a *flip* — insert a spatula under the top k pancakes, for some k between 1 and n, turn them all over, and put them back on top of the stack.



Flipping the top three pancakes

   (a) (3 pts) Describe an algorithm to sort an arbitrary stack of n pancakes using flips.

   (b) (3 pts) Prove that your algorithm is correct.

   (c) (2 pts) Exactly how many flips does your sorting algorithm perform in the worst case? A correct $\Theta()$ bound is worth one point.

   (d) (2 pts) Suppose one side of each pancake is burned. Exactly how many flips do you need to sort the pancakes, so that the burned side of every pancake is on the bottom? A correct $\Theta()$ bound is worth one point.

4. Suppose we want to maintain a set of values in a data structure subject to the following operations:

   - INSERT(x): Add x to the set (if it isn't already there).
   - DELETERANGE(a, b): Delete every element x in the range $a \leq x \leq b$. For example, if the set was $\{1, 5, 3, 4, 8\}$, then DELETERANGE$(4, 6)$ would change the set to $\{1, 3, 8\}$.

   Describe and analyze a data structure that supports these operations, such that the amortized cost of either operation is $O(\log n)$. *[Hint: Use a data structure you saw in class. If you use the same* INSERT *algorithm, just say so—you don't need to describe it again in your answer.]*

5. *[1-unit grad students must answer this question.]*

   A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, 'bananaananas' is a shuffle of 'banana' and 'ananas' in several different ways.

   $$\mathsf{banana_{ananas}} \qquad \mathsf{ban_{ana}ana_{nas}} \qquad \mathsf{b_{an}an_{a}a_{na}na_{s}}$$

   The strings 'prodgyrnamammiincg' and 'dyprongarmammicing' are both shuffles of 'dynamic' and 'programming':

   $$\mathsf{pro^{d}g^{y}r^{nam}ammi^{i}n^{c}g} \qquad \mathsf{{}^{dy}pro^{n}g^{a}r^{m}amm^{ic}ing}$$

   Given three strings $A[1\,..\,m]$, $B[1\,..\,n]$, and $C[1\,..\,m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B. For full credit, your algorithm should run in $\Theta(mn)$ time.

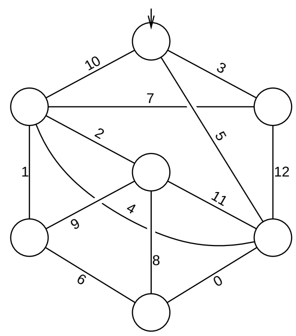
1. Using any method you like, compute the following subgraphs for the weighted graph below. Each subproblem is worth 3 points. Each incorrect edge costs you 1 point, but you cannot get a negative score for any subproblem.

   (a) a depth-first search tree, starting at the top vertex;

   (b) a breadth-first search tree, starting at the top vertex;

   (c) a shortest path tree, starting at the top vertex;

   (d) the **maximum** spanning tree.

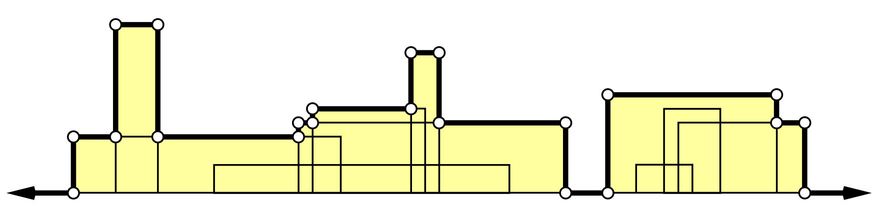


2. (a) *[4 pts]* Prove that a connected acyclic undirected graph with V vertices has exactly $V-1$ edges. ("It's a tree!" is not a proof.)

   (b) *[4 pts]* Describe and analyze an algorithm that determines whether a given undirected graph is a tree, where the graph is represented by an adjacency list.

   (c) *[2 pts]* What is the running time of your algorithm from part (b) if the graph is represented by an adjacency matrix?

3. Suppose we want to sketch the Manhattan skyline (minus the interesting bits like the Empire State and Chrysler builings). You are given a set of n rectangles, each rectangle represented by its left and right x-coordinates and its height. The bottom of each rectangle is on the x-axis. Describe and analyze an efficient algorithm to compute the vertices of the skyline.



A set of rectangles and its skyline. Compute the sequence of white points.

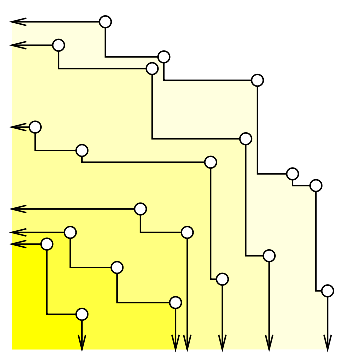
4. Suppose we model a computer network as a weighted undirected graph, where each vertex represents a computer and each edge represents a *direct* network connection between two computers. The weight of each edge represents the *bandwidth* of that connection—the number of bytes that can flow from one computer to the other in one second.[1] We want to implement a point-to-point network protocol that uses a single dedicated path to communicate between any pair of computers. Naturally, when two computers need to communciate, we should use the path with the highest bandwidth. The bandwidth of a *path* is the *minimum* bandwidth of its edges.

   Describe an algorithm to compute the maximum bandwidth path between *every* pair of computers in the network. Assume that the graph is represented as an adjacency list.

5. *[1-unit grad students must answer this question.]*

   Let P be a set of points in the plane. Recall that the *staircase* of P contains all the points in P that have no other point in P both above and to the right. We can define the *staircase layers* of P recursively as follows. The first staircase layer is just the staircase; for all $i > 1$, the ith staircase layer is the staircase of P after the first $i - 1$ staircase layers have been deleted.

   Describe and analyze an algorithm to compute the staircase layers of P in $O(n^2)$ time.[2] Your algorithm should label each point with an integer describing which staircase layer it belongs to. You can assume that no two points have the same x- or y-coordinates.



A set of points and its six staircase layers.

[1] Notice the bandwidth is symmetric; there are no cable modems or wireless phones. Don't worry about systems-level stuff like network load and latency. After all, this is a theory class!

[2] This is *not* the fastest possible running time for this problem.

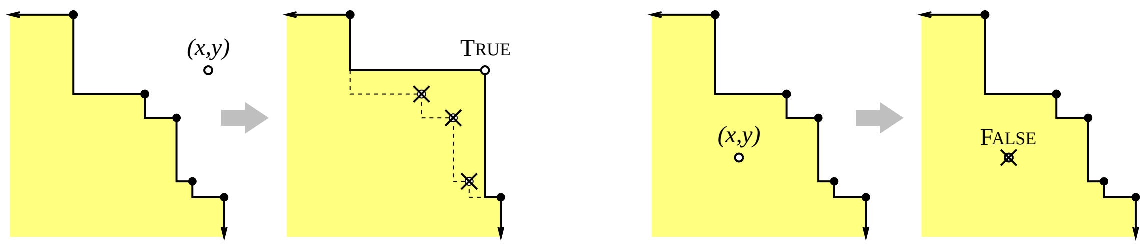
You must turn in this question sheet with your answers.

1. **Déjà vu**

   Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers—if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, and $17 = F_7 + F_4 + F_2$. *You must give a complete, self-contained proof, not just a reference to the posted homework solutions.*

2. **L'esprit d'escalier**

   Recall that the *staircase* of a set of points consists of the points with no other point both above and to the right. Describe a method to maintain the staircase as new points are added to the set. Specifically, describe and analyze a data structure that stores the staircase of a set of points, and an algorithm INSERT(x, y) that adds the point (x, y) to the set and returns TRUE or FALSE to indicate whether the staircase has changed. Your data structure should use $O(n)$ space, and your INSERT algorithm should run in $O(\log n)$ amortized time.



3. **Engage le jeu que je le gagne**

   A palindrome is a text string that is exactly the same as its reversal, such as `DEED`, `RACECAR`, or `SAIPPUAKAUPPIAS`.[1]

   (a) Describe and analyze an algorithm to find the longest *prefix* of a given string that is also a palindrome. For example, the longest palindrome prefix of `ILLINOISURBANACHAMPAIGN` is `ILLI`, and the longest palindrome prefix of `HYAKUGOJYUUICHI`[2] is the single letter `S`. For full credit, your algorithm should run in $O(n)$ time.

   (b) Describe and analyze an algorithm to find the length of the longest *subsequence* of a given string that is also a palindrome. For example, the longest palindrome subsequence of `ILLINOISURBANACHAMPAIGN` is `NIAACAAIN` (or `NIAAHAAIN`), and the longest palindrome subsequence of `HYAKUGOJYUUICHI` is `HUUUH`[3] (or `HUGUH` or `HUYUH` or...). You do not need to compute the actual subsequence; just its length. For full credit, your algorithm should run in $O(n^2)$ time.

[1]Finnish for 'soap dealer'.
[2]Japanese for 'one hundred fifty-one'.
[3]English for 'What the heck are you talking about?'

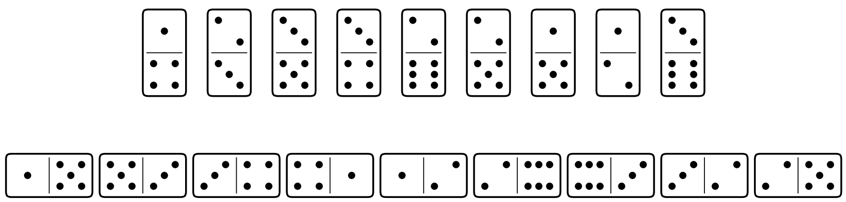
4. **Toute votre base sont appartiennent à nous**

   Prove that *exactly* $2n - 1$ comparisons are required in the worst case to merge two sorted arrays, each with n distinct elements. Describe and analyze an algorithm to prove the upper bound, and use an adversary argument to prove the lower bound. *You must give a complete, self-contained solution, not just a reference to the posted homework solutions.*[4]

5. **Plus ça change, plus ça même chose**

   A domino is a 2×1 rectangle divided into two squares, with a certain number of pips (dots) in each square. In most domino games, the players lay down dominos at either end of a single chain. Adjacent dominos in the chain must have matching numbers. (See the figure below.)

   Describe and analyze an efficient algorithm, or prove that it is NP-hard, to determine whether a given set of n dominos can be lined up in a single chain. For example, for the set of dominos shown below, the correct output is TRUE.



Top: A set of nine dominos
Bottom: The entire set lined up in a single chain

6. **Ceci n'est pas une pipe**

   Consider the following pair of problems:

   - BOXDEPTH: Given a set of n axis-aligned rectangles in the plane and an integer k, decide whether any point in the plane is covered by k or more rectangles.
   - MAXCLIQUE: Given a graph with n vertices and an integer k, decide whether the graph contains a clique with k or more vertices.

   (a) Describe and analyze a reduction of one of these problems to the other.

   (b) MAXCLIQUE is NP-hard. What does your answer to part (a) imply about the complexity of BOXDEPTH?

7. **C'est magique!** *[1-unit graduate students must answer this question.]*

   The recursion fairy's cousin, the reduction genie, shows up one day with a magical gift for you—a box that determines in constant time the size of the largest clique in any given graph. (Recall that a clique is a subgraph where every pair of vertices is joined by an edge.) The magic box does not tell you *where* the largest clique is, only its size. Describe and analyze an algorithm to actually find the largest clique in a given graph **in polynomial time**, using this magic box.

[4]The posted solution for this Homework 5 practice problem was incorrect. So don't use it!

CS 373: Combinatorial Algorithms, Fall 2002
Homework 0, due September 5, 2002 at the beginning of class

Name:		
Net ID:	Alias:	U G

Neatly print your name (first name first, with no comma), your network ID, and an alias of your choice into the boxes above. Circle U if you are an undergraduate, and G if you are a graduate student. **Do not sign your name. Do not write your Social Security number.** Staple this sheet of paper to the top of your homework.

Grades will be listed on the course web site by alias give us, so your alias should not resemble your name or your Net ID. If you don't give yourself an alias, we'll give you one that you won't like.

Before you do anything else, please read the Homework Instructions and FAQ on the CS 373 course web page (http://www-courses.cs.uiuc.edu/˜cs373/hwx/faq.html) and then check the box below. There are 300 students in CS 373 this semester; we are quite serious about giving zeros to homeworks that don't follow the instructions.

☐ I have read the CS 373 Homework Instructions and FAQ.

Every CS 373 homework has the same basic structure. There are six required problems, some with several subproblems. Each problem is worth 10 points. Only graduate students are required to answer problem 6; undergraduates can turn in a solution for extra credit. There are several practice problems at the end. Stars indicate problems we think are hard.

This homework tests your familiarity with the prerequisite material from CS 173, CS 225, and CS 273, primarily to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Rosen (the 173/273 textbook), CLRS (especially Chapters 1–7, 10, 12, and A–C), and the lecture notes on recurrences should be sufficient review, but you may want to consult other texts as well.

Required Problems

1. Sort the following functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. Please *don't* turn in proofs, but you should do them anyway to make sure you're right (and for practice).

$$\begin{array}{ccccc} 1 & n & n^2 & \lg n & n \lg n \\ n^{\lg n} & (\lg n)^n & (\lg n)^{\lg n} & n^{\lg \lg n} & n^{1/\lg n} \\ \log_{1000} n & \lg^{1000} n & \lg^{(1000)} n & \lg(n^{1000}) & \left(1 + \frac{1}{1000}\right)^n \end{array}$$

   To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions n^2, n, $\binom{n}{2}$, n^3 could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

2. Solve these recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Please *don't* turn in proofs, but you should do them anyway just for practice. Assume reasonable but nontrivial base cases, and state them if they affect your solution. Extra credit will be given for more exact solutions. *[Hint: Most of these are* <u>*very*</u> *easy.]*

$$\begin{aligned} A(n) &= 2A(n/2) + n & F(n) &= 9F(\lfloor n/3 \rfloor + 9) + n^2 \\ B(n) &= 3B(n/2) + n & G(n) &= 3G(n-1)/5G(n-2) \\ C(n) &= 2C(n/3) + n & H(n) &= 2H(\sqrt{n}) + 1 \\ D(n) &= 2D(n-1) + 1 & I(n) &= \min_{1 \le k \le n/2} \big(I(k) + I(n-k) + k\big) \\ E(n) &= \max_{1 \le k \le n/2} \big(E(k) + E(n-k) + n\big) & {}^*J(n) &= \max_{1 \le k \le n/2} \big(J(k) + J(n-k) + k\big) \end{aligned}$$

3. Recall that a binary tree is *full* if every node has either two children (an internal node) or no children (a leaf). Give at least *four different* proofs of the following fact:

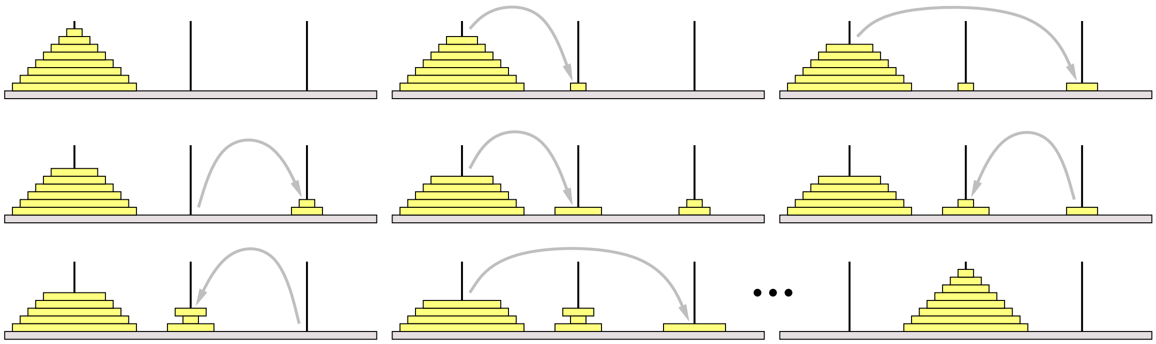
   > *In any full binary tree, the number of leaves is exactly one more than the number of internal nodes.*

   For full credit, each proof must be self-contained, the proof must be substantially different from each other, and at least one proof must *not* use induction. For each n, your nth correct proof is worth n points, so you need four proofs to get full credit. Each correct proof beyond the fourth earns you extra credit. *[Hint: I know of at least* <u>*six*</u> *different proofs.]*

4. Most of you are probably familiar with the story behind the Tower of Hanoï puzzle:[1]

> At the great temple of Benares, there is a brass plate on which three vertical diamond shafts are fixed. On the shafts are mounted n golden disks of decreasing size.[2] At the time of creation, the god Brahma placed all of the disks on one pin, in order of size with the largest at the bottom. The Hindu priests unceasingly transfer the disks from peg to peg, one at a time, never placing a larger disk on a smaller one. When all of the disks have been transferred to the last pin, the universe will end.

Recently the temple at Benares was relocated to southern California, where the monks are considerably more laid back about their job. At the "Towers of Hollywood", the golden disks were replaced with painted plywood, and the diamond shafts were replaced with Plexiglas. More importantly, the restriction on the order of the disks was relaxed. While the disks are being moved, the *bottom* disk on any pin must be the *largest* disk on that pin, but disks further up in the stack can be in any order. However, after all the disks have been moved, they must be in sorted order again.



The Towers of Hollywood.

Describe an algorithm[3] that moves a stack of n disks from one pin to the another using the smallest possible number of moves. For full credit, your algorithm should be non-recursive, but a recursive algorithm is worth significant partial credit. *Exactly* how many moves does your algorithm perform? *[Hint: The Hollywood monks can bring about the end of the universe quite a bit faster than the original monks at Benares could.]*

The problem of computing the minimum number of moves was posed in the most recent issue of the *American Mathematical Monthly* (August/September 2002). No solution has been published yet.

[1]The puzzle and the accompanying story were both invented by the French mathematician Eduoard Lucas in 1883. See http://www.cs.wm.edu/~pkstoc/toh.html

[2]In the original legend, $n = 64$. In the 1883 wooden puzzle, $n = 8$.

[3]Since you've read the Homework Instructions, you know exactly what this phrase means.

5. On their long journey from Denmark to England, Rosencrantz and Guildenstern amuse themselves by playing the following game with a fair coin. First Rosencrantz flips the coin over and over until it comes up tails. Then Guildenstern flips the coin over and over until he gets as many heads in a row as Rosencrantz got on his turn. Here are three typical games:

   Rosencrantz: H H T
   Guildenstern: H T H H

   Rosencrantz: T
   Guildenstern: (no flips)

   Rosencrantz: H H H T
   Guildenstern: T H H T H H T H T H H H

   (a) What is the expected number of flips in one of Rosencrantz's turns?

   (b) Suppose Rosencrantz flips k heads in a row on his turn. What is the expected number of flips in Guildenstern's next turn?

   (c) What is the expected total number of flips (by both Rosencrantz and Guildenstern) in a single game?

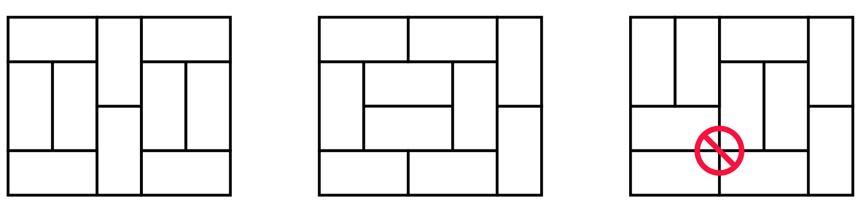
   Prove your answers are correct. If you have to appeal to "intuition" or "common sense", your answer is almost certainly wrong! You must give exact answers for full credit, but asymptotic bounds are worth significant partial credit.

6. *[This problem is required only for graduate students (including I2CS students); undergrads can submit a solution for extra credit.]*

   Tatami are rectangular mats used to tile floors in traditional Japanese houses. Exact dimensions of tatami mats vary from one region of Japan to the next, but they are always twice as long in one dimension than in the other. (In Tokyo, the standard size is 180cm×90cm.)

   (a) How many different ways are there to tile a $2 \times n$ rectangular room with 1×2 tatami mats? Set up a recurrence and derive an *exact* closed-form solution. *[Hint: The answer involves a familiar recursive sequence.]*

   (b) According to tradition, tatami mats are always arranged so that four corners never meet. Thus, the first two arrangements below are traditional, but not the third.



Two traditional tatami arrangements and one non-traditional arrangement.

   How many different *traditional* ways are there to tile a $3 \times n$ rectangular room with 1×2 tatami mats? Set up a recurrence and derive an *exact* closed-form solution.

   *(c) **[5 points extra credit]** How many different *traditional* ways are there to tile an $n \times n$ square with 1×2 tatami mats? Prove your answer is correct.

Practice Problems

These problems are only for your benefit; other problems can be found in previous semesters' homeworks on the course web site. You are *strongly* encouraged to do some of these problems as additional practice. Think of them as potential exam questions (hint, hint). Feel free to ask about any of these questions on the course newsgroup, during office hours, or during review sessions.

1. Removing any edge from a binary tree with n nodes partitions it into two smaller binary trees. If both trees have at least $\lceil (n-1)/3 \rceil$ nodes, we say that the partition is *balanced*.

   (a) Prove that every binary tree with more than one vertex has a balanced partition. *[Hint: I know of at least two different proofs.]*

   (b) If each smaller tree has more than $\lfloor n/3 \rfloor$ nodes, we say that the partition is *strictly balanced*. Show that for every n, there is an n-node binary tree with *no* strictly balanced partition.

2. Describe an algorithm $\textsc{CountToTenToThe}(n)$ that prints the integers from 1 to 10^n.

   Assume you have a subroutine $\textsc{PrintDigit}(d)$ that prints any integer d between 0 and 9, and another subroutine $\textsc{PrintSpace}$ that prints a space character. Both subroutines run in $O(1)$ time. You may want to write (and analyze) a separate subroutine $\textsc{PrintInteger}$ to print an arbitrary integer.

   Since integer variables cannot store arbitrarily large values in most programming languages, your algorithm must not store any value larger than $\max\{10, n\}$ in any single integer variable. Thus, the following algorithm is **not correct**:

```
BOGUSCOUNTTOTENTOTHE(n):
  for i ← 1 to POWER(10, n)
      PRINTINTEGER(i)
```

   (So what exactly *can* you pass to $\textsc{PrintInteger}$?)

   What is the running time of your algorithm (as a function of n)? How many digits and spaces does it print? How much space does it use?

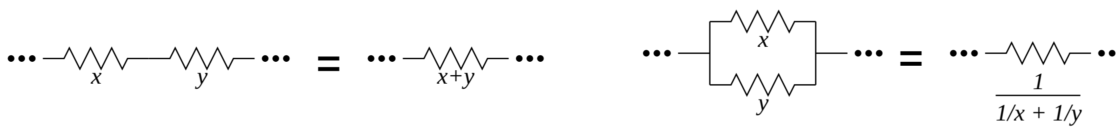
3. I'm sure you remember the following simple rules for taking derivatives:

   - Simple cases: $\frac{d}{dx}\alpha = 0$ for any constant α, and $\frac{d}{dx}x = 1$
   - Linearity: $\frac{d}{dx}(f(x) + g(x)) = f'(x) + g'(x)$
   - The product rule: $\frac{d}{dx}(f(x) \cdot g(x)) = f'(x) \cdot g(x) + f(x) \cdot g'(x)$
   - The chain rule: $\frac{d}{dx}(f(g(x)) = f'(g(x)) \cdot g'(x)$

   Using *only* these rules and induction, prove that $\frac{d}{dx}x^c = cx^{c-1}$ for any integer $c \neq -1$. Do not use limits, integrals, or any other concepts from calculus, except for the simple identities listed above. *[Hint: Don't forget about negative values of c!]*

4. This problem asks you to calculate the total resistance between two points in a series-parallel resistor network. Don't worry if you haven't taken a circuits class; everything you need to know can be summed up in two sentences and a picture.

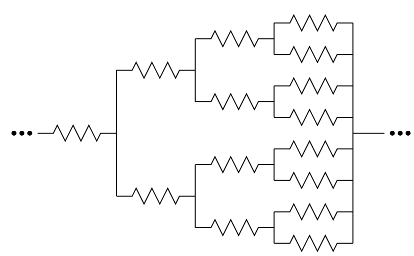
   - The total resistance of two resistors in *series* is the sum of their individual resistances.
   - The total resistance of two resistors in *parallel* is the reciprocal of the sum of the reciprocals of their individual resistances.



Equivalence laws for series-parallel resistor networks.

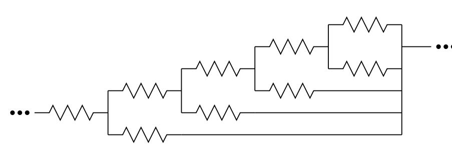
What is the *exact* total resistance[4] of the following resistor networks as a function of n? Prove your answers are correct. *[Hint: Use induction. Duh.]*

(a) A complete binary tree with depth n, with a 1Ω resistor at every node, and a common wire joining all the leaves. Resistance is measured between the root and the leaves.



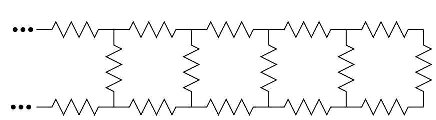
A balanced binary resistor tree with depth 3.

(b) A totally unbalanced full binary tree with depth n (every internal node has two children, one of which is a leaf) with a 1Ω resistor at every node, and a common wire joining all the leaves. Resistance is measured between the root and the leaves.



A totally unbalanced binary resistor tree with depth 4.

*(c) A ladder with n rungs, with a 1Ω resistor on every edge. Resistance is measured between the bottom of the legs.



A resistor ladder with 5 rungs.

[4]The ISO standard unit of resistance is the Ohm, written with the symbol Ω. Don't confuse this with the asymptotic notation $\Omega(f(n))$!

CS 373: Combinatorial Algorithms, Fall 2002
Homework 1, due September 17, 2002 at 23:59:59

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Undergrads	Grads

This homework is to be submitted in groups of up to three people. Graduate and undergraduate students are *not* allowed to work in the same group. Please indicate above whether you are undergraduate or graduate students. Only *one* submission per group will be accepted.

Required Problems

1. The traditional Devonian/Cornish drinking song "The Barley Mow" has the following pseudolyrics, where *container*$[i]$ is the name of a container [1] that holds 2^i ounces of beer.

```
BarleyMow(n):
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"

    "We'll drink it out of the jolly brown bowl,"
    "Here's a health to the barley-mow!"
    "Here's a health to the barley-mow, my brave boys,"
    "Here's a health to the barley-mow!"

    for i ← 1 to n
        "We'll drink it out of the container[i], boys,"
        "Here's a health to the barley-mow!"
        for j ← i downto 1
            "The container[j],"
        "And the jolly brown bowl!"
        "Here's a health to the barley-mow, my brave boys,"
        "Here's a health to the barley-mow!"
```

   (a) Suppose each container name *container*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

[1] One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.

(b) Suppose *container*$[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)

(c) Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *container*$[i]$. Assuming for purposes of this problem that you are over 21, *exactly* how many ounces of beer would you drink if you sang BARLEYMOW(n)? (Give an *exact* answer, not just an asymptotic bound.)

2. Suppose you have a set S of n numbers. Given two elements you *cannot* determine which is larger. However, you are given an oracle that will tell you the median of a set of three elements.

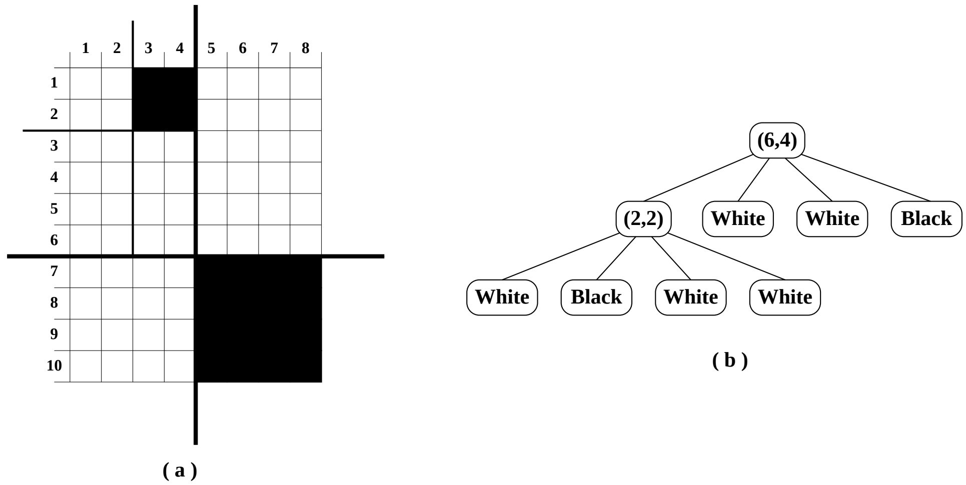
   (a) Give a linear time algorithm to find the pair of the largest and smallest numbers in S.

   (b) Give an algorithm to sort S in $O(n \lg n)$ time.

3. Given a black and white pixel image $A[1 \ldots m][1 \ldots n]$, our task is to represent A with a search tree T. Given a query (x, y), a simple search on T should return the color of pixel $A[x][y]$. The algorithm to construct T will be as follows.

```
ConstructSearchTree(A[1...m][1...n]):
    //Base Case
    if A contains only one color
        return a leaf node labeled with that color

    //Recurse on Subtrees
    (i, j) ← ChooseCut(A[1...m][1...n])
    T1 ← ConstructSearchTree(A[1...i][1...j])
    T2 ← ConstructSearchTree(A[1...i][j+1...n])
    T3 ← ConstructSearchTree(A[i+1...m][1...j])
    T4 ← ConstructSearchTree(A[i+1...m][j+1...n])

    //Construct the Root
    T.cut ← (i, j)
    T.children ← T1, T2, T3, T4
    return T
```

That is, this algorithm divides a multicolor image into quadrants and recursively constructs the search tree for each quadrant. Upon a query (x, y) of T (assuming $1 \leq x \leq m$ and $1 \leq y \leq n$), the appropriate subtree is searched. When the correct leaf node is reached, the pixel color is returned. Here's a toy example.

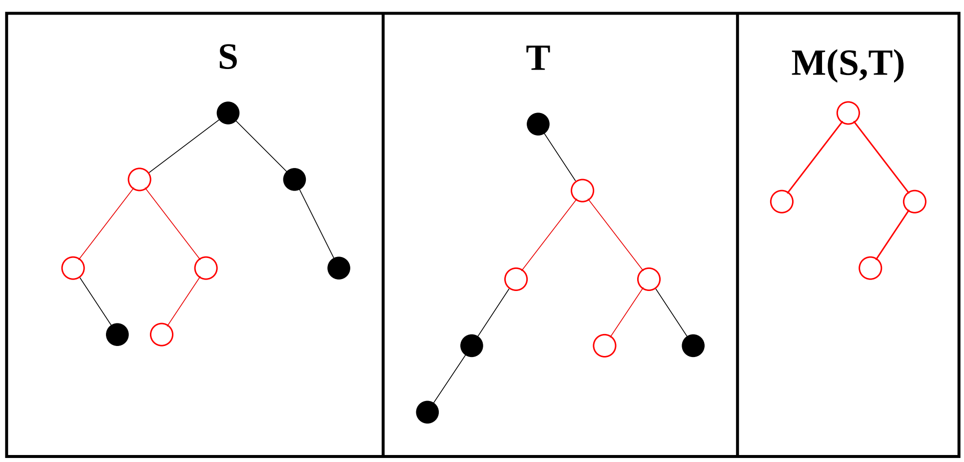


(a) An image A and the chosen cuts. (b) The corresponding search tree.

Your job in this problem is to give an algorithm for ChooseCut. The sequence of chosen cuts must result in an optimal search tree T. That is, the expected search depth of a uniformly chosen pixel must be minimized. You may use any external data structures ($i.e.$a global table) that you find necessary. You may also preprocess in order to initialize these structures before the initial call to ConstructSearchTree($A[1 \ldots m][1 \ldots n]$).

4. Let A be a set of n positive integers, all of which are no greater than some constant $M > 0$. Give an $O(n^2M)$ time algorithm to determine whether or not it is possible to split A into two subsets such that the sum of the numbers in each subset are equal.

5. Let S and T be two binary trees. A *matching* of S and T is a tree M which is isomorphic to some subtree in each of S and T. Here's an illustration.



A matching $M(S,T)$ of binary trees S and T.

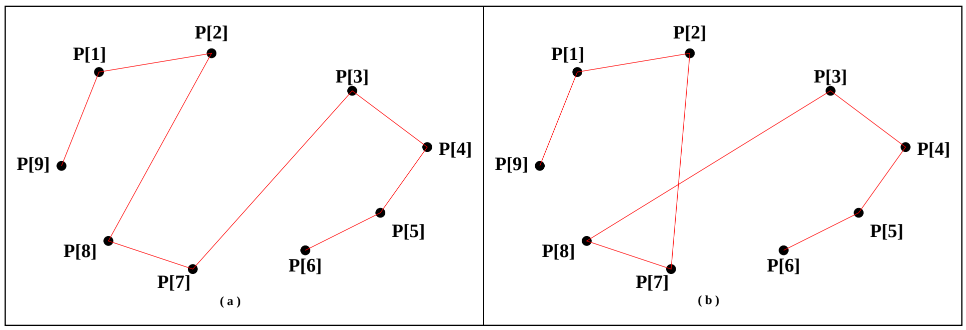
A *maximal* matching is a matching which contains at least as many vertices as any other matching. Give an algorithm to compute a maximal matching given the roots of two binary trees. Your algorithm should return the size of the match as well as the two roots of the matched subtrees of S and T.

6. *[This problem is required only for graduate students (including I2CS students); undergrads can submit a solution for extra credit.]*

   Let $P[1, \ldots, n]$ be a set of n *convex* points in the plane. Intuitively, if a rubber band were stretched to surround P then each point would touch the rubber band. Furthermore, suppose that the points are labeled such that $P[1], \ldots, P[n]$ is a simple path along the convex hull ($i.e. P[i]$ is adjacent to $P[i+1]$ along the rubber band).

   (a) Give a simple algorithm to compute a shortest *cyclic* tour of P.

   (b) A *monotonic* tour of P is a tour that never crosses itself. Here's an illustration.



(a) A monotonic tour of P. (b) A non-monotonic tour of P.

   Prove that any shortest tour of P must be monotonic.

   (c) Given an algorithm to compute a shortest tour of P starting at point $P[1]$ and finishing on point $P[\lfloor \frac{n}{2} \rfloor]$.

Practice Problems

These remaining practice problems are entirely for your benefit. Don't turn in solutions—we'll just throw them out—but feel free to ask us about these questions during office hours and review sessions. Think of these as potential exam questions (hint, hint).

1. Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:

   (a) the square immediately above,

   (b) the square that is one up and one left (but only if the checker is not already in the leftmost column),

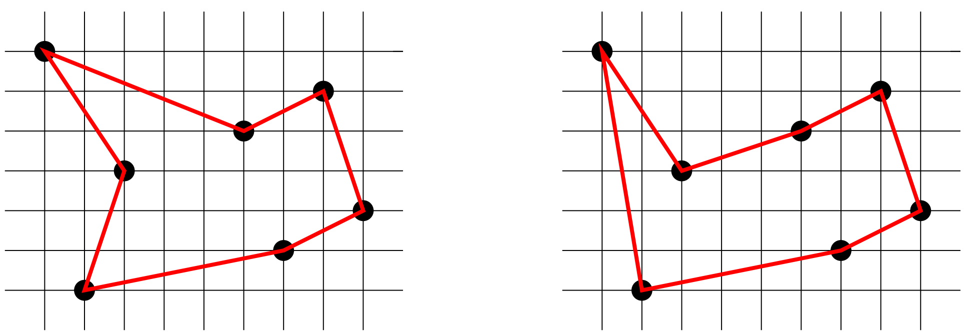
   (c) the square that is one up and one right (but only if the checker is not already in the rightmost column).

   Each time you move from square x to square y, you receive $p(x, y)$ dollars. You are given $p(x, y)$ for all pairs (x,y) for which a move from x to y is legal. Do not assume that $p(x, y)$ is positive.

   Give an algorithm that figures out the set of moves that will move the checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible. Your algorithm is free to pick any square along the bottom edge as a starting point and any square along the top edge as a destination in order to maximize the number of dollars gathered along the way. What is the running time of your algorithm?

2. (CLRS 15-1) The *euclidean traveling-salesman problem* is the problem of determining the shortest closed tour that connects a given set of n points in the plane. Figure (a) below shows the solution to a 7-point problem. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time.

   J.L. Bentley has suggested that we simplify the problem by restricting our attention to *bitonic tours* (Figure (b) below). That is, tours that start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right to left back to the starting point. In this case, a polynomial-time algorithm is possible.



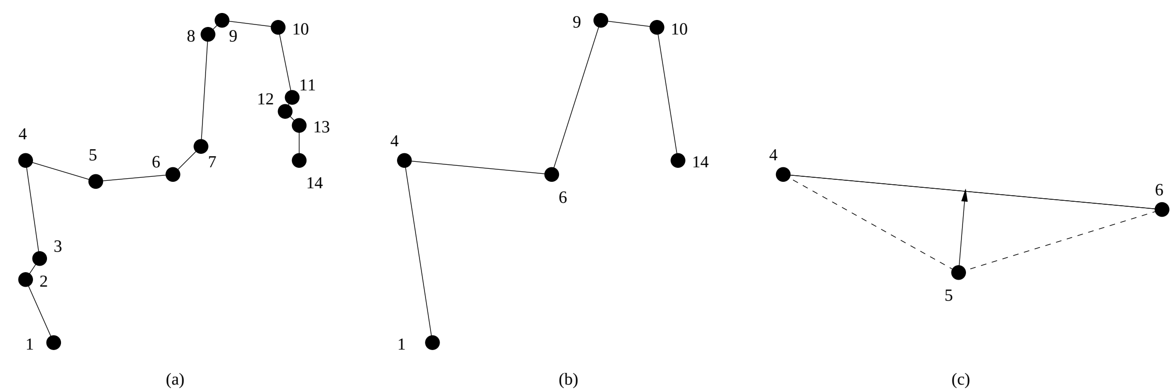
(a) (b)

Seven points in the plane, shown on a unit grid. (a) The shortest closed tour, with length approximately 24.89. This tour is not bitonic. (b) The shortest bitonic tour for the same set of points. It's length is approximately 25.58.

Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate. [Hint: Scan left to right, maintaining optimal possibilities for the two parts of the tour.]

3. You are given a polygonal line γ made out of n vertices in the plane. Namely, you are given a list of n points in the plane $p_1, \ldots, p_n$, where $p_i = (x_i, y_i)$. You need to display this polygonal line on the screen, however, you realize that you might be able to draw a polygonal line with considerably less vertices that looks identical on the screen (because of the limited resolution of the screen). It is crucial for you to minimize the number of vertices of the polygonal line. (Because, for example, your display is a remote Java applet running on the user computer, and for each vertex of the polygon you decide to draw, you need to send the coordinates of the points through the network which takes a *long long long time*. So the fewer vertices you send, the snappier your applet would be.)

   So, given such a polygonal line γ, and a parameter k, you would like to select k vertices of γ that yield the "best" polygonal line that looks like γ.



(a) The original polygonal line with 14 vertices. (b) A new polygonal line with 6 vertices. (c) The distance between p_5 on the original polygonal line and the simplification segment p_4p_6. The error of p_5 is $\mathrm{error}(p_5) = \mathrm{dist}(p_5, p_4p_6)$.

   Namely, you need to build a new polygonal line γ' and minimize the difference between the two polygonal-lines. The polygonal line γ' is built by selecting k vertices $\{p_{i_1}, p_{i_2}, ..., p_{i_k}\}$ from γ. It is required that $i_1 = 1$, $i_k = n$, and $i_j < i_{j+1}$ for $j = 1, 2, ..., k-1$.

   We define the error between γ and γ' by how far from γ' are the vertices of γ. More formally, The difference between the two polygonal lines is

$$error(\gamma, \gamma') = \sum_{j=1}^{k-1} \sum_{m=i_j+1}^{i_{j+1}-1} \mathrm{dist}(p_m, p_{i_j}p_{i_{j+1}}).$$

   Namely, for every vertex not in the simplification, its associated error, is the distance to the corresponding simplified segment (see (c) in above figure). The overall error is the sum over all vertices.

   You can assume that you are provided with a subroutine that can calculate $\mathrm{dist}(u, vw)$ in constant time, where $\mathrm{dist}(u, vw)$ is the distance between the point u and the segment vw.

   Give an $O(n^3)$ time algorithm to find the γ' that minimizes $\mathrm{error}(\gamma, \gamma')$.

CS 373: Combinatorial Algorithms, Fall 2002
Homework 2 (due Thursday, September 26, 2002 at 11:59:59 p.m.)

Name:		
Net ID:	Alias:	U G

Name:		
Net ID:	Alias:	U G

Name:		
Net ID:	Alias:	U G

Homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade. Since graduate students are required to solve problems that are worth extra credit for other students, **Grad students may not be on the same team as undergraduates.**

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate or 1-unit grad student by circling U or G, respectively. Staple this sheet to the top of your homework. **NOTE: You must use different sheet(s) of paper for each problem assigned.**

Required Problems

1. For each of the following problems, the input is a set of n nuts and n bolts. For each bolt, there is exactly one nut of the same size. Direct comparisons between nuts or between bolts are not allowed, but you can compare a nut and a bolt in constant time.

   (a) Describe and analyze a deterministic algorithm to find the largest bolt. *Exactly* how many comparisons does your algorithm perform in the worst case? *[Hint: This is very easy.]*

   (b) Describe and analyze a randomized algorithm to find the largest bolt. What is the *exact* expected number of comparisons performed by your algorithm?

   (c) Describe and analyze an algorithm to find the largest and smallest bolts. Your algorithm can be either deterministic or randomized. What is the *exact* worst-case expected number of comparisons performed by your algorithm? *[Hint: Running part (a) twice is definitely not the most efficient algorithm.]*

   In each case, to receive **full** credit, you need to describe the most efficient algorithm possible.

2. Consider the following algorithm:

```
SLOWSHUFFLE(A[1..n]) :
    for i ← 1 to n
        B[i] ←Null
    for i ← 1 to n
        index ← Random(1,n)
        while B[index] ≠ Null
            index ← Random(1,n)
        B[index] ← A[i]
    for i ← 1 to n
        A[i] ← B[i]
```

Suppose that Random(i,j) will return a random number between i and j inclusive in constant time. SLOWSHUFFLE will shuffle the input array into a random order such that every permutation is equally likely.

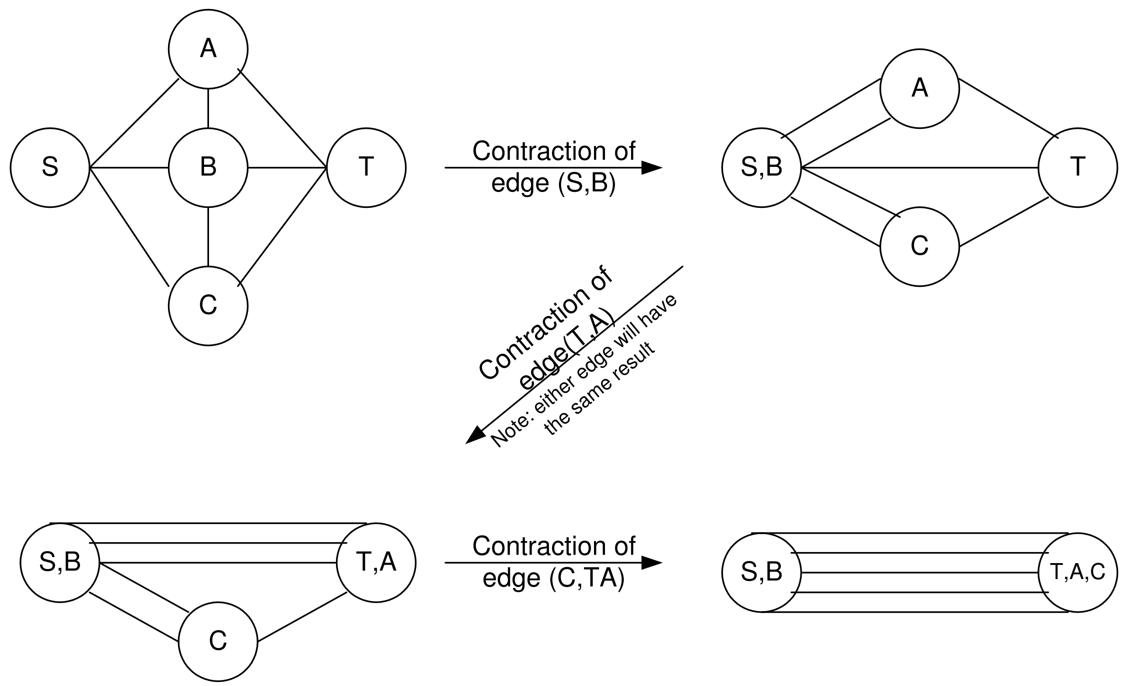
(a) What is the expected running time of the above algorithm. Justify your answer and give a tight asymptotic bound.

(b) Describe an algorithm that randomly shuffles an n-element array, so that every permutation is *equally* likely, in $O(n)$ time.

3. Suppose we are given an undirected graph $G = (V, E)$ together with two distinguished vertices s and t. An **s-t min-cut** is a set of edges that once removed from the graph, will disconnect s from t. We want to find such a set with the minimum cardinality(The smallest number of edges). In other words, we want to find the smallest set of edges that will seperate s and t

To do this we repeat the following step $|V| - 2$ times: Uniformly at random, pick an edge from the set E which contains all edges in the graph excluding those that directly connects vertices s and t. Merge the two vertices that is connected by this randomly selected edge. If as a result there are several edges between some pair of vertices, retain them all. Edges that are between the two merged vertices are removed so that there are never any self-loops. We refer to this process of merging the two end-points of an edge into a single vertex as the *contraction* of that edge. Notice with each contraction the number of vertices of G decreases by one.

As this algorithm proceeds, the vertex s may get merged with a new vertex as the result of an edge being contracted. We call this vertex the s-vertex. Similarly, we have a t-vertex. During the contraction algorithm, we ensure that we never contract an edge between the s-vertex and the t-vertex.



(a) Give an example of a graph in which the probability that this algorithm finds an s-t min-cut is exponentially small($O(1/a^n)$). Justify your answers.
(Hint: Think multigraphs)

(b) Give an example of a graph such that there are $O(2^n)$ number of s-t min-cuts. Justify your answers.

4. Describe a modification of treaps that supports the following operations, each in $O(\log n)$ expected time:

- INSERT(x): Insert a new element x into the data structure.
- DELETE(x): Delete an element x from the data structure.
- COMPUTERANK(x): Return the number of elements in the data structure less than or equal to x.
- FINDBYRANK(r): Return the kth smallest element in the data structure.

Describe and analyze the algorithms that implement each of these operations. *[Hint: Don't reinvent the wheel!]*

5. A *meldable priority queue* stores a set of keys from some totally ordered universe (such as the integers) and supports the following operations:

   - MAKEQUEUE: Return a new priority queue storing the empty set.
   - FINDMIN(Q): Return the smallest element stored in Q (if any).
   - DELETEMIN(Q): Delete the smallest element stored in Q (if any).
   - INSERT(Q, x): Insert element x into Q.
   - MELD(Q_1, Q_2): Return a new priority queue containing all the elements stored in Q_1 and Q_2. The component priority queues are destroyed.
   - DECREASEKEY(Q, x, y): Replace an element x of Q with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q storing x.
   - DELETE(Q, x): Delete an element $x \in Q$. The input is a pointer directly to the node in Q storing x.

   A simple way to implement this data structure is to use a heap-ordered binary tree, where each node stores an element, a pointer to its left child, a pointer to its right child, and a pointer to its parent. MELD(Q_1, Q_2) can be implemented with the following randomized algorithm.

   - If either one of the queues is empty, return the other one.
   - If the root of Q_1 is smaller than the root of Q_2, then recursively MELD Q_2 with either right(Q_1) or left(Q_1), each with probability 1/2.
   - Similarly, if the root of Q_2 is smaller than the root of Q_1, then recursively MELD Q_1 with a randomly chosen child of Q_2.

   (a) Prove that for *any* heap-ordered trees Q_1 and Q_2, the expected running time of MELD(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: How long is a random path in an n-node binary tree, if each left/right choice is made with equal probability?]* For extra credit, prove that the running time is $O(\log n)$ *with high probability.*

   (b) Show that each of the operations DELETEMIN, INSERT, DECREASEKEY, and DELETE can be implemented with one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ with high probability.)

6. *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

   The following randomized algorithm selects the rth smallest element in an unsorted array $A[1,..,n]$. For example, to find the smallest element, you would call RANDOMSELECT(A, 1); to find the median element, you would call RANDOMSELECT($A, \lfloor n/2 \rfloor$). Recall from lecture that PARTITION splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n - 1$ comparisons altogether, and returns the new index of the pivot element.

```
RandomSelect(A[1 .. n], r):
    p ← Random(1, n)
    k ← Partition(A[1 .. n], p)

    if r < k
        return RandomSelect(A[1 .. k − 1], r)
    else if r > k
        return RandomSelect(A[k + 1 .. n], r − k)
    else
        return A[k]
```

(a) State a recurrence for the expected running time of RandomSelect, as a function of both n and r.

(b) What is the *exact* probability that RandomSelect compares the ith smallest and jth smallest elements in the input array? The correct answer is a simple function of i, j, and r. *[Hint: Check your answer by trying a few small examples.]*

(c) Show that for any n and r, the expected running time of RandomSelect is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b). For extra credit, find the *exact* expected number of comparisons, as a function of n and r.

(d) What is the expected number of times that RandomSelect calls itself recursively?

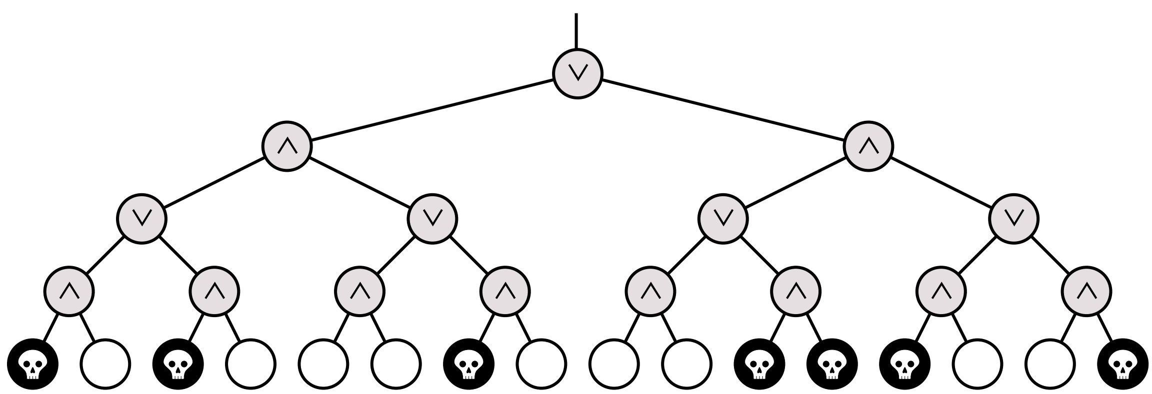
Practice Problems

1. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.

   You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are *and* gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

   (a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

   (b) Unfortunately, Death won't let you even look at every node in the tree. Describe a *randomized* algorithm that determines whether you can win in $\Theta(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*



2. WHat is the *exact* number of nodes in a skip list storing n keys, *not* counting the sentinel nodes at the beginning and end of each level? Justify your answer.

3. Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B. (For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, our algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicates. Your algorithm should be able to run in $\Theta(\log n)$ time. *[Hint: First try to solve the special case $k = n$.]*

CS 373: Combinatorial Algorithms, Fall 2002
Homework 3, due October 17, 2002 at 23:59:59

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Undergrads	Grads

This homework is to be submitted in groups of up to three people. Graduate and undergraduate students are *not* allowed to work in the same group. Please indicate above whether you are undergraduate or graduate students. Only *one* submission per group will be accepted.

Required Problems

1. (a) Prove that only one subtree gets rebalanced in a scapegoat tree insertion.

   (b) Prove that $I(v) = 0$ in every node of a perfectly balanced tree. (Recall that $I(v) = \max\{0, |T| - |s| - 1\}$, where T is the child of greater height and s the child of lesser height, and $|v|$ is the number of nodes in subtree v. A perfectly balanced tree has two perfectly balanced subtrees, each with as close to half the nodes as possible.)

   *(c) Show that you can rebuild a fully balanced binary tree from an unbalanced tree in $O(n)$ time using only $O(\log n)$ additional memory.

2. Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than 1/4 full, we we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method (it makes it much more difficult).

3. A stack is a FILO/LIFO data structure that represents a stack of objects; access is only allowed at the top of the stack. In particular, a stack implements two operations:

   - PUSH(x): adds x to the top of the stack.

- POP: removes the top element and returns it.

A queue is a FIFO/LILO data structure that represents a row of objects; elements are added to the front and removed from the back. In particular, a queue implements two operations:

- ENQUEUE(x): adds x to the front of the queue.
- DEQUEUE: removes the element at the back of the queue and returns it.

Using two stacks and no more than O(1) additional space, show how to simulate a queue for which the operations ENQUEUE and DEQUEUE run in constant amortized time. You should treat each stack as a black box (i.e., you may call PUSH and POP, but you do not have access to the underlying stack implementation). Note that each PUSH and POP performed by a stack takes $O(1)$ time.

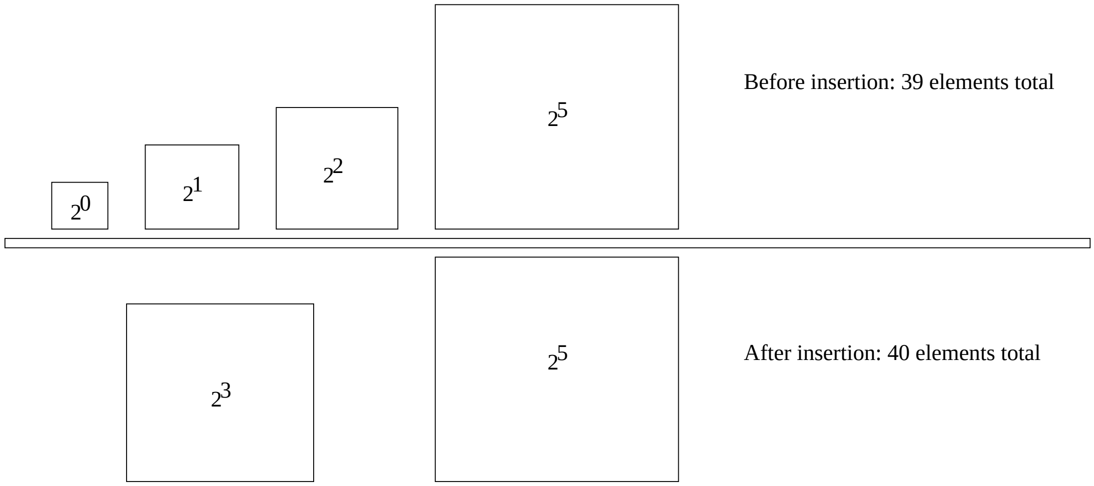
4. A data structure is *insertion-disabled* if there is no way to add elements to it. For the purposes of this problem, further assume that an insertion-disabled data structure implements the following operations with the given running times:

   - INITIALIZE(S): Return an insertion-disabled data structure that contains the elements of S. Running time: $O(n \log n)$.
   - SEARCH(D, x): Return TRUE if x is in D; return FALSE if not. Running time: $O(\log n)$.
   - RETURNALL(D, x): Return an unordered set of all elements in D. Running time: $O(n)$.
   - DELETE(D, x): Remove x from D if x is in D. Running time: $O(\log n)$.

   Using an approach known as the Bentley-Saxe Logarithmic Method (BSLM), it is possible to represent a dynamic (i.e., supports insertions) data structure with a collection of insertion-disabled data structures, where each insertion-disabled data structure stores a number of elements that is a distinct power of two. For example, to store $39 = 2^0 + 2^1 + 2^2 + 2^5$ elements in a BSLM data structure, we use four insertion-disabled data structures with 2^0, 2^1, 2^2, and 2^5 elements.

   To find an element in a BSLM data structure, we search the collection of insertion-disabled data structures until we find (or don't find) the element.

   To insert an element into a BSLM data structure, we think about adding a 2^0-size insertion-disabled data structure. However, an insertion-disabled data structure with 2^0 elements may already exist. In this case, we can combine two 2^0-size structures into a single 2^1-size structure. However, there may already be a 2^1-size structure, so we will need to repeat this process. In general, we do the following: Find the smallest i such that for all nonnegative $k < i$, there is a 2^k-sized structure in our collection. Create a 2^i-sized structure that contains the element to be inserted and all elements from 2^k-sized data structures for all $k < i$. Destroy all 2^k-sized data structures for $k < i$.



Inserting an element into a BSLM data structure.

We delete elements from the BSLM data structure lazily. To delete an element, we first search the collection of insertion-disabled data structures for it. Then we call DELETE to remove the element from its insertion-disabled data structure. This means that a 2^i-sized insertion-disabled data structure might store less than 2^i elements. That's okay; we just say that it stores 2^i elements and say that 2^i is its *pretend* size. We keep track of a single variable, called $Waste$, which is initially 0 and is incremented by 1 on each deletion. If $Waste$ exceeds three-quarters of the total pretend size of all insertion-disabled data structures in our collection (i.e., the total number of elements stored), we rebuild our collection of insertion-disabled data structures. In particular, we create a 2^m-sized insertion-disabled data structure, where 2^m is the smallest power that is greater than or equal to the total number of elements stored. All elements are stored in this 2^m-sized insertion-disabled data structure, and all other insertion-disabled data structures in our collection are destroyed. $Waste$ is reset to $2^m - n$, where n is the total number of elements stored in the BSLM data structure.

Your job is to prove the running times of the following three BSLM operations:

- SEARCHBSLM(D, x): Search for x in the collection of insertion-disabled data structures that represent the BSLM data structure D. Running time: $O(\log^2 n)$ worst-case.
- INSERTBSLM(D, x): Insert x into the collection of insertion-disabled data structures that represent the BSLM data structure D, modifying the collection as necessary. Running time: $O(\log^2 n)$ amortized.
- DELETEBSLM(D, x): Delete x from the collection of insertion-disabled data structures that represent the BSLM data structure D, rebuilding when there is a lot of wasted space. Running time: $O(\log^2 n)$ amortized.

5. Except as noted, the following sub-problems refer to a Union-Find data structure that uses both path compression and union by rank.

   (a) Prove that in a set of n elements, a sequence of n consecutive FIND operations takes $O(n)$ total time.

   (b) Show that any sequence of m MAKESET, FIND, and UNION operations takes only $O(m)$ time if all of the UNION operations occur before any of the FIND operations.

(c) Now consider part b with a Union-Find data structure that uses path compression but does *not* use union by rank. Is $O(m)$ time still correct? Prove your answer.

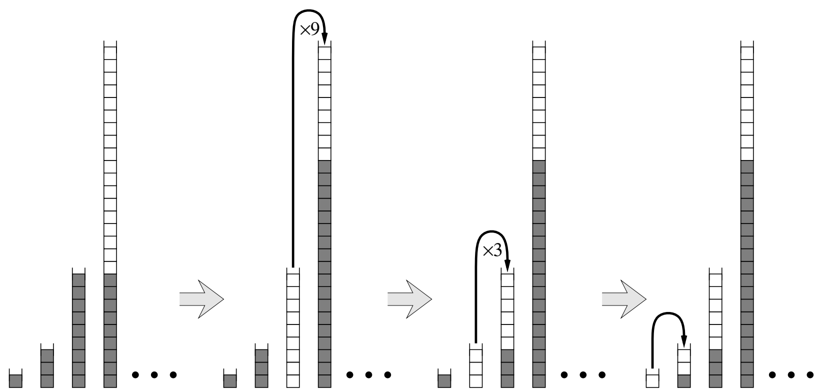
6. *[This problem is required only for graduate students (including I2CS students); undergrads can submit a solution for extra credit.]*

   Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of 'fits', where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string.]

Practice Problems

These remaining practice problems are entirely for your benefit. Don't turn in solutions—we'll just throw them out—but feel free to ask us about these questions during office hours and review sessions. Think of these as potential exam questions (hint, hint).

1. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i, we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2}, and so on. Moving a single element from one stack to the next takes $O(1)$ time.



Making room for one new element in a multistack.

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?

   (b) Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack. You can use any method you like.

2. A hash table of size m is used to store n items with $n \leq m/2$. Open addressing is used for collision resolution.

   (a) Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability that the i^{th} insertion requires strictly more than k probes is at most 2^{-k}.

   (b) Show that for $i = 1, 2, \ldots, n$, the probability that the i^{th} insertion requires more than $2 \lg n$ probes is at most $1/n^2$.

   Let the random variable X_i denote the number of probes required by the i^{th} insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} \leq 1/n^2$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

   (c) Show that $\Pr\{X > 2 \lg n\} \leq 1/n$.

   (d) Show that the expected length of the longest probe sequence is $E[X] = O(\lg n)$.

3. A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. That is operation i costs $f(i)$, where:

$$f(i) = \begin{cases} i, & i = 2^k, \\ 1, & \text{otherwise} \end{cases}$$

Determine the amortized cost per operation using the following methods of analysis:

(a) Aggregate method

(b) Accounting method

*(c) Potential method

CS 373: Combinatorial Algorithms, Fall 2002

Homework 4, due Thursday, October 31, 2002 at 23:59.99

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Undergrads	Grads

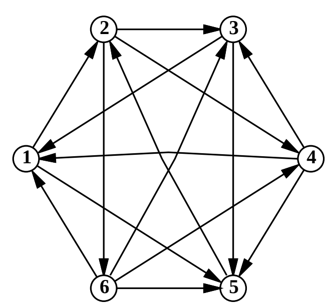
This homework is to be submitted in groups of up to three people. Graduate and undergraduate students are *not* allowed to work in the same group. Please indicate above whether you are undergraduate or graduate students. Only *one* submission per group will be accepted.

Required Problems

1. Tournament:

   A *tournament* is a directed graph with exactly one edge between every pair of vertices. (Think of the nodes as players in a round-robin tournament, where each edge points from the winner to the loser.) A *Hamiltonian path* is a sequence of directed edges, joined end to end, that visits every vertex exactly once.

   Prove that every tournament contains at least one Hamiltonian path.



A six-vertex tournament containing the Hamiltonian path $6 \to 4 \to 5 \to 2 \to 3 \to 1$.

2. Acrophobia:

   Consider a graph $G = (V, E)$ whose nodes are cities, and whose edges are roads connecting the cities. For each edge, the weight is assigned by h_e, the maximum altitude encountered when traversing the specified road. Between two cities s and t, we are interested in those paths whose maximum altitude is as low as possible. We will call a subgraph, G', of G an *acrophobic friendly subgraph*, if for any two nodes s and t the path of minimum altitude is always included in the subgraph. For simplicity, assume that the maximum altitude encountered on each road is unique.

(a) Prove that every graph of n nodes has an acrophobic friendly subgraph that has only $n-1$ edges.

(b) Construct an algorithm to find an acrophobic friendly subgraph given a graph $G = (V, E)$.

3. Refer to the lecture notes on single-source shortest paths. The GENERICSSSP algorithm described in class can be implemented using a stack for the 'bag'. Prove that the resulting algorithm, given a graph with n nodes as input, could perform $\Omega(2^n)$ relaxation steps before stopping. You need to describe, for any positive integer n, a specific weighted directed n-vertex graph that forces this exponential behavior. The easiest way to describe such a family of graphs is using an *algorithm*!

4. Neighbors:
Two spanning trees T and T' are defined as *neighbors* if T' can be obtained from T by swapping a single edge. More formally, there are two edges e and f such that T' is obtained from T by adding edge e and deleting edge f.

(a) Let T denote the minimum cost spanning tree and suppose that we want to find the second cheapest tree T' among all trees. Assuming unique costs for all edges, prove that T and T' are neighbors.

(b) Given a graph $G = (V, E)$, construct an algorithm to find the second cheapest tree, T'.

(c) Consider a graph, H, whose vertices are the spanning trees of the graph G. Two vertices are connected by an edge if and only if they are neighbors as previously defined. Prove that for any graph G this new graph H is connected.

5. Network Throughput:
Suppose you are given a graph of a (tremendously simplified) computer network $G = (V, E)$ such that a weight, b_e, is assigned to each edge representing the communication bandwidth of the specified channel in Kb/s and each node is assigned a value, l_v, representing the server latency measured in seconds/packet. Given a fixed packet size, and assuming all edge bandwidth values are a multiple of the packet size, your job is to build a system to decide which paths to route traffic between specfied servers.
More formally, a person wants to route traffic from server s to server t along the path of maximum throughput. Give an algorithm that will allow a network design engineer to choose an optimal path by which to route data traffic.

6. All-Pairs-Shortest-Path:
[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]
Given an undirected, unweighted, connected graph $G = (V, E)$, we wish to solve the distance version of the all-pairs-shortest-path problem. The algorithm APD takes the $n \times n$ 0-1 adjacency matrix A and returns an $n \times n$ matrix D such that d_{ij} represents the shortest path between vertices i and j.

```
APD(A)
    Z ← A · A
    let B be an n × n matrix, where b_ij = 1 iff i ≠ j and (a_ij = 1 or z_ij > 0)
    if b_ij = 1 for all i ≠ j
        return D ← 2B − A
    T ← APD(B)
    X ← T · A
    foreach x_ij
        if x_ij ≥ t_ij · degree(j)
            d_ij ← 2t_ij
        else
            d_ij ← 2t_ij − 1
    return D
```

(a) In the APD algorithm above, what do the matrices Z, B, T, and X represent? Justify your answers.

(b) Prove that the APD algorithm correctly computes the matrix of shortest path distances. In other words, prove that in the output matrix D, each entry d_{ij} represents the shortest path distance between node i and node j.

(c) Suppose we can multiply two $n \times n$ matrices in $M(n)$ time, where $M(n) = \Omega(n^2)$.[1] Prove that APD runs in $O(M(n) \log n)$ time.

[1] The matrix multiplication algorithm you already know runs in $O(n^3)$ time, but this is not the fastest known. The current record is $M(n) = O(n^{2.376})$, due to Don Coppersmith and Shmuel Winograd. Determinign the smallest possible value of $M(n)$ is a long-standing open problem.

Practice Problems

1. Makefiles:
   In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called 'make' that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of the other files which are listed. Thus a source file must be recompiled if a file on which it depends is changed.

   Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design an algorithm to recompile only those necessary. DO NOT worry about the details of parsing a Makefile.

2. The incidence matrix of an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that
$$b_{ij} = \begin{cases} 1 & \text{if vertex } v_i \text{ is an endpoint of edge } e_j \\ 0 & \text{otherwise} \end{cases}$$

   (a) Describe what all the entries of the matrix product BB^T represent (B^T is the matrix transpose). Justify.

   (b) Describe what all the entries of the matrix product $B^T B$ represent. Justify.

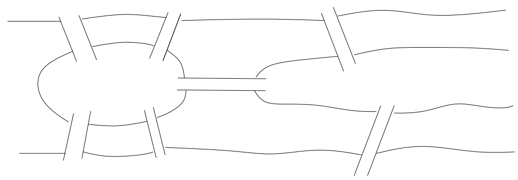
   ★(c) Let $C = BB^T - 2A$. Let C' be C with the first row and column removed. Show that $\det C'$ is the number of spanning trees. (A is the adjacency matrix of G, with zeroes on the diagonal).

3. Reliable Network:
   Suppose you are given a graph of a computer network $G = (V, E)$ and a function $r(u, v)$ that gives a reliability value for every edge $(u, v) \in E$ such that $0 \leq r(u, v) \leq 1$. The reliability value gives the probability that the network connection corresponding to that edge will *not* fail. Describe and analyze an algorithm to find the most reliable path from a given source vertex s to a given target vertex t.

4. Aerophobia:
   After graduating you find a job with Aerophobes-R′-Us, the leading traveling agency for aerophobic people. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying so the trip should be as short as possible.

   In other words, a person wants to fly from city A to city B in the shortest possible time. S/he turns to the traveling agent who knows all the departure and arrival times of all the flights on the planet. Give an algorithm that will allow the agent to choose an optimal route to minimize the total time in transit. Hint: rather than modify Dijkstra's algorithm, modify the data. The total transit time is from departure to arrival at the destination, so it will include layover time (time waiting for a connecting flight).

5. The Seven Bridges of Königsberg:
   During the eighteenth century the city of Königsberg in East Prussia was divided into four sections by the Pregel river. Seven bridges connected these regions, as shown below. It was said that residents spent their Sunday walks trying to find a way to walk about the city so as to cross each bridge exactly once and then return to their starting point.



   (a) Show how the residents of the city could accomplish such a walk or prove no such walk exists.

   (b) Given any undirected graph $G = (V, E)$, give an algorithm that finds a cycle in the graph that visits every edge exactly once, or says that it can't be done.

6. Given an undirected graph $G = (V, E)$ with costs $c_e \geq 0$ on the edges $e \in E$ give an $O(|E|)$ time algorithm that tests if there is a minimum cost spanning tree that contains the edge e.

7. Combining Boruvka and Prim:
   Give an algorithm that find the MST of a graph G in $O(m \log \log n)$ time by combining Boruvka's and Prim's algorithm.

8. Minimum Spanning Tree changes:
   Suppose you have a graph G and an MST of that graph (i.e. the MST has already been constructed).

   (a) Give an algorithm to update the MST when an edge is added to G.

   (b) Give an algorithm to update the MST when an edge is deleted from G.

   (c) Give an algorithm to update the MST when a vertex (and possibly edges to it) is added to G.

9. Nesting Envelopes
   You are given an unlimited number of each of n different types of envelopes. The dimensions of envelope type i are $x_i \times y_i$. In nesting envelopes inside one another, you can place envelope A inside envelope B if and only if the dimensions A are *strictly smaller* than the dimensions of B. Design and analyze an algorithm to determine the largest number of envelopes that can be nested inside one another.

10. $o(V^2)$ Adjacency Matrix Algorithms

   (a) Give an $O(V)$ algorithm to decide whether a directed graph contains a *sink* in an adjacency matrix representation. A sink is a vertex with in-degree $V - 1$.

(b) An undirected graph is a scorpion if it has a vertex of degree 1 (the sting) connected to a vertex of degree two (the tail) connected to a vertex of degree $V - 2$ (the body) connected to the other $V - 3$ vertices (the feet). Some of the feet may be connected to other feet.

Design an algorithm that decides whether a given adjacency matrix represents a scorpion by examining only $O(V)$ of the entries.

(c) Show that it is impossible to decide whether G has at least one edge in $O(V)$ time.

11. Shortest Cycle:
Given an **undirected** graph $G = (V, E)$, and a weight function $f : E \to \mathbf{R}$ on the **edges**, give an algorithm that finds (in time polynomial in V and E) a cycle of smallest weight in G.

12. Longest Simple Path:
Let graph $G = (V, E)$, $|V| = n$. A *simple path* of G, is a path that does not contain the same vertex twice. Use dynamic programming to design an algorithm (not polynomial time) to find a simple path of maximum length in G. Hint: It can be done in $O(n^c 2^n)$ time, for some constant c.

13. Minimum Spanning Tree:
Suppose all edge weights in a graph G are equal. Give an algorithm to compute an MST.

14. Transitive reduction:
Give an algorithm to construct a *transitive reduction* of a directed graph G, i.e. a graph G^{TR} with the fewest edges (but with the same vertices) such that there is a path from a to b in G iff there is also such a path in G^{TR}.

15. (a) What is $5^{2^{29^{5^0}+23^{4^1}+17^{3^2}+11^{2^3}+5^{1^4}}} \bmod 6$?

(b) What is the capital of Nebraska? Hint: It is not Omaha. It is named after a famous president of the United States that was not George Washington. The distance from the Earth to the Moon averages roughly 384,000 km.

CS 373: Combinatorial Algorithms, Fall 2002

http://www-courses.cs.uiuc.edu/~cs373

Homework 5 (due Thur. Nov. 21, 2002 at 11:59 pm)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. (10 points) Given two arrays, $A[1..n]$ and $B[1..m]$ we want to determine whether there is an $i \geq 0$ such that $B[1] = A[i+1]$,$B[2] = A[i+2]$,...,$B[m] = A[i+m]$. In other words, we want to determine if B is a substring of A. Show how to solve this problem in $O(n \log n)$ time with high probability.

2. (5 points) Let $a, b, c \in \mathbb{Z}^+$.

   (a) Prove that $\gcd(a,b) \cdot \operatorname{lcm}(a,b) = ab$.

   (b) Prove $\operatorname{lcm}(a,b,c) = \operatorname{lcm}(\operatorname{lcm}(a,b),c)$.

   (c) Prove $\gcd(a,b,c)\operatorname{lcm}(ab,ac,bc) = abc$.

3. (5 points) Describe an efficient algorithm to compute multiplicative inverses modulo a prime p. Does your algorithm work if the modulos is composite?

4. (10 points) Describe an efficient algorithm to compute $F_n \bmod m$, given integers n and m as input.

5. (10 points) Let n have the prime factorization $p_1^{k_1} p_2^{k_2} \cdots p_t^{k_t}$, where the primes p_i are distinct and have exponents $k_i > 0$. Prove that

$$\phi(n) = \prod_{i=1}^{t} p_i^{k_i - 1}(p_i - 1).$$

Conclude that $\phi(n)$ can be computed in polynomial time given the prime factorization of n.

6. (10 points) Suppose we want to compute the Fast Fourier Transform of an intger vector $P[0 \mathinner{..} n-1]$. We could choose an integer m larger than any coefficient $P[i]$, and then perform all arithmetic modulo m (or more formally, in the ring $\mathbb{Z}_m$). In order to make the FFT algorithm work, we need to find an integer that functions as a "primitive nth root of unity modulo m".

   For this problem, let's assume that $m = 2^{n/2} + 1$, where as usual n is a power of two.

   (a) Prove that $2^n \equiv 1 \pmod{m}$.

   (b) Prove that $\sum_{k=0}^{n-1} 2^k \equiv 0 \pmod{m}$. These two conditions imply that 2 is a primitive nth root of unity in $\mathbb{Z}_m$.

   (c) Given (a), (b), and (c), *briefly* argue that the "FFT modulo m" of P is well-defined and be computed in $O(n \log n)$ arithmetic operations.

   (d) Prove that n has a multiplicative inverse in $\mathbb{Z}_m$. *[Hint: n is a power of 2, and m is odd.]* We need this property to implement the inverse FFT modulo m.

   (e) What is the FFT of the sequence $[3, 1, 3, 3, 7, 3, 7, 3]$ modulo 17?

7. (10 points) *[This problem is required only for graduate students taking CS 373 for a full unit; anyone else can submit a solution for extra credit.]*

   (a) Prove that for any integer $n > 1$, if the n-th Fibonacci number F_n is prime then either n is prime or $n = 4$.

   (b) Prove that if a divides b, then F_a divides F_b.

   (c) Prove that $\gcd(F_a, F_b) = F_{\gcd(a,b)}$. This immediately implies parts (a) and (b), so if you solve this part, you don't have to solve the other two.

Practice Problems

1. Let $a, b, n \in Z \setminus \{0\}$. Assume $\gcd(a,b)|n$. Prove the entire set of solutions to the equation

$$n = ax + by$$

is given by:

$$\Gamma = \{x_o + \frac{tb}{\gcd(a,b)}, y_0 - \frac{ta}{\gcd(a,b)} : t \in Z\}.$$

2. Show that in the RSA cryptosystem the decryption exponent d can be chosen such that $de \equiv 1 \mod \operatorname{lcm}(p-1, q-1)$.

3. Let (n, e) be a public RSA key. For a plaintext $m \in \{0, 1, ..., n-1\}$, let $c = m^e \mod n$ be the corresponding ciphertext. Prove that there is a positive integer k such that

$$m^{e^k} \equiv m \mod n.$$

For such an integer k, prove that

$$c^{e^{k-1}} \equiv m \mod n.$$

Is this dangerous for RSA?

4. Prove that if Alice's RSA public exponent e is 3 and an adversary obtains Alice's secret exponent d, then the adversary can factor Alice's modulus n in time polynomial in the number of bits in n.

CS 373: Combinatorial Algorithms, Fall 2002

http://www-courses.cs.uiuc.edu/˜cs373

Homework 6 (Do not hand in!)

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

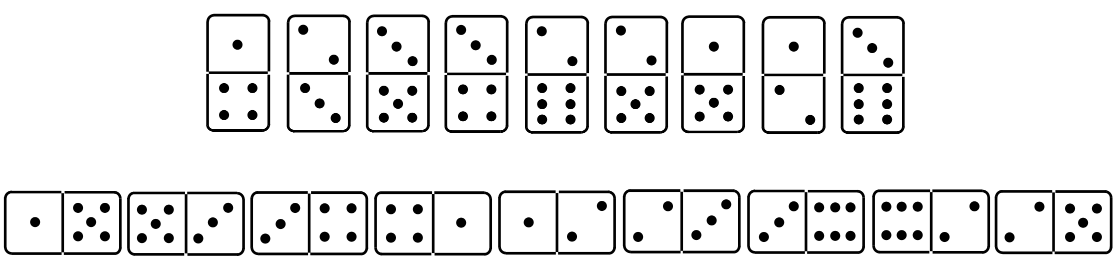
Name:		
Net ID:	Alias:	U $^3\!/_4$ 1

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Please also tell us whether you are an undergraduate, 3/4-unit grad student, or 1-unit grad student by circling U, $^3\!/_4$, or 1, respectively. Staple this sheet to the top of your homework.

Required Problems

1. (10 points) Prove that SAT is still a NP-complete problem even under the following constraints: each variable must show up once as a positive literal and once or twice as a negative literal in the whole expression. For instance, $(A \vee \overline{B}) \wedge (\overline{A} \vee C \vee D) \wedge (\overline{A} \vee B \vee \overline{C} \vee \overline{D})$ satisfies the constraints, while $(A \vee \overline{B}) \wedge (\overline{A} \vee C \vee D) \wedge (A \vee B \vee \overline{C} \vee \overline{D})$ does not, because positive literal A appears twice.

2. (10 points) A domino is 2×1 rectanble divided into two squares, with a certain number of pips(dots) in each square. In most domino games, the players lay down dominos at either end of a single chain. Adjacent dominos in the chain must have matching numbers. (See the figure below.)
Describe and analyze an efficient algorithm, or prove that it is NP-complete, to determine wheter a given set of n dominos can be lined up in a single chain. For example, for the sets of dominos shown below, the correct output is TRUE.



Top: A set of nine dominos
Bottom:The entire set lined up in a single chain

3. (10 points) Prove that the following 2 problems are NP-complete. Given an undirected Graph $G = (V, E)$, a subset of vertices $V' \subseteq V$, and a positive integer k:

   (a) determine whether there is a spanning tree T of G whose leaves are the same as V'.

   (b) determine whether there is a spanning tree T of G whose degree of vertices are all less than k.

4. (10 points) An optimized version of Knapsack problem is defined as follows. Given a finite set of elements U where each element of the set $u \in U$ has its own size $s(u) > 0$ and the value $v(u) > 0$, maximize $A(U') = \sum_{u \in U'} v(u)$ under the condition $\sum_{u \in U'} s(u) \leq B$ and $U' \subseteq U$. This problem is NP-hard. Consider the following polynomial time approximation algorithm. Determine the worst case approximation ratio $R(U) = \max_{U} Opt(U)/Approx(U)$ and prove it.

```
APPROXIMATIONALGORITHM:
    A1 ← Greedy()
    A2 ← SingleElement()
    return max(A1, A2)
```

```
GREEDY:
    Put all the elements u∈U into an array A[i]
    Sort A[i] by v(u)/s(u) in a decreasing order
    S ← 0
    V ← 0
    for i ← 0 to NumOfElements
        if (S + s(u[i]) > B)
            break
        S ← S + s(u[i])
        V ← V + v(u[i])
    return V
```

```
SINGLEELEMENT:
    Put all the elements u∈U into an array A[i]
    V ← 0
    for i ← 0 to NumOfElements
        if (s(u[i]) ≤ B & V < v(u[i]))
            V ← v(u[i])
    return V
```

5. (10 points) The recursion fairy's distant cousin, the reduction genie, shows up one day with a magical gift for you: a box that determines in constant time whether or not a graph is 3-colorable.(A graph is 3-colorable if you can color each of the vertices red, green, or blue, so that every edge has do different colors.) The magic box does not tell you how to color the graph, just wheter or not it can be done. Devise and analyze an algorithm to 3-color any graph in **polynomial time** using the magic box.

6. (10 points) The following is an NP-hard version of PARTITION problem.

   PARTITION(NP-HARD):

   Given a set of n positive integers $S = \{a_i | i = 0 \ldots n-1\}$,

   $$\text{minimize} \max\left(\sum_{a_i \in T} a_i, \sum_{a_i \in S-T} a_i\right)$$

   where T is a subset of S.

   A polynomial time approximation algorithm is given in what follows. Determine the worst case approximation ratio $\min_{S} Approx(S)/Opt(S)$ and prove it.

```
APPROXIMATIONALGORITHM:
    Sort S in an increasing order
    s1 ← 0
    s2 ← 0
    for i ← 0 to n
        if s1 ≤ s2
            s1 ← s1 + a_i
        else
            s2 ← s2 + a_i
    result ← max(s1, s2)
```

Practice Problems

1. Construct a linear time algorithm for 2 SAT problem.

2. Assume that $P \neq NP$. Prove that there is no polynomial time approximation algorithm for an optimized version of Knapsack problem, which outputs $A(I)$ s.t. $|Opt(I) - A(I)| \leq K$ for any instance I, where K is a constant.

3. Your friend Toidi is planning to hold a party for the coming Christmas. He wants to take a picture of all the participants including himself, but he is quite **shy** and thus cannot take a picture of a person whom he does not know very well. Since he has only **shy** friends, every participant coming to the party is also **shy**. After a long struggle of thought he came up with a seemingly good idea:

   - At the beginning, he has a camera.
   - A person, holding a camera, is able to take a picture of another participant whom the person knows very well, and pass a camera to that participant.
   - Since he does not want to waste films, everyone has to be taken a picture exactly once.

   Although there can be some people whom he does not know very well, he knows completely who knows whom well. Therefore, in theory, given a list of all the participants, he can determine if it is possible to take all the pictures using this idea. Since it takes only linear time to take all the pictures if he is brave enough (say "Say cheese!" N times, where N is the number of people), as a student taking CS373, you are highly expected to give him an advice:

   - show him an efficient algorithm to determine if it is possible to take pictures of all the participants using his idea, given a list of people coming to the party.
   - or prove that his idea is essentially facing a NP-complete problem, make him give up his idea, and give him an efficient algorithm to practice saying "Say cheese!":

     e.g.,
     ```
     for i ← 0 to N
         Make him say "Say cheese!" 2^i times
     ```
     oops, it takes exponential time...

4. Show, given a set of numbers, that you can decide wheter it has a subset of size 3 that adds to zero in polynomial time.

5. Given a CNF-normalized form that has at most one negative literal in each clause, construct an efficient algorithm to solve the satisfiability problem for these clauses. For instance,

$$(A \vee B \vee \overline{C}) \wedge (B \vee \overline{A}),$$
$$(A \vee \overline{B} \vee C) \wedge (B \vee \overline{A} \vee D) \wedge (A \vee D),$$
$$(\overline{A} \vee B) \wedge (B \vee \overline{A} \vee C) \wedge (C \vee D)$$

satisfy the condition, while

$$(\overline{A} \vee B \vee \overline{C}) \wedge (B \vee \overline{A}),$$
$$(A \vee \overline{B} \vee C) \wedge (B \vee \overline{A} \vee \overline{D}) \wedge (A \vee D),$$
$$(\overline{A} \vee B) \wedge (B \vee \overline{A} \vee C) \wedge (\overline{C} \vee \overline{D})$$

do not.

6. The ExactCoverByThrees problem is defined as follows: given a finite set X and a collection C of 3-element subsets of X, does C contain an exact cover for X, that is, a sub-collection $C' \subseteq C$ where every element of X occurs in exactly one member of C'? Given that ExactCoverByThrees is NP-complete, show that the similar problem ExactCoverByFours is also NP-complete.

7. The *LongestSimpleCycle* problem is the problem of finding a simple cycle of maximum length in a graph. Convert this to a formal definition of a decision problem and show that it is NP-complete.

Write your answers in the separate answer booklet.

1. **Multiple Choice:** Each question below has one of the following answers.

   A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$ X: I don't know.

   For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point. Each X earns you $\frac{1}{4}$ point. **Each incorrect answer *costs* you $\frac{1}{2}$ point.** Your total score will be rounded **down** to an integer. Negative scores will be rounded up to zero.

   (a) What is $\sum_{i=1}^{n} \frac{i}{n}$?

   (b) What is $\sum_{i=1}^{n} \frac{n}{i}$?

   (c) How many bits do you need to write 10^n in binary?

   (d) What is the solution of the recurrence $T(n) = 9T(n/3) + n$?

   (e) What is the solution of the recurrence $T(n) = T(n-2) + \frac{3}{n}$?

   (f) What is the solution of the recurrence $T(n) = 5T\left(\left\lceil \frac{n-17}{25} \right\rceil - \lg \lg n\right) + \pi n + 2^{\sqrt{\log^* n}} - 6$?

   (g) What is the worst-case running time of randomized quicksort?

   (h) The expected time for inserting one item into a randomized treap is $O(\log n)$. What is the worst-case time for a sequence of n insertions into an initially empty treap?

   (i) Suppose StupidAlgorithm produces the correct answer to some problem with probability $1/n$. How many times do we have to run StupidAlgorithm to get the correct answer with high probability?

   (j) Suppose you correctly identify three of the possible answers to this question as obviously wrong. If you choose one of the three remaining answers at random, each with equal probability, what is your expected score for this question?

2. Consider the following algorithm for finding the smallest element in an unsorted array:

```
RandomMin(A[1..n]):
    min ← ∞
    for i ← 1 to n in random order
        if A[i] < min
            min ← A[i]    (⋆)
    return min
```

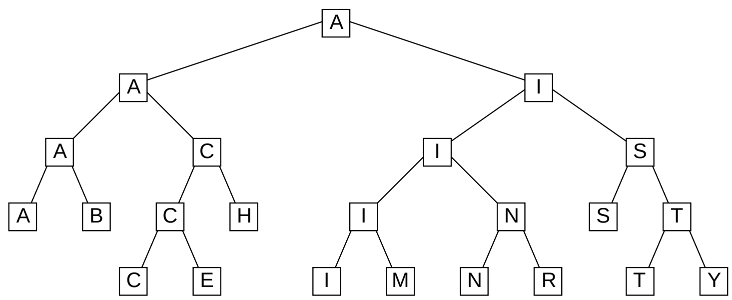
   (a) **[1 point]** In the worst case, how many times does RandomMin execute line $(\star)$?

   (b) **[3 points]** What is the probability that line $(\star)$ is executed during the nth iteration of the for loop?

   (c) **[6 points]** What is the exact expected number of executions of line $(\star)$? (A correct $\Theta()$ bound is worth 4 points.)

3. Algorithms and data structures were developed millions of years ago by the Martians, but not quite in the same way as the recent development here on Earth. Intelligent life evolved independently on Mars' two moons, Phobos and Deimos.[1] When the two races finally met on the surface of Mars, after thousands of Phobos-orbits[2] of separate philosophical, cultural, religious, and scientific development, their disagreements over the proper structure of binary search trees led to a bloody (or more accurately, ichorous) war, ultimately leading to the destruction of all Martian life.

   A *Phobian* binary search tree is a full binary tree that stores a set X of search keys. The root of the tree stores the *smallest* element in X. If X has more than one element, then the left subtree stores all the elements less than some pivot value p, and the right subtree stores everything else. Both subtrees are *nonempty* Phobian binary search trees. The actual pivot value p is *never* stored in the tree.



A Phobian binary search tree for the set $\{M, A, R, T, I, N, B, Y, S, C, H, E\}$.

   (a) **[2 points]** Describe and analyze an algorithm $\text{Find}(x, T)$ that returns True if x is stored in the Phobian binary search tree T, and False otherwise.

   (b) **[2 points]** Show how to perform a rotation in a Phobian binary search tree in $O(1)$ time.

   (c) **[6 points]** A *Deimoid* binary search tree is almost exactly the same as its Phobian counterpart, except that the *largest* element is stored at the root, and both subtrees are Deimoid binary search trees. Describe and analyze an algorithm to transform an n-node Phobian binary search tree into a Deimoid binary search tree in $O(n)$ time, using as little additional space as possible.

4. Suppose we are given an array $A[1\,..\,n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are five local minima in the following array:

9	7	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9

   We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array **must** have at least one local minimum. Why?]*

[1] Greek for "fear" and "panic", respectively. Doesn't that make you feel better?
[2] 1000 Phobos orbits $\approx$ 1 Earth year

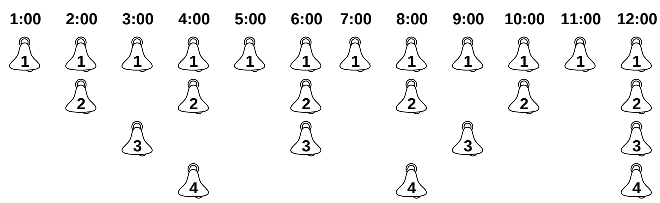
5. *[Graduate students must answer this question.]*

   A *common supersequence* of two strings A and B is a string of minimum total length that includes both the characters of A in order and the characters of B in order. Design and analyze and algorithm to compute the length of the *shortest* common supersequence of two strings $A[1..m]$ and $B[1..n]$. For example, if the input strings are `ANTHROHOPOBIOLOGICAL` and `PRETERDIPLOMATICALLY`, your algorithm should output 31, since a shortest common supersequence of those two strings is `PREANTHEROHODPOBIOPLOMATGICALLY`. You do not need to compute an actual supersequence, just its length. For full credit, your algorithm must run in $\Theta(nm)$ time.

> **Write your answers in the separate answer booklet.**
> **This is a 90-minute exam. The clock started when you got the questions.**

1. Professor Quasimodo has built a device that automatically rings the bells in the tower of the Cathédrale de Notre Dame de Paris so he can finally visit his true love Esmerelda. Every hour exactly on the hour (when the minute hand is pointing at the 12), the device rings at least one of the n bells in the tower. Specifically, the ith bell is rung once every i hours.

   For example, suppose $n = 4$. If Quasimodo starts his device just after midnight, then his device rings the bells according to the following twelve-hour schedule:

   What is the *amortized* number of bells rung per hour, as a function of n? For full credit, give an exact closed-form solution; a correct $\Theta()$ bound is worth 5 points.

2. Let G be a directed graph, where every edge $u \to v$ has a weight $w(u \to v)$. To compute the shortest paths from a start vertex s to every other node in the graph, the generic single-source shortest path algorithm calls InitSSSP once and then repeatedly calls Relax until there are no more tense edges.

   InitSSSP(s):
   - $\text{dist}(s) \leftarrow 0$
   - $\text{pred}(s) \leftarrow$ Null
   - for all vertices $v \neq s$
     - $\text{dist}(v) \leftarrow \infty$
     - $\text{pred}(v) \leftarrow$ Null

   Relax($u \to v$):
   - if $\text{dist}(v) > \text{dist}(u) + w(u \to v)$
     - $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \to v)$
     - $\text{pred}(v) \leftarrow u$

   Suppose the input graph has no negative cycles. Let v be an arbitrary vertex in the input graph. **Prove** that after every call to Relax, if $\text{dist}(v) \neq \infty$, then $\text{dist}(v)$ is the total weight of some path from s to v.

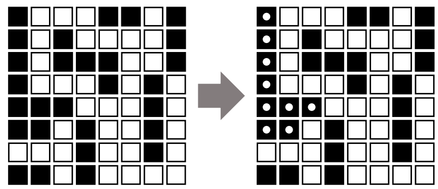
3. Suppose we want to maintain a dynamic set of values, subject to the following operations:

   - Insert(x): Add x to the set (if it isn't already there).
   - Print&DeleteRange(a, b): Print and delete every element x in the range $a \leq x \leq b$. For example, if the current set is $\{1, 5, 3, 4, 8\}$, then Print&DeleteRange($4, 6$) prints the numbers 4 and 5 and changes the set to $\{1, 3, 8\}$.

   Describe and analyze a data structure that supports these operations, each with amortized cost $O(\log n)$.

4. (a) **[4 pts]** Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$.

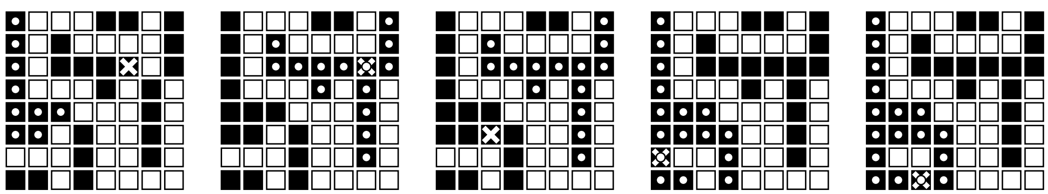
   For example, given the bitmap below as input, your algorithm should return the number 9, because the largest conected black component (marked with white dots on the right) contains nine pixels.

   9

   (b) **[4 pts]** Design and analyze an algorithm BLACKEN(i, j) that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

   For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the BLACKEN algorithm.

   9 14 14 16 17

   (c) **[2 pts]** What is the *worst-case* running time of your BLACKEN algorithm?

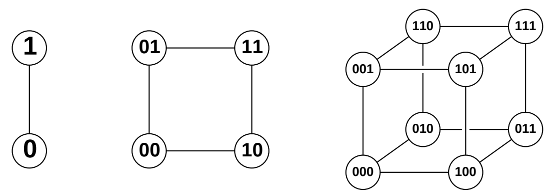
5. *[Graduate students must answer this question.]*

   After a grueling 373 midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between bus lines at least once.

   Describe and analyze an algorithm to determine the sequence of bus rides that will get you home as early as possible, assuming there are b different bus lines, and each bus stops n times per day. Your goal is to minimize your *arrival time*, not the time you actually spend travelling. Assume that the buses run exactly on schedule, that you have an accurate watch, and that you are too tired to walk between bus stops.

Write your answers in the separate answer booklet.
This is a 180-minute exam. The clock started when you got the questions.

1. The *d-dimensional hypercube* is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if their labels differ in exactly one bit.

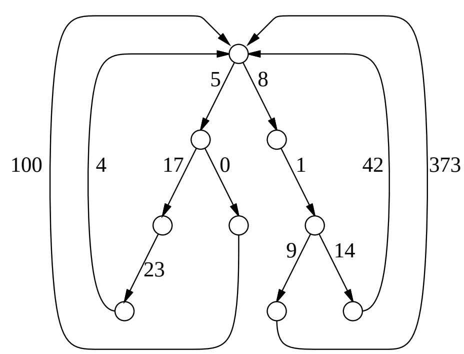


The 1-dimensional, 2-dimensional, and 3-dimensional hypercubes.

   (a) **[8 pts]** Recall that a Hamiltonian cycle passes through every vertex in a graph exactly once. ***Prove*** that for all $d \geq 2$, the d-dimensional hypercube has a Hamiltonian cycle.

   (b) **[2 pts]** Which hypercubes have an Eulerian circuit (a closed walk that visits every edge exactly once)? *[Hint: This is very easy.]*

2. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight. The number of nodes in the graph is n.



   (a) How long would it take Dijkstra's algorithm to compute the shortest path between two vertices u and v in a looped tree?

   (b) Describe and analyze a faster algorithm.

3. Prove that $(x + y)^p \equiv x^p + y^p \pmod{p}$ for any prime number p.

4. A *palindrome* is a string that reads the same forwards and backwards, like X, 373, noon, redivider, or amanaplanacatahamayakayamahatacanalpanama. *Any* string can be written as a sequence of palindromes. For example, the string bubbaseesabanana ('Bubba sees a banana.') can be decomposed in several ways; for example:

   bub + baseesab + anana
   b + u + bb + a + sees + aba + nan + a
   b + u + bb + a + sees + a + b + anana
   b + u + b + b + a + s + e + e + s + a + b + a + n + a + n + a

   Describe an efficient algorithm to find the *minimum* number of palindromes that make up a given input string. For example, given the input string bubbaseesabanana, your algorithm would return the number 3.

5. Your boss wants you to find a *perfect* hash function for mapping a known set of n items into a table of size m. A hash function is perfect if there are *no* collisions; each of the n items is mapped to a different slot in the hash table. Of course, this requires that $m \geq n$.

   After cursing your 373 instructor for not teaching you about perfect hashing, you decide to try something simple: repeatedly pick *random* hash functions until you find one that happens to be perfect. A random hash function h satisfies two properties:

   - $\Pr\big[h(x) = h(y)\big] = \frac{1}{m}$ for any pair of items $x \neq y$.
   - $\Pr\big[h(x) = i\big] = \frac{1}{m}$ for any item x and any integer $1 \leq i \leq m$.

   (a) **[2 pts]** Suppose you pick a random hash function h. What is the *exact* expected number of collisions, as a function of n (the number of items) and m (the size of the table)? Don't worry about how to *resolve* collisions; just count them.

   (b) **[2 pts]** What is the *exact* probability that a random hash function is perfect?

   (c) **[2 pts]** What is the *exact* expected number of different random hash functions you have to test before you find a perfect hash function?

   (d) **[2 pts]** What is the *exact* probability that none of the first N random hash functions you try is perfect?

   (e) **[2 pts]** How many random hash functions do you have to test to find a perfect hash function *with high probability*?

   To get full credit for parts (a)–(d), give *exact* closed-form solutions; correct $\Theta(\cdot)$ bounds are worth significant partial credit. Part (e) requires only a $\Theta(\cdot)$ bound; an exact answer is worth extra credit.

6. Your friend Toidi is planning to hold a Christmas party. He wants to take a picture of all the participants, including himself, but he is quite shy and thus cannot take a picture of a person whom he does not know very well. Since he has only shy friends[1], everyone at the party is also shy. After thinking hard for a long time, he came up with a seemingly good idea:

   - Toidi brings a disposable camera to the party.
   - Anyone holding the camera can take a picture of someone they know very well, and then pass the camera to that person.
   - In order not to waste any film, every person must have their picture taken exactly once.

   Although there can be some people Toidi does not know very well, he knows completely who knows whom well. Thus, *in principle*, given a list of all the participants, he can determine whether it is possible to take all the pictures using this idea. But how quickly?

   Either describe an efficient algorithm to solve Toidi's problem, or show that the problem is NP-complete.

7. The recursion fairy's cousin, the reduction genie, shows up one day with a magical gift for you: a box that can solve the NP-complete PARTITION problem in constant time! Given a set of positive integers as input, the magic box can tell you in constant time it can be split into two subsets whose total weights are equal.

   For example, given the set $\{1, 4, 5, 7, 9\}$ as input, the magic box cheerily yells "YES!", because that set can be split into $\{1, 5, 7\}$ and $\{4, 9\}$, which both add up to 13. Given the set $\{1, 4, 5, 7, 8\}$, however, the magic box mutters a sad "Sorry, no."

   The magic box does not tell you *how* to partition the set, only whether or not it can be done. Describe an algorithm to actually split a set of numbers into two subsets whose sums are equal, **in polynomial time**, using this magic box.[2]

[1] Except you, of course. Unfortunately, you can't go to the party because you're taking a final exam. Sorry!
[2] Your solution to problem 4 in homework 1 does *not* solve this problem in polynomial time.

CS 373U: Combinatorial Algorithms, Spring 2004
Homework 0

Due January 28, 2004 at noon

Name:	
Net ID:	Alias:

☐ I understand the Homework Instructions and FAQ.

- Neatly print your full name, your NetID, and an alias of your choice in the boxes above. Grades will be listed on the course web site by alias; for privacy reasons, your alias should not resemble your name or NetID. By providing an alias, you agree to let us list your grades; if you do not provide an alias, your grades will not be listed. ***Never*** *give us your Social Security number!*
- Before you do anything else, read the Homework Instructions and FAQ on the course web page, and then check the box above. This web page gives instructions on how to write and submit homeworks—staple your solutions together in order, start each numbered problem on a new sheet of paper, write your name and NetID one every page, don't turn in source code, analyze and prove everything, use good English and good logic, and so on. See especially the policies regarding the magic phrases "I don't know" and "and so on". If you have *any* questions, post them to the course newsgroup or ask in lecture.
- This homework tests your familiarity with prerequisite material—basic data structures, big-Oh notation, recurrences, discrete probability, and most importantly, induction—to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Chapters 1–10 of CLRS should be sufficient review, but you may also want consult your discrete mathematics and data structures textbooks.
- Every homework will have five required problems and one extra-credit problem. Each numbered problem is worth 10 points.

#	1	2	3	4	5	6*	Total
Score							
Grader							

1. Sort the functions in each box from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway, just for practice. Don't merge the lists together.

   To simplify your answers, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

   (a)

$2^{\sqrt{\lg n}}$	$2^{\lg \sqrt{n}}$	$\sqrt{2^{\lg n}}$	$\sqrt{2}^{\lg n}$	$\lg 2^{\sqrt{n}}$	$\lg \sqrt{2}^n$	$\lg \sqrt{2^n}$	$\sqrt{\lg 2^n}$
$\lg n^{\sqrt{2}}$	$\lg \sqrt{n}^2$	$\lg \sqrt{n^2}$	$\sqrt{\lg n^2}$	$\lg^2 \sqrt{n}$	$\lg^{\sqrt{2}} n$	$\sqrt{\lg^2 n}$	$\sqrt{\lg n}^2$

   *(b)

$\lg(\sqrt{n}!)$	$\lg(\sqrt{n!})$	$\sqrt{\lg(n!)}$	$(\lg \sqrt{n})!$	$(\sqrt{\lg n})!$	$\sqrt{(\lg n)!}$

   [Hint: Use Stirling's approximation for factorials: $n! \approx n^{n+1/2}/e^n$]

2. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Proofs are *not* required; just give us the list of answers. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway, just for practice. Assume reasonable but nontrivial base cases. If your solution requires specific base cases, state them! Extra credit will be awarded for more exact solutions.

   (a) $A(n) = 9A(n/3) + n^2$

   (b) $B(n) = 2B(n/2) + n/\lg n$

   (c) $C(n) = \dfrac{2C(n-1)}{C(n-2)}$ *[Hint: This is easy!]*

   (d) $D(n) = D(n-1) + 1/n$

   (e) $E(n) = E(n/2) + D(n)$

   (f) $F(n) = 2F\big(\lfloor (n+3)/4 \rfloor - \sqrt{5n \lg n} + 6\big) + 7\sqrt{n+8} - \lg^9 \lg\lg n + 10^{\lg^* n} - 11/n^{12}$

   (g) $G(n) = 3G(n-1) - 3G(n-2) + G(n-3)$

   *(h) $H(n) = 4H(n/2) - 4H(n/4) + 1$ *[Hint: Careful!]*

   (i) $I(n) = I(n/3) + I(n/4) + I(n/6) + I(n/8) + I(n/12) + I(n/24) + n$

   $^\bigstar$(j) $J(n) = \sqrt{n} \cdot J(2\sqrt{n}) + n$
   [Hint: First solve the secondary recurrence $j(n) = 1 + j(2\sqrt{n})$.]

3. Scientists have recently discovered a planet, tentatively named "Ygdrasil", which is inhabited by a bizarre species called "nertices" (singular "nertex"). All nertices trace their ancestry back to a particular nertex named Rudy. Rudy is still quite alive, as is every one of his many descendants. Nertices reproduce asexually, like bees; each nertex has exactly one parent (except Rudy). There are three different types of nertices—red, green, and blue. The color of each nertex is correlated exactly with the number and color of its children, as follows:

   - Each red nertex has two children, exactly one of which is green.
   - Each green nertex has exactly one child, which is not green.
   - Blue nertices have no children.

   In each of the following problems, let R, G, and B respectively denote the number of red, green, and blue nertices on Ygdrasil.

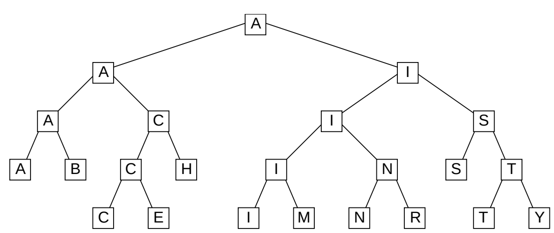
   (a) Prove that $B = R + 1$.

   (b) Prove that either $G = R$ or $G = B$.

   (c) Prove that $G = B$ if and only if Rudy is green.

4. Algorithms and data structures were developed millions of years ago by the Martians, but not quite in the same way as the recent development here on Earth. Intelligent life evolved independently on Mars' two moons, Phobos and Deimos.[1] When the two races finally met on the surface of Mars, after thousands of years of separate philosophical, cultural, religious, and scientific development, their disagreements over the proper structure of binary search trees led to a bloody (or more accurately, ichorous) war, ultimately leading to the destruction of all Martian life.

   A *Phobian* binary search tree is a full binary tree that stores a set X of search keys. The root of the tree stores the *smallest* element in X. If X has more than one element, then the left subtree stores all the elements less than some pivot value p, and the right subtree stores everything else. Both subtrees are *nonempty* Phobian binary search trees. The actual pivot value p is *never* stored in the tree.



A Phobian binary search tree for the set $\{M, A, R, T, I, N, B, Y, S, E, C, H\}$.

   (a) Describe and analyze an algorithm $\textsc{Find}(x, T)$ that returns $\textsc{True}$ if x is stored in the Phobian binary search tree T, and $\textsc{False}$ otherwise.

   (b) A *Deimoid* binary search tree is almost exactly the same as its Phobian counterpart, except that the *largest* element is stored at the root, and both subtrees are Deimoid binary search trees. Describe and analyze an algorithm to transform an n-node Phobian binary search tree into a Deimoid binary search tree in $O(n)$ time, using as little additional space as possible.

[1] Greek for "fear" and "panic", respectively. Doesn't that make you feel better?

5. Penn and Teller agree to play the following game. Penn shuffles a standard deck[2] of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3$\clubsuit$), at which point the remaining undrawn cards instantly burst into flames.

   The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.[3] To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

   (a) What is the expected number of cards that Teller draws?

   (b) What is the expected *maximum* value among the cards Teller gives to Penn?

   (c) What is the expected *minimum* value among the cards Teller gives to Penn?

   (d) What is the expected number of cards that Teller gives to Penn?

   Full credit will be given only for *exact* answers (with correct proofs, of course).

*6. **[Extra credit]**[4]

   Lazy binary is a variant of standard binary notation for representing natural numbers where we allow each "bit" to take on one of three values: 0, 1, or 2. Lazy binary notation is defined inductively as follows.

   - The lazy binary representation of zero is 0.
   - Given the lazy binary representation of any non-negative integer n, we can construct the lazy binary representation of $n + 1$ as follows:
     (a) increment the rightmost digit;
     (b) if any digit is equal to 2, replace the rightmost 2 with 0 and increment the digit immediately to its left.

   Here are the first several natural numbers in lazy binary notation:

   0, 1, 10, 11, 20, 101, 110, 111, 120, 201, 210, 1011, 1020, 1101, 1110, 1111, 1120, 1201, 1210, 2011, 2020, 2101, 2110, 10111, 10120, 10201, 10210, 11011, 11020, 11101, 11110, 11111, 11120, 11201, 11210, 12011, 12020, 12101, 12110, 20111, 20120, 20201, 20210, 21011, 21020, 21101, 21110, 101111, 101120, 101201, 101210, 102011, 102020, 102101, 102110, ...

   (a) Prove that in any lazy binary number, between any two 2s there is at least one 0, and between two 0s there is at least one 2.

   (b) Prove that for any natural number N, the sum of the digits of the lazy binary representation of N is exactly $\lfloor \lg(N + 1) \rfloor$.

---

[2]In a standard deck of 52 cards, each card has a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$ and a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$, and every possible suit-value pair appears in the deck exactly once. Actually, to make the game more interesting, Penn and Teller normally use razor-sharp ninja throwing cards.

[3]Specifically, he hurls them from the opposite side of the stage directly into the back of Penn's right hand.

[4]The "I don't know" rule does not apply to extra credit problems. There is no such thing as "partial extra credit".

CS 373U: Combinatorial Algorithms, Spring 2004
Homework 1

Due Monday, February 9, 2004 at noon

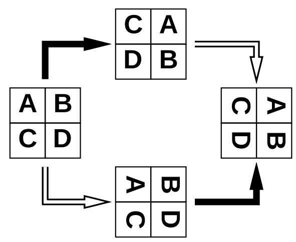
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

- For this and all following homeworks, groups of up to three people can turn in a single solution. Please write *all* your names and NetIDs on *every* page you turn in.

#	1	2	3	4	5	6^*	Total
Score							
Grader							

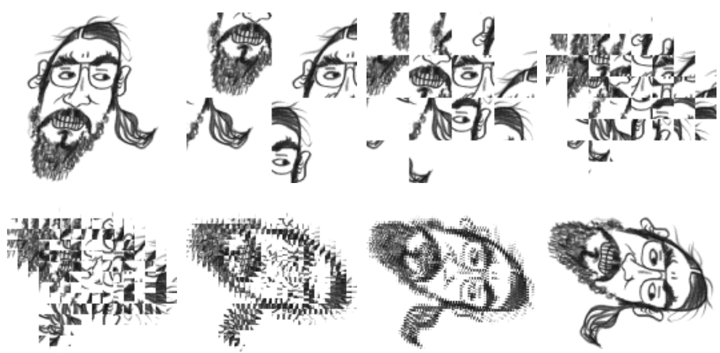
1. Some graphics hardware includes support for an operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixelmap (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

   Suppose we want to rotate an $n \times n$ pixelmap 90° clockwise. One way to do this is to split the pixelmap into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we can first recursively rotate the blocks and blit them into place afterwards.



Two algorithms for rotating a pixelmap.
Black arrows indicate blitting the blocks into place.
White arrows indicate recursively rotating the blocks.

   The following sequence of pictures shows the first algorithm (blit then recurse) in action.



   In the following questions, assume n is a power of two.

   (a) Prove that both versions of the algorithm are correct. *[Hint: If you exploit all the available symmetries, your proof will only be a half of a page long.]*

   (b) *Exactly* how many blits does the algorithm perform?

   (c) What is the algorithm's running time if each $k \times k$ blit takes $O(k^2)$ time?

   (d) What if each $k \times k$ blit takes only $O(k)$ time?

2. The traditional Devonian/Cornish drinking song "The Barley Mow" has the following pseudolyrics[1], where *container*$[i]$ is the name of a container that holds 2^i ounces of beer.[2]

```
BarleyMow(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for i ← 1 to n
      "We'll drink it out of the container[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The container[j],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

(a) Suppose each container name *container*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

(b) If you want to sing this song for $n > 20$, you'll have to make up your own container names, and to avoid repetition, these names will get progressively longer as n increases[3]. Suppose *container*$[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

(c) Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *container*$[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BarleyMow(n)? (Give an *exact* answer, not just an asymptotic bound.)

[1] Pseudolyrics are to lyrics as pseudocode is to code.

[2] One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.

[3] "We'll drink it out of the hemisemidemiyottapint, boys!"

3. In each of the problems below, you are given a 'magic box' that can solve one problem quickly, and you are asked to construct an algorithm that uses the magic box to solve a different problem.

   (a) **3-Coloring**: A graph is *3-colorable* if it is possible to color each vertex red, green, or blue, so that for every edge, its two vertices have two different colors. Suppose you have a magic box that can tell you whether a given graph is 3-colorable in constant time. Describe an algorithm that constructs a 3-coloring of a given graph (if one exists) as quickly as possible.

   (b) **3SUM:** The 3SUM problem asks, given a set of integers, whether any three elements sum to zero. Suppose you have a magic box that can solve the 3SUM problem in constant time. Describe an algorithm that actually finds, given a set of integers, three elements that sum to zero (if they exist) as quickly as possible.

   (c) **Traveling Salesman:** A *Hamiltonian cycle* in a graph is a cycle that visits every vertex exactly once. Given a complete graph where every edge has a weight, the *traveling salesman* cycle is the Hamiltonian cycle with minimum total weight; that is, the sum of the weight of the edges is smaller than for any other Hamiltonian cycle. Suppose you have a magic box that can tell you the weight of the traveling salesman cycle of a weighted graph in constant time. Describe an algorithm that actually constructs the traveling salesman cycle of a given weighted graph as quickly as possible.

4. (a) Describe and analyze an algorithm to sort an array $A[1\,..\,n]$ by calling a subroutine $\textsc{SqrtSort}(k)$, which sorts the subarray $A[k+1\,..\,k+\lceil\sqrt{n}\rceil]$ in place, given an arbitrary integer k between 0 and $n-\lceil\sqrt{n}\rceil$ as input. Your algorithm is *only* allowed to inspect or modify the input array by calling SqrtSort; in particular, your algorithm must not directly compare, move, or copy array elements. How many times does your algorithm call SqrtSort in the worst case?

   (b) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SqrtSort, prove that no algorithm can sort using $o(f(n))$ calls to SqrtSort.

   (c) Now suppose SqrtSort is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k}, so that repeated square roots are always integers.)

5. In a previous incarnation, you worked as a cashier in the lost Antarctican colony of Nadira, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource on Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadira, called Dream Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365.[4]

   (a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream Dollar bills than the minimum possible.

   (b) Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream Dollars. (Don't worry about making your algorithm fast; just make sure it's correct.)

   (c) Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream Dollars. (This one needs to be fast.)

*6. **[Extra Credit]** A popular puzzle called "Lights Out!", made by Tiger Electronics, has the following description. The game consists of a 5×5 array of lighted buttons. By pushing any button, you toggle (on to off, off to on) that light and its four (or fewer) immediate neighbors. The goal of the game is to have every light off at the same time.

   We generalize this puzzle to a graph problem. We are given an arbitrary graph with a lighted button at every vertex. Pushing the button at a vertex toggles its light and the lights at all of its neighbors in the graph. A *light configuration* is just a description of which lights are on and which are off. We say that a light configuration is *solvable* if it is possible to get from that configuration to the everything-off configuration by pushing buttons. Some (but clearly not all) light configurations are unsolvable.

   (a) Suppose the graph is just a cycle of length n. Give a simple and complete characterization of the solvable light configurations in this case. (What we're really looking for here is a *fast* algorithm to decide whether a given configuration is solvable or not.) *[Hint: For which cycle lengths is* ***every*** *configuration solvable?]*

   *(b) Characterize the set of solvable light configurations when the graph is an arbitrary tree.

   $^\bigstar$(c) A *grid graph* is a graph whose vertices are a regular $h \times w$ grid of integer points, with edges between immediate vertical or horizontal neighbors. Characterize the set of solvable light configurations for an arbitrary grid graph. (For example, the original Lights Out puzzle can be modeled as a 5×5 grid graph.)

[4] For more details on the history and culture of Nadira, including images of the various denominations of Dream Dollars, see http://www.dream-dollars.com.

CS 373U: Combinatorial Algorithms, Spring 2004
Homework 2

Due Friday, February 20, 2004 at noon
(so you have the whole weekend to study for the midterm)

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

- Starting with this homework, we are changing the way we want you to submit solutions. For each numbered problem, if you use more than one page, staple all those pages together. **Please do *not* staple your entire homework together.** This will allow us to moreeasily distribute the problems to the graders. Remember to print the name and NetID of every member of your group, as well as the assignment and problem numbers, on every page you submit. You do not need to turn in this cover page.
- Unless specifically stated otherwise, you can use the fact that the following problems are NP-hard to prove that other problems are NP-hard: Circuit-SAT, 3SAT, Vertex Cover, Maximum Clique, Maximum Independent Set, Hamiltonian Path, Hamiltonian Cycle, k-Colorability for any $k \geq 3$, Traveling Salesman Path, Travelling Salesman Cycle, Subset Sum, Partition, 3Partition, Hitting Set, Minimum Steiner Tree, Minesweeper, Tetris, or any other NP-hard problem described in the lecture notes.
- This homework is a little harder than the last one. You might want to start early.

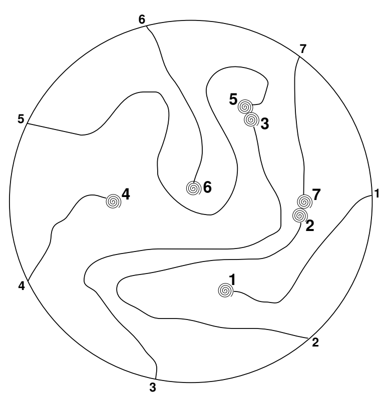
#	1	2	3	4	5	6*	Total
Score							
Grader							

1. In lecture on February 5, Jeff presented the following algorithm to compute the length of the longest increasing subsequence of an n-element array $A[1..n]$ in $O(n^2)$ time.

```
LengthOfLIS(A[1..n]):
  A[n+1] = ∞
  for i ← 1 to n+1
      L[i] ← 1
      for j ← 1 to i−1
          if A[j] < A[i] and 1 + L[j] < L[i]
              L[i] ← 1 + L[j]
  return L[n+1] − 1
```

   Describe another algorithm for this problem that runs in $O(n \log n)$ time. *[Hint: Use a data structure to replace the inner loop with something faster.]*

2. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Union Glacier hold a Round Table Mating Race. A large number of high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. The snails wander around the table, each snail leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail (even their own). When any two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if n is even and the race goes on forever.



The end of an Antarctican SLUG race. Snails 1, 4, and 6 never find a mate.
The organizers must pay $M[3,5] + M[2,7]$.

   For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i,j] = M[j,i]$ is the reward if snails i and j meet.

   Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the $n \times n$ array M as input.

3. Describe and analyze a *polynomial-time* algorithm to determine whether a boolean formula in conjunctive normal form, with exactly *two* literals in each clause, is satisfiable.

4. This problem asks you to prove that four different variants of the minimum spanning tree problem are NP-hard. In each case, the input is a connected undirected graph G with weighted edges. Each problem considers a certain subset of the possible spanning trees of G, and asks you to compute the spanning tree with minimum total weight in that subset.

   (a) Prove that finding the minimum-weight *depth first search* tree is NP-hard. (To remind yourself what depth first search is, and why it computes a spanning tree, see Jeff's introductory notes on graphs or Chapter 22 in CLRS.)

   (b) Suppose a subset S of the nodes in the input graph are marked. Prove that it is NP-hard to compute the minimum spanning tree whose leaves are all in S. *[Hint: First consider the case $|S| = 2$.]*

   (c) Prove that for any integer $\ell \geq 2$, it is NP-hard to compute the minimum spanning tree with exactly ℓ leaves. *[Hint: First consider the case $\ell = 2$.]*

   (d) Prove that for any integer $d \geq 2$, it is NP-hard to compute the minimum spanning tree with maximum degree d. *[Hint: First consider the case $d = 2$. By now this should start to look familiar.]*

   You're welcome to use reductions among these four problems. For example, even if you can't solve part (d), if you can prove that (d) implies (b), you will get full credit for (b). Just don't argue circularly.

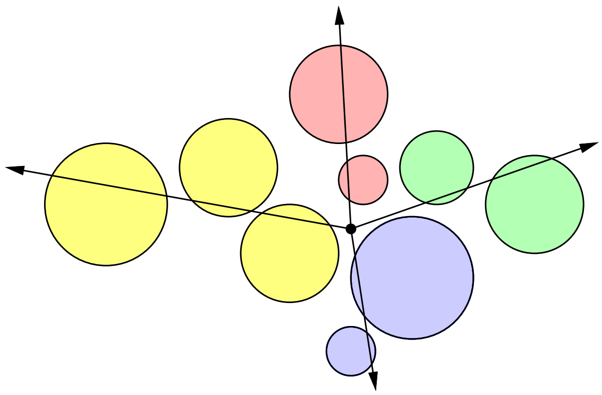
5. Consider a machine with a row of n processors numbered 1 through n. A *job* is some computational task that occupies a contiguous set of processors for some amount of time. Each processor can work on only one job at a time. Each job is represented by a pair $J_i = (n_i, t_i)$, where n_i is the number of processors required and t_i is the amount of processing time required to perform the job. A *schedule* for a set of jobs $\{J_1, \ldots, J_m\}$ assigns each job J_i to some set of n_i contiguous processors for an interval of t_i seconds, so that no processor works on more than one job at any time. The *make-span* of a schedule is the time from the start to the finish of all jobs.

   The *parallel scheduling problem* asks, given a set of jobs as input, to compute a schedule for those jobs with the smallest possible make-span.

   (a) Prove that the parallel scheduling problem is NP-hard.

   (b) Give an algorithm that computes a 3-approximation of the minimum make-span of a set of jobs in $O(m \log m)$ time. That is, if the minimum make-span is M, your algorithm should compute a schedule with make-span at most $3M$. You can assume that n is a power of 2.

*6. **[Extra credit]** Suppose you are standing in a field surrounded by several large balloons. You want to use your brand new Acme Brand Zap-O-Matic™ to pop all the balloons, without moving from your current location. The Zap-O-Matic™ shoots a high-powered laser beam, which pops all the balloons it hits. Since each shot requires enough energy to power a small country for a year, you want to fire as few shots as possible.



Nine balloons popped by 4 shots of the Zap-O-Matic™

The *minimum zap* problem can be stated more formally as follows. Given a set C of n circles in the plane, each specified by its radius and the (x, y) coordinates of its center, compute the minimum number of rays from the origin that intersect every circle in C. Your goal is to find an efficient algorithm for this problem.

(a) Describe and analyze a greedy algorithm whose output is within 1 of optimal. That is, if m is the minimum number of rays required to hit every circle in the input, then your greedy algorithm must output either m or $m + 1$. (Of course, you must prove this fact.)

(b) Describe an algorithm that solves the minimum zap problem in $O(n^2)$ time.

*(c) Describe an algorithm that solves the minimum zap problem in $O(n \log n)$ time.

Assume you have a subroutine Intersects(r, c) that determines, in $O(1)$ time, whether a ray r intersects a circle c. It's not that hard to write this subroutine, but it's not the interesting part of the problem.

CS 373U: Combinatorial Algorithms, Spring 2004
Homework 3

Due Friday, March 12, 2004 at noon

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

- For each numbered problem, if you use more than one page, staple all those pages together. **Please do *not* staple your entire homework together.** This will allow us to more easily distribute the problems to the graders. Remember to print the name and NetID of every member of your group, as well as the assignment and problem numbers, on every page you submit. You do not need to turn in this cover page.
- This homework is challenging. You might want to start early.

#	1	2	3	4	5	6*	Total
Score							
Grader							

1. Let S be a set of n points in the plane. A point p in S is called *Pareto-optimal* if no other point in S is both above and to the right of p.

   (a) Describe and analyze a deterministic algorithm that computes the Pareto-optimal points in S in $O(n \log n)$ time.

   (b) Suppose each point in S is chosen independently and uniformly at random from the unit square $[0,1] \times [0,1]$. What is the *exact* expected number of Pareto-optimal points in S?

2. Suppose we have an oracle $\text{Random}(k)$ that returns an integer chosen independently and uniformly at random from the set $\{1, \ldots, k\}$, where k is the input parameter; Random is our only source of random bits. We wish to write an efficient function $\text{RandomPermutation}(n)$ that returns a permutation of the integers $\langle 1, \ldots, n \rangle$ chosen uniformly at random.

   (a) Consider the following implementation of RandomPermutation.

```
RandomPermutation(n):
  for i = 1 to n
      π[i] ← NULL
  for i = 1 to n
      j ← Random(n)
      while (π[j] != NULL)
          j ← Random(n)
      π[j] ← i
  return π
```

   Prove that this algorithm is correct. Analyze its expected runtime.

   (b) Consider the following partial implementation of RandomPermutation.

```
RandomPermutation(n):
  for i = 1 to n
      A[i] ← Random(n)
  π ← SomeFunction(A)
  return π
```

   Prove that if the subroutine SomeFunction is deterministic, then this algorithm cannot be correct. *[Hint: There is a one-line proof.]*

   (c) Consider a correct implementation of $\text{RandomPermutation}(n)$ with the following property: whenever it calls $\text{Random}(k)$, the argument k is at most m. Prove that this algorithm ***always*** calls Random at least $\Omega(\frac{n \log n}{\log m})$ times.

   (d) Describe and analyze an implementation of RandomPermutation that runs in expected worst-case time $O(n)$.

3. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

   - MAKEQUEUE: Return a new priority queue containing the empty set.
   - FINDMIN(Q): Return the smallest element of Q (if any).
   - DELETEMIN(Q): Remove the smallest element in Q (if any).
   - INSERT(Q, x): Insert element x into Q, if it is not already there.
   - DECREASEKEY(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
   - DELETE(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
   - MELD(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

   A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

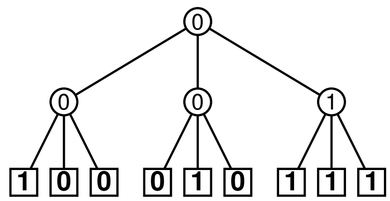
   ```
   MELD(Q1, Q2):
     if Q1 is empty return Q2
     if Q2 is empty return Q1

     if key(Q1) > key(Q2)
         swap Q1 ↔ Q2

     with probability 1/2
         left(Q1) ← MELD(left(Q1), Q2)
     else
         right(Q1) ← MELD(right(Q1), Q2)

     return Q1
   ```

   (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of MELD(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if each left/right choice is made with equal probability?]*

   (b) **[Extra credit]** Prove that MELD(Q_1, Q_2) runs in $O(\log n)$ time with high probability.

   (c) Show that each of the other meldable priority queue operations cab be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

4. A *majority tree* is a complete binary tree with depth n, where every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. Consider the problem of computing the value of the root of a majority tree, given the sequence of 3^n leaf labels as input. For example, if $n = 2$ and the leaves are labeled $1, 0, 0, 0, 1, 0, 1, 1, 1$, the root has value 0.



A majority tree with depth $n = 2$.

   (a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. *[Hint: Consider the special case $n = 1$. Recurse.]*

   (b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. *[Hint: Consider the special case $n = 1$. Recurse.]*

5. Suppose n lights labeled $0, \ldots, n-1$ are placed clockwise around a circle. Initially, each light is set to the off position. Consider the following random process.

```
LightTheCircle(n):
  k ← 0
  turn on light 0
  while at least one light is off
      with probability 1/2
          k ← (k + 1) mod n
      else
          k ← (k − 1) mod n
      if light k is off, turn it on
```

   Let $p(i, n)$ be the probability that light i is the last to be turned on by LightTheCircle$(n, 0)$. For example, $p(0, 2) = 0$ and $p(1, 2) = 1$. Find an exact closed-form expression for $p(i, n)$ in terms of n and i. Prove your answer is correct.

6. **[Extra Credit]** Let G be a *bipartite* graph on n vertices. Each vertex v has an associated set $C(v)$ of $\lg 2n$ colors with which v is compatible. We wish to find a coloring of the vertices in G so that every vertex v is assigned a color from its set $C(v)$ and no edge has the same color at both ends. Describe and analyze a randomized algorithm that computes such a coloring in expected worst-case time $O(n \log^2 n)$. *[Hint: For any events A and B, $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$.]*

CS 373U: Combinatorial Algorithms, Spring 2004
Homework 4

Due Friday, April 2, 2004 at noon

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

- For each numbered problem, if you use more than one page, staple all those pages together. **Please do *not* staple your entire homework together.** This will allow us to more easily distribute the problems to the graders. Remember to print the name and NetID of every member of your group, as well as the assignment and problem numbers, on every page you submit. You do not need to turn in this cover page.
- As with previous homeworks, we strongly encourage you to begin early.

#	1	2	3	4	5	6^*	Total
Score							
Grader							

1. Suppose we can insert or delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than 3/4 full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than 1/4 full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still a constant. Do *not* use the potential method (like CLRS does); there is a much easier solution.

2. Remember the difference between stacks and queues? Good.

   (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard subroutines Push and Pop.

   (b) A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

   - **Push:** add a new item to the left end of the list;
   - **Pop:** remove the item on the left end of the list;
   - **Pull:** remove the item on the right end of the list.

   Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any push, pop, or pull operation is $O(1)$. Again, you are *only* allowed to access the stacks through the standard functions Push and Pop.

3. Some applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Maintaining these secondary structures usually complicates algorithms for keeping the top-level search tree balanced.

   Suppose we have a binary search tree T where every node v stores a secondary structure of size $O(|v|)$, where $|v|$ denotes the number of descendants of v in T. Performing a rotation at a node v in T now requires $O(|v|)$ time, because we have to rebuild one of the secondary structures.

   (a) **[1 pt]** Overall, how much space does this data structure use in the worst case?

   (b) **[1 pt]** How much space does this structure use if the top-level search tree T is balanced?

   (c) **[2 pt]** Suppose T is a splay tree. Prove that the *amortized* cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. *[Hint: This is easy!]*

   (d) **[3 pts]** Now suppose T is a scapegoat tree, and that rebuilding the subtree rooted at v requires $\Theta(|v| \log |v|)$ time (because we also have to rebuild all the secondary structures). What is the *amortized* cost of inserting a new element into T?

   (e) **[3 pts]** Finally, suppose T is a treap. What's the worst-case *expected* time for inserting a new element into T?

4. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

   Describe an algorithm to purge an arbitrary n-node dirty binary search tree in $O(n)$ time, using only $O(\log n)$ additional memory. For 5 points extra credit, reduce the additional memory requirement to $O(1)$ *without repeating an old CS373 homework solution.*[1]

5. This problem considers a variant of the lazy binary notation introduced in the extra credit problem from Homework 0. In a *doubly lazy binary number*, each bit can take one of *four* values: -1, 0, 1, or 2. The only legal representation for zero is 0. To increment, we add 1 to the least significant bit, then carry the rightmost 2 (if any). To decrement, we subtract 1 from the lest significant bit, and then borrow the rightmost -1 (if any).

```
LazyIncrement(B[0..n]):
  B[0] ← B[0] + 1
  for i ← 1 to n − 1
      if B[i] = 2
          B[i] ← 0
          B[i + 1] ← B[i + 1] + 1
          return
```

```
LazyDecrement(B[0..n]):
  B[0] ← B[0] − 1
  for i ← 1 to n − 1
      if B[i] = −1
          B[i] ← 1
          B[i + 1] ← B[i + 1] − 1
          return
```

   For example, here is a doubly lazy binary count from zero up to twenty and then back down to zero. The bits are written with the least significant bit (*i.e.*, $B[0]$) on the right. For succinctness, we write $\cancel{1}$ instead of -1 and omit any leading 0's.

$$0 \xrightarrow{++} 1 \xrightarrow{++} 10 \xrightarrow{++} 11 \xrightarrow{++} 20 \xrightarrow{++} 101 \xrightarrow{++} 110 \xrightarrow{++} 111 \xrightarrow{++} 120 \xrightarrow{++} 201 \xrightarrow{++} 210$$
$$\xrightarrow{++} 1011 \xrightarrow{++} 1020 \xrightarrow{++} 1101 \xrightarrow{++} 1110 \xrightarrow{++} 1111 \xrightarrow{++} 1120 \xrightarrow{++} 1201 \xrightarrow{++} 1210 \xrightarrow{++} 2011 \xrightarrow{++} 2020$$
$$\xrightarrow{--} 2011 \xrightarrow{--} 2010 \xrightarrow{--} 2001 \xrightarrow{--} 2000 \xrightarrow{--} 20\cancel{1}1 \xrightarrow{--} 2\cancel{1}10 \xrightarrow{--} 2\cancel{1}01 \xrightarrow{--} 1100 \xrightarrow{--} 11\cancel{1}1 \xrightarrow{--} 1010$$
$$\xrightarrow{--} 1001 \xrightarrow{--} 1000 \xrightarrow{--} 10\cancel{1}1 \xrightarrow{--} 1\cancel{1}10 \xrightarrow{--} 1\cancel{1}01 \xrightarrow{--} 100 \xrightarrow{--} 1\cancel{1}1 \xrightarrow{--} 10 \xrightarrow{--} 1 \xrightarrow{--} 0$$

   Prove that for any intermixed sequence of increments and decrements of a doubly lazy binary number, starting with 0, the amortized time for each operation is $O(1)$. Do *not* assume, as in the example above, that all the increments come before all the decrements.

[1] That was for a slightly different problem anyway.

6. **[Extra credit]** My wife is teaching a class[2] where students work on homeworks in groups of exactly three people, subject to the following rule: *No two students may work together on more than one homework.* At the beginning of the semester, it was easy to find homework groups, but as the course progresses, it is becoming harder and harder to find a legal grouping. Finally, in despair, she decides to ask a computer scientist to write a program to find the groups for her.

   (a) We can formalize this homework-group-assignment problem as follows. The input is a graph, where the vertices are the n students, and two students are joined by an edge if they have not yet worked together. Every node in this graph has the same degree; specifically, if there have been k homeworks so far, each student is connected to exactly $n - 1 - 2k$ other students. The goal is to find $n/3$ disjoint triangles in the graph, or conclude that no such triangles exist. Prove (or disprove!) that this problem is NP-hard.

   (b) Suppose my wife had planned ahead and assigned groups for every homework at the beginning of the semester. How many homeworks can she assign, as a function of n, without violating the no-one-works-together-twice rule? Prove the best upper and lower bounds you can. To prove the upper bound, describe an algorithm that actually assigns the groups for each homework.

[2] Math 302: Non-Euclidean Geometry. Problem 1 from last week's homework assignment: "Invert Mr. Happy."

CS 373U: Combinatorial Algorithms, Spring 2004
Homework 5

Due Wednesday, April 28, 2004 at noon

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

- For each numbered problem, if you use more than one page, staple all those pages together. **Please do *not* staple your entire homework together.** This will allow us to more easily distribute the problems to the graders. Remember to print the name and NetID of every member of your group, as well as the assignment and problem numbers, on every page you submit. You do not need to turn in this cover page.
- As with previous homeworks, we strongly encourage you to begin early.
- This will be the last graded homework.

#	1	2	3	4	5	Total
Score						
Grader						

1. (a) Prove that every graph with the same number of vertices and edges has a cycle.

   (b) Prove that every graph with exactly two fewer edges than vertices is disconnected.

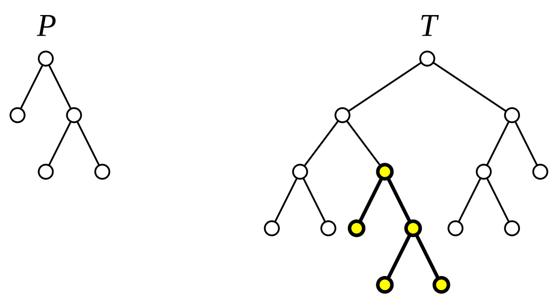
   Both proofs should be entirely self-contained. In particular, they should not use the word "tree" or any properties of trees that you saw in CS 225 or CS 273.

2. A *palindrome* is a string of characters that is exactly the same as its reversal, like `X`, `FOOF`, `RADAR`, or `AMANAPLANACATACANALPANAMA`.

   (a) Describe and analyze an algorithm to compute the longest *prefix* of a given string that is a palindrome. For example, the longest palindrome prefix of `RADARDETECTAR` is `RADAR`, and the longest palindrome prefix of `ALGORITHMSHOMEWORK` is the single letter `A`.

   (b) Describe and analyze an algorithm to compute a longest *subsequence* of a given string that is a palindrome. For example, the longest palindrome subsequnce of `RADARDETECTAR` is `RAETEAR` (or `RADADAR` or `RADRDAR` or `RATETAR` or `RATCTAR`), and the longest palindrome subsequence of `ALGORITHMSHOMEWORK` is `OMOMO` (or `RMHMR` or `OHSHO` or...).

3. Describe and analyze an algorithm that decides, given two binary trees P and T, whether T is a subtree of P. There is no actual *data* stored in the nodes—these are not binary *search* trees or binary *heaps*. You are only trying to match the *shape* of the trees.



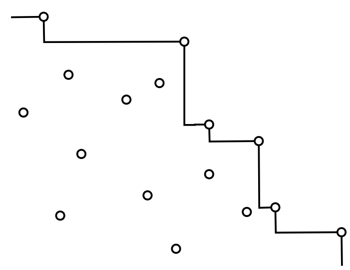
P appears exactly once as a subtree of T.

4. Describe and analyze an algorithm that computes the *second* smallest spanning tree of a given connected, undirected, edge-weighted graph.

5. Show that if the input graph is allowed to have negative edges (but no negative cycles), Dijkstra's algorithm[1] runs in exponential time in the worst case. Specifically, describe how to construct, for every integer n, a weighted directed graph G_n without negative cycles that forces Dijkstra's algorithm to perform $\Omega(2^n)$ relaxation steps. Give your description in the form of an algorithm! *[Hint: Towers of Hanoi.]*

[1] This refers to the version of Dijkstra's algorithm described in Jeff's lecture notes. The version in CLRS is always fast, but sometimes gives incorrect results for graphs with negative edges.

1. Let P be a set of n points in the plane. Recall that a point $p \in P$ is *Pareto-optimal* if no other point is both above and to the right of p. Intuitively, the sorted sequence of Pareto-optimal points describes a *staircase* with all the points in P below and to the left. Your task is to describe some algorithms that compute this staircase.



The staircase of a set of points

   (a) Describe an algorithm to compute the staircase of P in $O(nh)$ time, where h is the number of Pareto-optimal points.

   (b) Describe a divide-and-conquer algorithm to compute the staircase of P in $O(n \log n)$ time. *[Hint: I know of at least two different ways to do this.]*

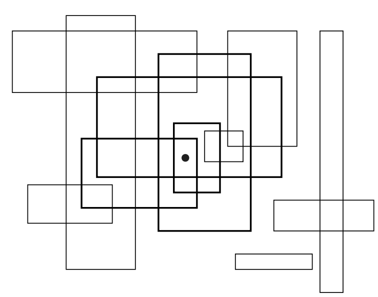
   *(c) Describe an algorithm to compute the staircase of P in $O(n \log h)$ time, where h is the number of Pareto-optimal points. *[Hint: I know of at least two different ways to do this.]*

   (d) Finally, suppose the points in P are already given in sorted order from left to right. Describe an algorithm to compute the staircase of P in $O(n)$ time. *[Hint: I know of at least two different ways to do this.]*

2. Let R be a set of n rectangles in the plane.

   (a) Describe and analyze a plane sweep algorithm to decide, in $O(n \log n)$ time, whether any two rectangles in R intersect.

   *(b) The *depth* of a point is the number of rectangles in R that contain that point. The *maximum depth* of R is the maximum, over all points p in the plane, of the depth of p. Describe a plane sweep algorithm to compute the maximum depth of R in $O(n \log n)$ time.

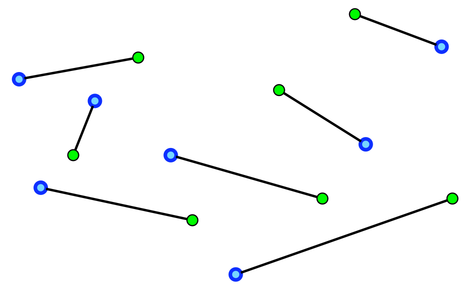


A point with depth 4 in a set of rectangles.

   (c) Describe and analyze a polynomial-time reduction from the maximum depth problem in part (b) to MaxClique: Given a graph G, how large is the largest clique in G?

   (d) MaxClique is NP-hard. So does your reduction imply that P=NP? Why or why not?

3. Let G be a set of n green points, called "Ghosts", and let B be a set of n blue points, called "ghostBusters", so that no three points lie on a common line. Each Ghostbuster has a gun that shoots a stream of particles in a straight line until it hits a ghost. The Ghostbusters want to kill all of the ghosts at once, by having each Ghostbuster shoot a different ghost. It is **very important** that the streams do not cross.



A non-crossing matching between 7 ghosts and 7 Ghostbusters

(a) Prove that the Ghostbusters can succeed. More formally, prove that there is a collection of n non-intersecting line segments, each joining one point in G to one point in B. *[Hint: Think about the set of joining segments with minimum total length.]*

(b) Describe and analyze an algorithm to find a line ℓ that passes through one ghost and one Ghostbuster, so that same number of ghosts as Ghostbusters are above ℓ.

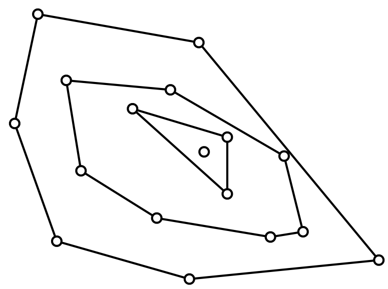
*(c) Describe and analyze an algorithm to find a line ℓ such that exactly half the ghosts and exactly half the Ghostbusters are above ℓ. (Assume n is even.)

(d) Using your algorithm for part (b) or part (c) as a subroutine, describe and analyze an algorithm to find the line segments described in part (a). (Assume n is a power of two if necessary.)

Spengler: *Don't cross the streams.*
Venkman: *Why?*
Spengler: *It would be bad.*
Venkman: *I'm fuzzy on the whole good/bad thing. What do you mean "bad"?*
Spengler: *Try to imagine all life as you know it stopping instantaneously and every molecule in your body exploding at the speed of light.*
Stantz: *Total protonic reversal!*
Venkman: *That's bad. Okay. Alright, important safety tip, thanks Egon.*

— Dr. Egon Spengler (Harold Ramis), Dr. Peter Venkman (Bill Murray), and Dr. Raymond Stanz (Dan Aykroyd), *Ghostbusters*, 1984

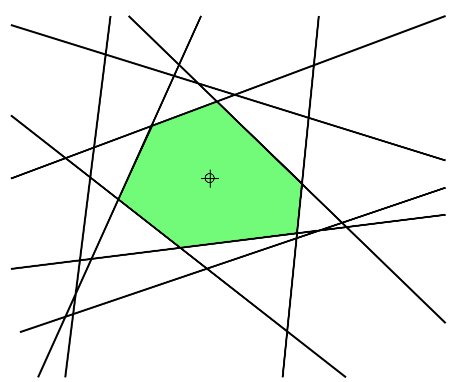
4. The *convex layers* of a point set P consist of a series of nested convex polygons. The convex layers of the empty set are empty. Otherwise, the first layer is just the convex hull of P, and the remaining layers are the convex layers of the points that are not on the convex hull of P.



The convex layers of a set of points.

Describe and analyze an efficient algorithm to compute the convex layers of a given n-point set. For full credit, your algorithm should run in $O(n^2)$ time.

5. Suppose we are given a set of n lines in the plane, where none of the lines passes through the origin $(0,0)$ and at most two lines intersect at any point. These lines divide the plane into several convex polygonal regions, or *cells*. Describe and analyze an efficient algorithm to compute the cell containing the origin. The output should be a doubly-linked list of the cell's vertices. *[Hint: There are literally dozens of solutions. One solution is to reduce this problem to the convex hull problem. Every other solution looks like a convex hull algorithm.]*



The cell containing the origin in an arrangement of lines.

Write your answers in the separate answer booklet.

1. **Multiple Choice:** Each question below has one of the following answers.

   A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$ X: I don't know.

   For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point. Each X earns you $\frac{1}{4}$ point. **Each incorrect answer *costs* you $\frac{1}{2}$ point.** Your total score will be rounded **down** to an integer. Negative scores will be rounded up to zero.

   (a) What is $\sum_{i=1}^{n} \lg i$?

   (b) What is $\sum_{i=1}^{\lg n} i2^i$?

   (c) How many decimal digits are required write the nth Fibonacci number?

   (d) What is the solution of the recurrence $T(n) = 4T(n/8) + n \log n$?

   (e) What is the solution of the recurrence $T(n) = T(n-3) + \frac{5}{n}$?

   (f) What is the solution of the recurrence $T(n) = 5T\left(\left\lceil \frac{n+13}{3} \right\rceil + \lfloor \sqrt{n} \rfloor\right) + (10n-7)^2 - \frac{\lg^3 n}{\lg \lg n}$?

   (g) How long does it take to construct a Huffman code, given an array of n character frequencies as input?

   (h) How long does it take to sort an array of size n using quicksort?

   (i) Given an unsorted array $A[1 \,..\, n]$, how long does it take to construct a binary search tree for the elements of A?

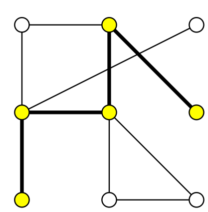
   (j) A train leaves Chicago at 8:00pm and travels south at 75 miles per hour. Another train leaves New Orleans at 1:00pm and travels north at 60 miles per hour. The conductors of both trains are playing a game of chess over the phone. After each player moves, the other player must move before his train has traveled five miles. How many moves do the two players make before their trains pass each other (somewhere near Memphis)?

2. Describe and analyze efficient algorithms to solve the following problems:

   (a) Given a set of n integers, does it contain a pair of elements a, b such that $a + b = 0$?

   (b) Given a set of n integers, does it contain three elements a, b, c such that $a + b = c$?

3. A *tonian path* in a graph G is a simple path in G that visits more than half of the vertices of G. (Intuitively, a tonian path is "most of a Hamiltonian path".) Prove that it is NP-hard to determine whether or not a given graph contains a tonian path.



A tonian path in a 9-vertex graph.

4. *Vankin's Mile* is a solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

   For example, given the grid below, the player can score $8 - 6 + 7 - 3 + 4 = 10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is *not* the best possible score for these values.)

-1	7	-8	10	-5
-4	-9	**8**	-6	0
5	-2	**-6**	-6	7
-7	4	**7**	**-3**	-3
7	1	-6	**4**	-9

   Describe and analyze an algorithm to compute the maximum possible score for a game of Vankin's Mile, given the $n \times n$ array of values as input.

5. Suppose you are given two sorted arrays $A[1 \,..\, m]$ and $B[1 \,..\, n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B in $\Theta(\log(m + n))$ time. For example, given the input

   $$A[1 \,..\, 8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1 \,..\, 5] = [2, 5, 8, 17, 19] \qquad k = 6$$

   your algorithm should return 8. You can assume that the arrays contain no duplicates. *[Hint: What can you learn from comparing one element of A to one element of B?]*

Write your answers in the separate answer booklet.

1. **Multiple Choice:** Each question below has one of the following answers.

   A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$ X: I don't know.

   For each question, write the letter that corresponds to your answer. You do not need to justify your answers. Each correct answer earns you 1 point. Each X earns you $\frac{1}{4}$ point. **Each incorrect answer *costs* you $\frac{1}{2}$ point.** Your total score will be rounded **down** to an integer. Negative scores will be rounded up to zero.

   (a) What is $\sum_{i=1}^{n} \frac{n}{i}$?

   (b) What is $\sum_{i=1}^{\lg n} 4^i$?

   (c) How many bits are required to write $n!$ in binary?

   (d) What is the solution of the recurrence $T(n) = 4T(n/2) + n \log n$?

   (e) What is the solution of the recurrence $T(n) = T(n-3) + \frac{5}{n}$?

   (f) What is the solution of the recurrence $T(n) = 9T\left(\left\lceil \frac{n+13}{3} \right\rceil\right) + 10n - 7\sqrt{n} - \frac{\lg^3 n}{\lg \lg n}$?

   (g) How long does it search for a value in an n-node binary search tree?

   (h) Given a sorted array $A[1 \,..\, n]$, how long does it take to construct a binary search tree for the elements of A?

   (i) How long does it take to construct a Huffman code, given an array of n character frequencies as input?

   (j) A train leaves Chicago at 8:00pm and travels south at 75 miles per hour. Another train leaves New Orleans at 1:00pm and travels north at 60 miles per hour. The conductors of both trains are playing a game of chess over the phone. After each player moves, the other player must move before his train has traveled five miles. How many moves do the two players make before their trains pass each other (somewhere near Memphis)?

2. Describe and analyze an algorithm to find the length of the longest substring that appears both forward and backward in an input string $T[1 \,.\, n]$. The forward and backward substrings must not overlap. Here are several examples:

   - Given the input string ALGORITHM, your algorithm should return 0.
   - Given the input string RECURSION, your algorithm should return 1, for the substring R.
   - Given the input string REDIVIDE, your algorithm should return 3, for the substring EDI. (The forward and backward substrings must not overlap!)
   - Given the input string DYNAMICPROGRAMMINGMANYTIMES, your algorithm should return 4, for the substring YNAM.

   For full credit, your algorithm should run in $O(n^2)$ time.

3. The *median* of a set of size n is its $\lceil n/2 \rceil$th largest element, that is, the element that is as close as possible to the middle of the set in sorted order. It's quite easy to find the median of a set in $O(n \log n)$ time—just sort the set and look in the middle—but you (correctly!) think that you can do better.

   During your lifelong quest for a faster median-finding algorithm, you meet and befriend the Near-Middle Fairy. Given any set X, the Near-Middle Fairy can find an element $m \in X$ that is *near* the middle of X in $O(1)$ time. Specifically, at least a third of the elements of X are smaller than m, and at least a third of the elements of X are larger than m.

   Describe and analyze an algorithm to find the median of a set in $O(n)$ time if you are allowed to ask the Near-Middle Fairy for help. *[Hint: You may need the* PARTITION *subroutine from* QUICKSORT.*]*

4. SUBSETSUM and PARTITION are two closely related NP-hard problems.

   - SUBSETSUM: Given a set X of integers and an integer k, does X have a subset whose elements sum up to k?
   - PARTITION: Given a set X of integers and an integer k, can X be partitioned into two subsets whose sums are equal?

   (a) Describe and analyze a polynomial-time reduction from SUBSETSUM to PARTITION.

   (b) Describe and analyze a polynomial-time reduction from PARTITION to SUBSETSUM.

5. Describe and analyze efficient algorithms to solve the following problems:

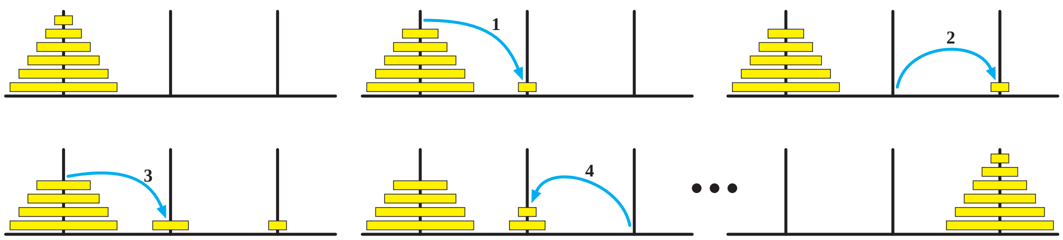
   (a) Given a set of n integers, does it contain a pair of elements a, b such that $a + b = 0$?

   (b) Given a set of n integers, does it contain three elements a, b, c such that $a + b + c = 0$?

Write your answers in the separate answer booklet.

1. In the well-known *Tower of Hanoi* problem, we have three spikes, one of which has a tower of n disks of different sizes, stacked with smaller disks on top of larger ones. In a single step, we are allowed to take the top disk on any spike and move it to the top of another spike. We are never allowed to place a larger disk on top of a smaller one. Our goal is to move all the disks from one spike to another.

   Hmmm.... You've probably known how to solve this problem since CS 125, so make it more interesting, let's add another constraint: The three spikes are arranged in a row, and we are also forbidden to move a disk directly from the left spike to the right spike or vice versa. In other words, we *must* move a disk either to or from the middle spike at *every* step.

   The first four steps required to move the disks from the left spike to the right spike.

   (a) **[4 pts]** Describe an algorithm that moves the stack of n disks from the left needle to the right needle in as few steps as possible.

   (b) **[6 pts]** **Exactly** how many steps does your algorithm take to move all n disks? A correct Θ-bound is worth 3 points. *[Hint: Set up and solve a recurrence.]*

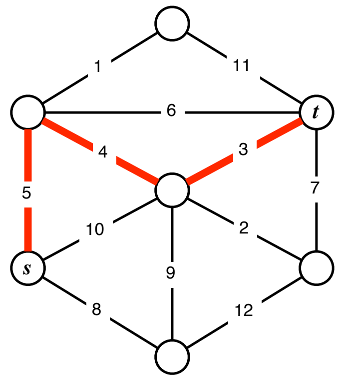
2. Consider a random walk on a path with vertices numbered $1, 2, \ldots, n$ from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n. In Midterm 2, you were asked to prove that if we start at vertex 1, the probability that the walk ends by falling off the *left* end of the path is exactly $n/(n + 1)$.

   (a) **[6 pts]** **Prove** that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n. *[Hint: Set up and solve a recurrence. Use the result from Midterm 2.]*

   (b) **[4 pts]** Suppose we start at vertex $n/2$ instead. State a tight Θ-bound on the expected length of the random walk in this case. **No proof is required.** *[Hint: Set up and solve a recurrence. Use part (a), even if you can't prove it.]*

3. **Prove** that any connected acyclic graph with n vertices has exactly $n - 1$ edges. Do not use the word "tree" or any well-known properties of trees; your proof should follow entirely from the definitions.

4. Consider a path between two vertices s and t in an undirected weighted graph G. The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t. (If there are no paths from u to v, the bottleneck distance between s and t is ∞.)



The bottleneck distance between s and t is 5.

Describe and analyze an algorithm to compute the bottleneck distance between every pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.

5. The 5Color asks, given a graph G, whether the vertices of a graph G can be colored with five colors so that no edge has two endpoints with the same color. You already know from class that this problem is NP-complete.

   Now consider the related problem 5Color$\pm$1: Given a graph G, can we color each vertex with an integer from the set $\{0, 1, 2, 3, 4\}$, so that for every edge, the colors of the two endpoints differ by exactly 1 modulo 5? (For example, a vertex with color 4 can only be adjacent to vertices colored 0 or 3.) We would like to show that 5Color$\pm$1 is NP-complete as well.

   (a) **[2 pts]** Show that 5Color$\pm$1 is in NP.

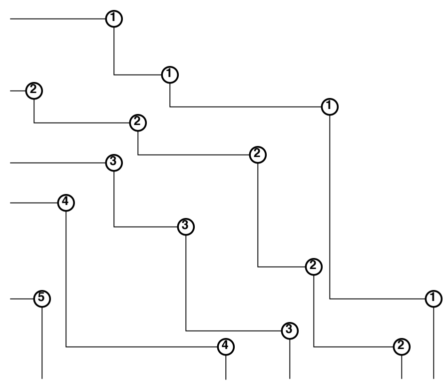
   (b) **[1 pt]** To prove that 5Color$\pm$1 is NP-hard (and therefore NP-complete), we must describe a polynomial time algorithm for *one* of the following problems. Which one?
   - Given an arbitrary graph G, compute a graph H such that 5Color(G) is true if and only if 5Color$\pm$1(H) is true.
   - Given an arbitrary graph G, compute a graph H such that 5Color$\pm$1(G) is true if and only if 5Color(H) is true.

   (c) **[1 pt]** Explain briefly why the following argument is not correct.

   > For any graph G, if 5Color$\pm$1(G) is true, then 5Color(G) is true (using the same coloring). Therefore if we could solve 5Color$\pm$1 quickly, we could also solve 5Color quickly. In other words, 5Color$\pm$1 is at least as hard as 5Color. We know that 5Color is NP-hard, so 5Color$\pm$1 must also be NP-hard!

   (d) **[6 pts]** Prove that 5Color$\pm$1 is NP-hard. *[Hint: Look at some small examples. Replace the edges of G with a simple gadget, so that the resulting graph H has the desired property from part (b).]*

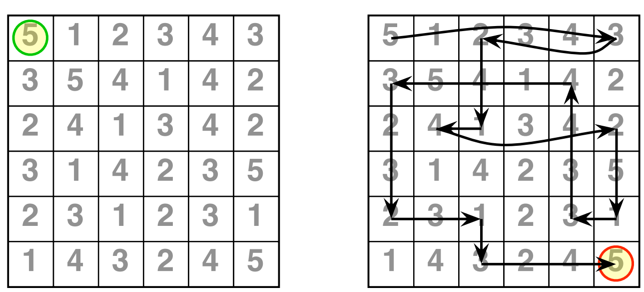
6. Let P be a set of points in the plane. Recall that a point $p \in P$ is *Pareto-optimal* if no other points in P are both above and to the right of p. Intuitively, the sequence of Pareto-optimal points forms a *staircase* with all the other points in P below and to the left. The *staircase layers* of P are defined recursively as follows. The empty set has no staircase layers. Otherwise, the first staircase layer contains all the Pareto-optimal points in P, and the remaining layers are the staircase layers of P minus the first layer.



A set of points with 5 staircase layers

Describe and analyze an algorithm to compute the number of staircase layers of a point set P as quickly as possible. For example, given the points illustrated above, your algorithm would return the number 5.

7. Consider the following puzzle played on an $n \times n$ square grid, where each square is labeled with a positive integer. A token is placed on one of the squares. At each turn, you may move the token left, right, up, or down; the distance you move the token must be equal to the number on the current square. For example, if the token is on a square labeled "3", you are allowed more the token three squares down, three square left, three squares up, or three squares right. You are never allowed to move the token off the board.



A sequence of legal moves from the top left corner to the bottom right corner.

(a) **[4 pts]** Describe and analyze an algorithm to determine, given an $n \times n$ array of labels and two squares s and t, whether there is a sequence of legal moves that takes the token from s to t.

(b) **[6 pts]** Suppose you are only given the $n \times n$ array of labels. Describe how to preprocess these values, so that afterwards, given any two squares s and t, you can determine in $O(1)$ time whether there is a sequence of legal moves from s to t.

Answer four of these seven problems; the lowest three scores will be dropped.

1. Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are five local minima in the following array:

9	7	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9

   We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array* ***must*** *have at least one local minimum. Why?]*

2. Consider a random walk on a path with vertices numbered $1, 2, \ldots, n$ from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n. In Midterm 2, you were asked to prove that if we start at vertex 1, the probability that the walk ends by falling off the *left* end of the path is exactly $n/(n+1)$.

   (a) **[6 pts] Prove** that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n. *[Hint: Set up and solve a recurrence. Use the result from Midterm 2.]*

   (b) **[4 pts]** Suppose we start at vertex $n/2$ instead. State **and prove** a tight Θ-bound on the expected length of the random walk in this case. *[Hint: Set up and solve a recurrence. Use part (a), even if you can't prove it.]*

3. **Prove** that any connected acyclic graph with $n \geq 2$ vertices has at least two vertices with degree 1. Do not use the words "tree" of "leaf", or any well-known properties of trees; your proof should follow entirely from the definitions.

4. Consider the following sketch of a "reverse greedy" algorithm. The input is a connected undirected graph G with weighted edges, represented by an adjacency list.

```
ReverseGreedyMST(G):
  sort the edges E of G by weight
  for i ← 1 to |E|
      e ← ith heaviest edge in E
      if G \ e is connected
          remove e from G
```

   (a) **[4 pts]** What is the worst-case running time of this algorithm? (Answering this question will require fleshing out a few details.)

   (b) **[6 pts] Prove** that the algorithm transforms G into its minimum spanning tree.

5. SubsetSum and Partition are two closely related NP-hard problems.

   - SubsetSum: Given a set X of integers and an integer k, does X have a subset whose elements sum up to k?
   - Partition: Given a set X of integers, can X be partitioned into two subsets whose sums are equal?

   (a) **[2 pts]** Prove that Partition and SubsetSum are both in NP.

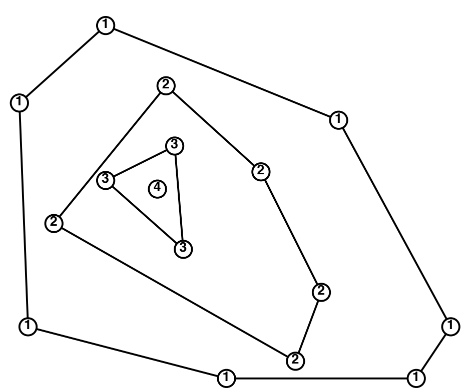
   (b) **[1 pt]** Suppose we knew that SubsetSum is NP-hard, and we wanted to prove that Partition is NP-hard. Which of the following arguments should we use?

   - Given a set X and an integer k, compute a set Y such that Partition(Y) is true if and only if SubsetSum(X, k) is true.
   - Given a set X, construct a set Y and an integer k such that Partition(X) is true if and only if SubsetSum(Y, k) is true.

   (c) **[3 pts]** Describe and analyze a polynomial-time reduction from Partition to SubsetSum. (See part (b).)

   (d) **[4 pts]** Describe and analyze a polynomial-time reduction from SubsetSum to Partition. (See part (b).)

6. Let P be a set of points in the plane. The *convex layers* of P are defined recursively as follows. If P is empty, it ha no convex layers. Otherwise, the first convex layer is the convex hull of P, and the remaining convex layers are the convex layers of P minus its convex hull.

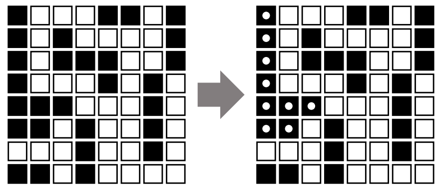


A set of points with 4 convex layers

Describe and analyze an algorithm to compute the number of convex layers of a point set P as quickly as possible. For example, given the points illustrated above, your algorithm would return the number 4.

7. (a) **[4 pts]** Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$.

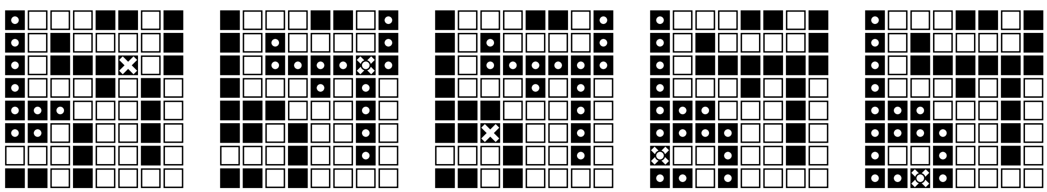
   For example, given the bitmap below as input, your algorithm should return the number 9, because the largest conected black component (marked with white dots on the right) contains nine pixels.



9

   (b) **[4 pts]** Design and analyze an algorithm BLACKEN(i, j) that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

   For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the BLACKEN algorithm.



9 14 14 16 17

   (c) **[2 pts]** What is the *worst-case* running time of your BLACKEN algorithm?

Write your answers in the separate answer booklet.

1. A *data stream* is an extremely long sequence of items that you can only read only once, in order. A good example of a data stream is the sequence of packets that pass through a router. Data stream algorithms must process each item in the stream quickly, using very little memory; there is simply too much data to store, and it arrives too quickly for any complex computations. Every data stream algorithm looks roughly like this:

```
DoSomethingInteresting(stream S):
  repeat
      x ← next item in S
      ⟨⟨do something fast with x⟩⟩
  until S ends
  return ⟨⟨something⟩⟩
```

   Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, *without knowing the length of the stream in advance*. Your algorithm should spend $O(1)$ time per stream element and use $O(1)$ space (not counting the stream itself). Assume you have a subroutine Random(n) that returns a random integer between 1 and n, each with equal probability, given any integer n as input.

2. Consider a random walk on a path with vertices numbered $1, 2, \ldots, n$ from left to right. We start at vertex 1. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n.

   Prove that the probability that the walk ends by falling off the *left* end of the path is exactly $n/(n+1)$. *[Hint: Set up a recurrence and verify that $n/(n+1)$ satisfies it.]*

3. Consider the following algorithms for maintaining a family of disjoint sets. The Union algorithm uses a heuristic called *union by size*.

```
MakeSet(x):
  parent(x) ← x
  size(x) ← 1
```

```
Find(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x
```

```
Union(x, y):
  x̄ ← Find(x)
  ȳ ← Find(y)
  if size(x̄) < size(ȳ)
      parent(x̄) ← ȳ
      size(x̄) ← size(x̄) + size(ȳ)
  else
      parent(ȳ) ← x̄
      size(ȳ) ← size(x̄) + size(ȳ)
```

   Prove that if we use union by size, Find(x) runs in $O(\log n)$ time *in the worst case*, where n is the size of the set containing element x.

4. Recall the SUBSETSUM problem: Given a set X of integers and an integer k, does X have a subset whose elements sum to k?

   (a) **[7 pts]** Describe and analyze an algorithm that solves SUBSETSUM in time $O(nk)$.

   (b) **[3 pts]** SUBSETSUM is NP-hard. Does part (a) imply that P=NP? Justify your answer.

5. Suppose we want to maintain a set X of numbers under the following operations:

   - INSERT(x): Add x to the set X.
   - PRINTANDDELETEBETWEEN(a, z): Print every element $x \in X$ such that $a \leq x \leq z$, in order from smallest to largest, and then delete those elements from X.

   For example, PRINTANDDELETEBETWEEN($-\infty, \infty$) prints all the elements of X in sorted order and then deletes everything.

   (a) **[6 pts]** Describe and analyze a data structure that supports these two operations, each in $O(\log n)$ *amortized* time, where n is the maximum number of elements in X.

   (b) **[2 pts]** What is the running time of your INSERT algorithm in the worst case?

   (c) **[2 pts]** What is the running time of your PRINTANDDELETEBETWEEN algorithm in the worst case?

CS 473G: Combinatorial Algorithms, Fall 2005
Homework 0

Due Thursday, September 1, 2005, at the beginning of class (12:30pm CDT)

Name:	
Net ID:	Alias:

☐ I understand the Homework Instructions and FAQ.

- Neatly print your full name, your NetID, and an alias of your choice in the boxes above. Grades will be listed on the course web site by alias. Please write the same alias on every homework and exam! For privacy reasons, your alias should not resemble your name or NetID. By providing an alias, you agree to let us list your grades; if you do not provide an alias, your grades will not be listed. ***Never*** *give us your Social Security number!*

- Read the "Homework Instructions and FAQ" on the course web page, and then check the box above. This page describes what we expect in your homework solutions—start each numbered problem on a new sheet of paper, write your name and NetID on every page, don't turn in source code, analyze and prove everything, use good English and good logic, and so on—as well as policies on grading standards, regrading, and plagiarism. **See especially the course policies regarding the magic phrases "I don't know" and "and so on".** If you have *any* questions, post them to the course newsgroup or ask during lecture.

- Don't forget to submit this cover sheet with the rest of your homework solutions.

- This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, discrete probability, and most importantly, induction—to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Chapters 1–10 of CLRS should be sufficient review, but you may also want consult your discrete mathematics and data structures textbooks.

- Every homework will have five required problems. Most homeworks will also include one extra-credit problem and several practice (no-credit) problems. Each numbered problem is worth 10 points.

#	1	2	3	4	5	6*	Total
Score							
Grader							

1. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway, just for practice. Assume reasonable but nontrivial base cases. **If your solution requires specific base cases, state them!**

   (a) $A(n) = 2A(n/4) + \sqrt{n}$

   (b) $B(n) = \max\limits_{n/3<k<2n/3} \big(B(k) + B(n-k) + n\big)$

   (c) $C(n) = 3C(n/3) + n/\lg n$

   (d) $D(n) = 3D(n-1) - 3D(n-2) + D(n-3)$

   (e) $E(n) = \dfrac{E(n-1)}{3E(n-2)}$ *[Hint: This is easy!]*

   (f) $F(n) = F(n-2) + 2/n$

   (g) $G(n) = 2G\big(\lceil (n+3)/4 \rceil - 5n/\sqrt{\lg n} + 6\lg\lg n\big) + 7\sqrt[8]{n-9} - \lg^{10} n/\lg\lg n + 11^{\lg^* n} - 12$

   *(h) $H(n) = 4H(n/2) - 4H(n/4) + 1$ *[Hint: Careful!]*

   (i) $I(n) = I(n/2) + I(n/4) + I(n/8) + I(n/12) + I(n/24) + n$

   $\bigstar$(j) $J(n) = 2\sqrt{n} \cdot J(\sqrt{n}) + n$
   [Hint: First solve the secondary recurrence $j(n) = 1 + j(\sqrt{n})$.]

2. Penn and Teller agree to play the following game. Penn shuffles a standard deck[1] of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3♣), at which point the remaining undrawn cards instantly burst into flames and the game is over.

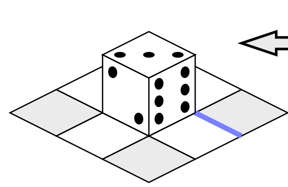
   The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the previous card he gave to Penn, he gives the new card to Penn. To make the rules unambiguous, they agree on the numerical values $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

   (a) What is the expected number of cards that Teller draws?
   (b) What is the expected *maximum* value among the cards Teller gives to Penn?
   (c) What is the expected *minimum* value among the cards Teller gives to Penn?
   (d) What is the expected number of cards that Teller gives to Penn?

   Full credit will be given only for *exact* answers (with correct proofs, of course).

[1] In a standard deck of 52 cards, each card has a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$ and a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$, and every possible suit-value pair appears in the deck exactly once. Penn and Teller normally use exploding razor-sharp ninja throwing cards for this trick.

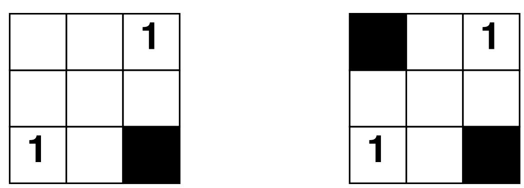
3. A *rolling die maze* is a puzzle involving a standard six-sided die[2] and a grid of squares. You should imagine the grid lying on top of a table; the die always rests on and exactly covers one square. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.



Rolling a die.

Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number of pips on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

For example, here are two rolling die mazes. Black squares are blocked. The maze on the left can be solved by placing the die on the lower left square with 1 pip on the top face, and then rolling it north, then north, then east, then east. The maze on the right is not solvable.



Two rolling die mazes. Only the maze on the left is solvable.

(a) Suppose the input is a two-dimensional array $L[1..n][1..n]$, where each entry $L[i][j]$ stores the label of the square in the ith row and jth column, where 0 means the square is free and -1 means the square is blocked. Describe and analyze a polynomial-time algorithm to determine whether the given rolling die maze is solvable.

*(b) Now suppose the maze is specified *implicitly* by a list of labeled and blocked squares. Specifically, suppose the input consists of an integer M, specifying the height and width of the maze, and an array $S[1..n]$, where each entry $S[i]$ is a triple (x, y, L) indicating that square (x, y) has label L. As in the explicit encoding, label -1 indicates that the square is blocked; free squares are not listed in S at all. Describe and analyze an efficient algorithm to determine whether the given rolling die maze is solvable. For full credit, the running time of your algorithm should be polynomial in the input size n.

[Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can only be solved if the initial position is chosen correctly.]

[2] A standard die is a cube, where each side is labeled with a different number of dots, called *pips*, between 1 and 6. The labeling is chosen so that any pair of opposite sides has a total of 7 pips.

4. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons will always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. (Like most things, revenge is a foreign concept to pigeons.) Surprisingly, the overall pecking order in a set of pigeons can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

   Prove that any set of pigeons can be arranged in a row so that every pigeon pecks the pigeon immediately to its right.

5. Scientists have recently discovered a planet, tentatively named "Ygdrasil", which is inhabited by a bizarre species called "vodes". All vodes trace their ancestry back to a particular vode named Rudy. Rudy is still quite alive, as is every one of his many descendants. Vodes reproduce asexually, like bees; each vode has exactly one parent (except Rudy, who has no parent). There are three different colors of vodes—cyan, magenta, and yellow. The color of each vode is correlated exactly with the number and colors of its children, as follows:

   - Each cyan vode has two children, exactly one of which is yellow.
   - Each yellow vode has exactly one child, which is not yellow.
   - Magenta vodes have no children.

   In each of the following problems, let C, M, and Y respectively denote the number of cyan, magenta, and yellow vodes on Ygdrasil.

   (a) Prove that $M = C + 1$.

   (b) Prove that either $Y = C$ or $Y = M$.

   (c) Prove that $Y = M$ if and only if Rudy is yellow.

   [Hint: Be very careful to ***prove*** *that you have considered* ***all*** *possibilities.]*

*6. **[Extra credit]**[3]

Lazy binary is a variant of standard binary notation for representing natural numbers where we allow each "bit" to take on one of three values: 0, 1, or 2. Lazy binary notation is defined inductively as follows.

- The lazy binary representation of zero is 0.
- Given the lazy binary representation of any non-negative integer n, we can construct the lazy binary representation of $n+1$ as follows:
  (a) increment the rightmost digit;
  (b) if any digit is equal to 2, replace the rightmost 2 with 0 and increment the digit immediately to its left.

Here are the first several natural numbers in lazy binary notation:

0, 1, 10, 11, 20, 101, 110, 111, 120, 201, 210, 1011, 1020, 1101, 1110, 1111, 1120, 1201, 1210, 2011, 2020, 2101, 2110, 10111, 10120, 10201, 10210, 11011, 11020, 11101, 11110, 11111, 11120, 11201, 11210, 12011, 12020, 12101, 12110, 20111, 20120, 20201, 20210, 21011, 21020, 21101, 21110, 101111, 101120, 101201, 101210, 102011, 102020, 102101, 102110, ...

(a) Prove that in any lazy binary number, between any two 2s there is at least one 0, and between two 0s there is at least one 2.

(b) Prove that for any natural number N, the sum of the digits of the lazy binary representation of N is exactly $\lfloor \lg(N+1) \rfloor$.

[3]The "I don't know" rule does not apply to extra credit problems. There is no such thing as "partial extra credit".

Practice Problems

The remaining problems are for practice only. Please do not submit solutions. On the other hand, feel free to discuss these problems in office hours or on the course newsgroup.

1. Sort the functions in each box from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to turn in proofs (in fact, please *don't* turn in proofs), but you should do them anyway, just for practice.

1	$\lg n$	$\lg^2 n$	$\sqrt{n}$	n	n^2	$2^{\sqrt{n}}$	$\sqrt{2}^n$
$2^{\sqrt{\lg n}}$	$2^{\lg \sqrt{n}}$	$\sqrt{2^{\lg n}}$	$\sqrt{2}^{\lg n}$	$\lg 2^{\sqrt{n}}$	$\lg \sqrt{2}^n$	$\lg \sqrt{2^n}$	$\sqrt{\lg 2^n}$
$\lg n^{\sqrt{2}}$	$\lg \sqrt{n}^2$	$\lg \sqrt{n^2}$	$\sqrt{\lg n^2}$	$\lg^2 \sqrt{n}$	$\lg^{\sqrt{2}} n$	$\sqrt{\lg^2 n}$	$\sqrt{\lg n}^2$

To simplify your answers, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

2. Recall the standard recursive definition of the Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Prove the following identities for all positive integers n and m.

   (a) F_n is even if and only if n is divisible by 3.

   (b) $\sum_{i=0}^{n} F_i = F_{n+2} - 1$

   (c) $F_n^2 - F_{n+1}F_{n-1} = (-1)^{n+1}$

   ★(d) If n is an integer multiple of m, then F_n is an integer multiple of F_m.

3. Penn and Teller have a special deck of fifty-two cards, with no face cards and nothing but clubs—the ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, …, 52 of clubs. (They're big cards.) Penn shuffles the deck until each each of the 52! possible orderings of the cards is equally likely. He then takes cards one at a time from the top of the deck and gives them to Teller, stopping as soon as he gives Teller the three of clubs.

   (a) On average, how many cards does Penn give Teller?

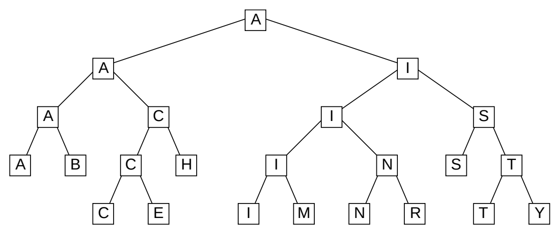
   (b) On average, what is the smallest-numbered card that Penn gives Teller?

   ⋆(c) On average, what is the largest-numbered card that Penn gives Teller?

   Prove that your answers are correct. (If you have to appeal to "intuition" or "common sense", your answers are probably wrong.) *[Hint: Solve for an n-card deck, and then set n to 52.]*

4. Algorithms and data structures were developed millions of years ago by the Martians, but not quite in the same way as the recent development here on Earth. Intelligent life evolved independently on Mars' two moons, Phobos and Deimos.[4] When the two races finally met on the surface of Mars, after thousands of years of separate philosophical, cultural, religious, and scientific development, their disagreements over the proper structure of binary search trees led to a bloody (or more accurately, ichorous) war, ultimately leading to the destruction of all Martian life.

   A *Phobian* binary search tree is a full binary tree that stores a set X of search keys. The root of the tree stores the *smallest* element in X. If X has more than one element, then the left subtree stores all the elements less than some pivot value p, and the right subtree stores everything else. Both subtrees are *nonempty* Phobian binary search trees. The actual pivot value p is *never* stored in the tree.



   A Phobian binary search tree for the set $\{M, A, R, T, I, N, B, Y, S, E, C, H\}$.

   (a) Describe and analyze an algorithm $\text{FIND}(x, T)$ that returns TRUE if x is stored in the Phobian binary search tree T, and FALSE otherwise.

   (b) A *Deimoid* binary search tree is almost exactly the same as its Phobian counterpart, except that the *largest* element is stored at the root, and both subtrees are Deimoid binary search trees. Describe and analyze an algorithm to transform an n-node Phobian binary search tree into a Deimoid binary search tree in $O(n)$ time, using as little additional space as possible.

5. Tatami are rectangular mats used to tile floors in traditional Japanese houses. Exact dimensions of tatami mats vary from one region of Japan to the next, but they are always twice as long in one dimension than in the other. (In Tokyo, the standard size is 180cm×90cm.)

   (a) How many different ways are there to tile a $2 \times n$ rectangular room with 1×2 tatami mats? Set up a recurrence and derive an *exact* closed-form solution. *[Hint: The answer involves a familiar recursive sequence.]*

   (b) According to tradition, tatami mats are always arranged so that four corners never meet. How many different *traditional* ways are there to tile a $3 \times n$ rectangular room with 1×2 tatami mats? Set up a recurrence and derive an *exact* closed-form solution.

   *(c) How many different *traditional* ways are there to tile an $n \times n$ square with 1×2 tatami mats? Prove your answer is correct.

[4]Greek for "fear" and "panic", respectively. Doesn't that make you feel better?

CS 473G: Combinatorial Algorithms, Fall 2005
Homework 1

Due Tuesday, September 13, 2005, by midnight (11:59:59pm CDT)

Name:	
Net ID:	Alias:

Name:	
Net ID:	Alias:

Name:	
Net ID:	Alias:

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade.

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Staple this sheet to the top of your answer to problem 1.

There are two steps required to prove NP-completeness: (1) Prove that the problem is in NP, by describing a polynomial-time verification algorithm. (2) Prove that the problem is NP-hard, by describing a polynomial-time reduction from some other NP-hard problem. Showing that the reduction is correct requires proving an if-and-only-if statement; don't forget to prove both the "if" part and the "only if" part.

Required Problems

1. Some NP-Complete problems

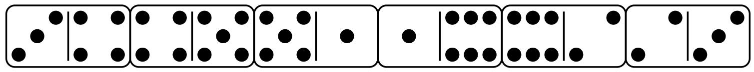
   (a) Show that the problem of deciding whether one graph is a subgraph of another is NP-complete.

   (b) Given a boolean circuit that embeds in the plane so that no 2 wires cross, PlanarCircuitSat is the problem of determining if there is a boolean assignment to the inputs that makes the circuit output true. Prove that PlanarCircuitSat is NP-Complete.

   (c) Given a set S with $3n$ numbers, 3partition is the problem of determining if S can be partitioned into n disjoint subsets, each with 3 elements, so that every subset sums to the same value. Given a set S and a collection of three element subsets of S, X3M (or *exact 3-dimensional matching*) is the problem of determining whether there is a subcollection of n disjoint triples that exactly cover S.

   Describe a polynomial-time reduction from 3partition to X3M.

(d) A *domino* is a 1×2 rectangle divided into two squares, each of which is labeled with an integer.[1] In a *legal arrangement* of dominoes, the dominoes are lined up end-to-end so that the numbers on adjacent ends match.



A legal arrangement of dominos, where every integer between 1 and 6 appears twice

Prove that the following problem is NP-complete: Given an arbitrary collection D of dominoes, is there a legal arrangement of a subset of D in which every integer between 1 and n appears exactly twice?

2. Prove that the following problems are all polynomial-time equivalent, that is, if *any* of these problems can be solved in polynomial time, then *all* of them can.

   - Clique: Given a graph G and an integer k, does there exist a clique of size k in G?
   - FindClique: Given a graph G and an integer k, find a clique of size k in G if one exists.
   - MaxClique: Given a graph G, find the size of the largest clique in the graph.
   - FindMaxClique: Given a graph G, find a clique of maximum size in G.

3. Consider the following problem: Given a set of n points in the plane, find a set of line segments connecting the points which form a closed loop and do not intersect each other.

   Describe a linear time reduction from the problem of sorting n numbers to the problem described above.

4. In graph coloring, the vertices of a graph are assigned colors so that no adjacent vertices recieve the same color. We saw in class that determining if a graph is 3-colorable is NP-Complete.

   Suppose you are handed a magic black box that, given a graph as input, tells you *in constant time* whether or not the graph is 3-colorable. Using this black box, give a *polynomial-time* algorithm to 3-color a graph.

5. Suppose that Cook had proved that graph coloring was NP-complete first, instead of CircuitSAT. Using only the fact that graph coloring is NP-complete, show that CircuitSAT is NP-complete.

[1]These integers are usually represented by pips, exactly like dice. On a standard domino, the number of pips on each side is between 0 and 6; we will allow arbitrary integer labels. A standard set of dominoes has one domino for each possible unordered pair of labels; we do not require that every possible label pair is in our set.

Practice Problems

1. Given an initial configuration consisting of an undirected graph $G = (V, E)$ and a function $p : V \to \mathbb{N}$ indicating an initial number of pebbles on each vertex, PEBBLE-DESTRUCTION asks if there is a sequence of pebbling moves starting with the initial configuration and ending with a single pebble on only one vertex of V. Here, a pebbling move consists of removing two pebbles from a vertex v and adding one pebble to a neighbor of v. Prove that PEBBLE-DESTRUCTION is NP-complete.

2. Consider finding the median of 5 numbers by using only comparisons. What is the *exact* worst case number of comparisons needed to find the median? To prove your answer is correct, you must exhibit both an algorithm that uses that many comparisons and a proof that there is no faster algorithm. Do the same for 6 numbers.

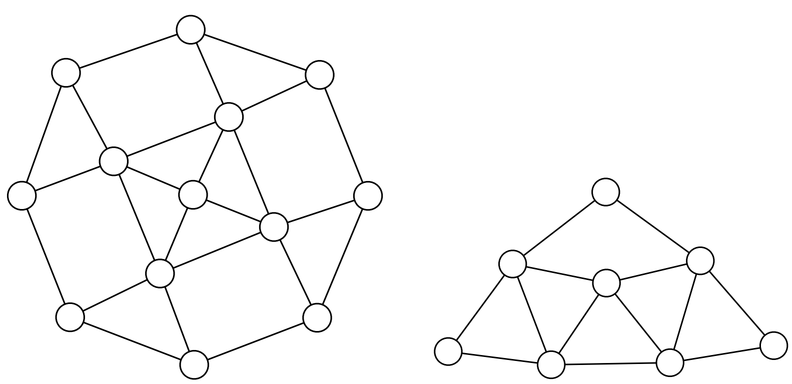
3. PARTITION is the problem of deciding, given a set S of numbers, whether it can be partitioned into two subsets whose sums are equal. (A *partition* of S is a collection of disjoint subsets whose union is S.) SUBSETSUM is the problem of deciding, given a set S of numbers and a target sum t, whether any subset of number in S sum to t.

   (a) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.

   (b) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.

4. Recall from class that the problem of deciding whether a graph can be colored with three colors, so that no edge joins nodes of the same color, is NP-complete.

   (a) Using the gadget in Figure 1(a), prove that deciding whether a *planar* graph can be 3-colored is NP-complete. *[Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]*



(a) (b)

Figure 1. (a) Gadget for planar 3-colorability. (b) Gadget for degree-4 planar 3-colorability.

   (b) Using the previous result and the gadget in figure 1(b), prove that deciding whether a planar graph *with maximum degree 4* can be 3-colored is NP-complete. *[Hint: Show that you can replace any vertex with degree greater than 4 with a collection of gadgets connected in such a way that no degree is greater than four.]*

5. (a) Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian. Describe a polynomial-time algorithm to find a hamiltonian cycle in an undirected bipartite graph, or establish that no such cycle exists.

   (b) Describe a polynomial time algorithm to find a hamiltonian *path* in a directed acyclic graph, or establish that no such path exists.

   (c) Why don't these results imply that P=NP?

6. Consider the following pairs of problems:

   (a) MIN SPANNING TREE and MAX SPANNING TREE

   (b) SHORTEST PATH and LONGEST PATH

   (c) TRAVELING SALESMAN PROBLEM and VACATION TOUR PROBLEM (the longest tour is sought).

   (d) MIN CUT and MAX CUT (between s and t)

   (e) EDGE COVER and VERTEX COVER

   (f) TRANSITIVE REDUCTION and MIN EQUIVALENT DIGRAPH

   (all of these seem dual or opposites, except the last, which are just two versions of minimal representation of a graph).

   Which of these pairs are polytime equivalent and which are not? Why?

7. Prove that PRIMALITY (Given n, is n prime?) is in NP $\cap$ co-NP. *[Hint: co-NP is easy—What's a certificate for showing that a number is composite? For NP, consider a certificate involving primitive roots and recursively their primitive roots. Show that this tree of primitive roots can be verified an used to show that n is prime in polynomial time.]*

8. How much wood would a woodchuck chuck if a woodchuck could chuck wood?

CS 473G: Combinatorial Algorithms, Fall 2005
Homework 2

Due Thursday, September 22, 2005, by midnight (11:59:59pm CDT)

Name:	
Net ID:	Alias:

Name:	
Net ID:	Alias:

Name:	
Net ID:	Alias:

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade.

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Staple this sheet to the top of your homework.

Required Problems

1. (a) Suppose Lois has an algorithm to compute the shortest common supersequence of two arrays of integers in $O(n)$ time. Describe an $O(n \log n)$-time algorithm to compute the longest common subsequence of two arrays of integers, using Lois's algorithm as a subroutine.

   (b) Describe an $O(n \log n)$-time algorithm to compute the longest increasing subsequence of an array of integers, using Lois's algorithm as a subroutine.

   (c) Now suppose Lisa has an algorithm that can compute the longest increasing subsequence of an array of integers in $O(n)$ time. Describe an $O(n \log n)$-time algorithm to compute the longest common subsequence of two arrays $A[1 .. n]$ and $B[1 .. n]$ of integers, **where $\boldsymbol{A[i] \neq A[j]}$ for all $\boldsymbol{i \neq j}$**, using Lisa's algorithm as a subroutine.[1]

[1]For extra credit, remove the assumption that the elements of A are distinct. This is probably impossible.

2. In a previous incarnation, you worked as a cashier in the lost 19th-century Antarctican colony of Nadira, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource on Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadira, called Dream Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365.[2]

   (a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream Dollar bills than the minimum possible.

   (b) Describe and analyze an efficient algorithm that computes, given an integer n, the minimum number of bills needed to make n Dream Dollars.

3. Scientists have branched out from the bizarre planet of Yggdrasil to study the vodes which have settled on Ygdrasil's moon, Xryltcon. All vodes on Xryltcon are descended from the first vode to arrive there, named George. Each vode has a color, either cyan, magenta, or yellow, but breeding patterns are *not* the same as on Yggdrasil; every vode, regardless of color, has either two children (with arbitrary colors) or no children.

   George and all his descendants are alive and well, and they are quite excited to meet the scientists who wish to study them. Unsurprisingly, these vodes have had some strange mutations in their isolation on Xryltcon. Each vode has a *weirdness* rating; weirder vodes are more interesting to the visiting scientists. (Some vodes even have negative weirdness ratings; they make other vodes more boring just by standing next to them.)

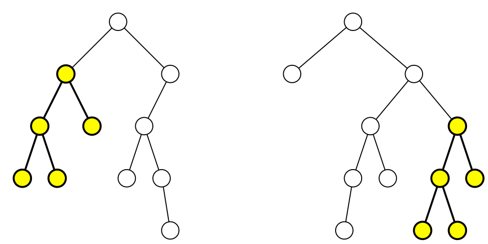
   Also, Xryltconian society is strictly governed by a number of sacred cultural traditions.

   - No cyan vode may be in the same room as its non-cyan children (if it has any).
   - No magenta vode may be in the same room as its parent (if it has one).
   - Each yellow vode must be attended at all times by its grandchildren (if it has any).
   - George must be present at any gathering of more than fifty vodes.

   The scientists have exactly one chance to study a group of vodes in a single room. You are given the family tree of all the vodes on Xryltcon, along with the wierdness value of each vode. Design and analyze an efficient algorithm to decide which vodes the scientists should invite to maximize the sum of the wierdness values of the vodes in the room. Be careful to respect all of the vodes' cultural taboos.

[2] For more details on the history and culture of Nadira, including images of the various denominations of Dream Dollars, see http://www.dream-dollars.com. Really.

4. A *subtree* of a (rooted, ordered) binary tree T consists of a node and all its descendants. Design and analyze an efficient algorithm to compute the *largest common subtree* of two given binary trees T_1 and T_2, that is, the largest subtree of T_1 that is isomorphic to a subtree in T_2. The *contents* of the nodes are irrelevant; we are only interested in matching the underlying combinatorial structure.



Two binary trees, with their largest common subtree emphasized

5. Let $D[1\,..\,n]$ be an array of digits, each an integer between 0 and 9. An *digital subsequence* of D is an sequence of positive integers composed in the usual way from disjoint substrings of D. For example, $3, 4, 5, 6, 23, 38, 62, 64, 83, 279$ is an increasing digital subsequence of the first several digits of π:

$$\boxed{3}, 1, \boxed{4}, 1, \boxed{5}, 9, \boxed{6}, \boxed{2, 3}, 4, \boxed{3, 8}, 4, \boxed{6, 2}, \boxed{6, 4}, 3, 3, \boxed{8, 3}, \boxed{2, 7, 9}$$

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the previous example has length 10.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D. *[Hint: Be careful about your computational assumptions. How long does it take to compare two k-digit numbers?]*

*6. *[Extra credit]* The *chromatic number* of a graph G is the minimum number of colors needed to color the nodes of G so that no pair of adjacent nodes have the same color.

(a) Describe and analyze a *recursive* algorithm to compute the chromatic number of an n-vertex graph in $O(4^n \operatorname{poly}(n))$ time. *[Hint: Catalan numbers play a role here.]*

(b) Describe and analyze an algorithm to compute the chromatic number of an n-vertex graph in $O(3^n \operatorname{poly}(n))$ time. *[Hint: Use dynamic programming. What is $(1+x)^n$?]*

(c) Describe and analyze an algorithm to compute the chromatic number of an n-vertex graph in $O((1+3^{1/3})^n \operatorname{poly}(n))$ time. *[Hint: Use (but don't regurgitate) the algorithm in the lecture notes that counts all the maximal independent sets in an n-vertex graph in $O(3^{n/3})$ time.]*

Practice Problems

*1. Describe an algorithm to solve 3SAT in time $O(\phi^n \operatorname{poly}(n))$, where $\phi = (1+\sqrt{5})/2$. *[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals.]*

2. Describe and analyze an algorithm to compute the longest increasing subsequence in an n-element array of integers in $O(n \log n)$ time. *[Hint: Modify the $O(n^2)$-time algorithm presented in class.]*

3. The *edit distance* between two strings A and B, denoted $\operatorname{Edit}(A, B)$, is the minimum number of insertions, deletions, or substitutions required to transform A into B (or vice versa). Edit distance is sometimes also called the *Levenshtein distance*.

   Let $\mathbf{A} = \{A_1, A_2, \ldots, A_k\}$ be a set of strings. The *edit radius* of $\mathbf{A}$ is the minimum over all strings X of the maximum edit distance from X to any string A_i:

$$\operatorname{EditRadius}(\mathbf{A}) = \min_{\text{strings } X} \max_{1 \le i \le k} \operatorname{Edit}(X, A_i)$$

   A string X that achieves this minimum is called an *edit center* of $\mathbf{A}$. A set of strings may have several edit centers, but the edit radius is unique.

   Describe an efficient algorithm to compute the edit radius of three given strings.

4. Given 5 sequences of numbers, each of length n, design and analyze an efficent algorithm to compute the longest common subsequence among all 5 sequences.

5. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the ith word is $W[i]$ pixels wide. We want to break the paragraph into several lines, each exactly L pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. (Look at the paragraph you are reading right now!) There must be at least one pixel of white space between any two words on the same line. Thus, if a line contains words i through j, then the amount of *extra* white space on that line is $L - j + i - \sum_{k=i}^{j} W[k]$.

   Define the *slop* of a paragraph layout as the sum, over all lines *except the last*, of the *cube* of the extra white space in each line. Describe an efficient algorithm to layout the paragraph with minimum slop, given the list $W[1 \,..\, n]$ of word widths as input. You can assume that $W[i] < L/2$ for each i, so that each line contains at least two words.

6. A *partition* of a positive integer n is a multiset of positive integers that sum to n. Traditionally, the elements of a partition are written in non-decreasing order, separated by + signs. For example, the integer 7 has exactly twelve partitions:

$$\begin{array}{ccc} 1+1+1+1+1+1+1 & 3+1+1+1+1 & 4+1+1+1 \\ 2+1+1+1+1+1 & 3+2+1+1 & 4+2+1 \\ 2+2+1+1+1 & 3+2+2 & 4+3 \\ 2+2+2+1 & 3+3+1 & 7 \end{array}$$

The *roughness* of a partition $a_1 + a_2 + \cdots + a_k$ is defined as follows:

$$\rho(a_1 + a_2 + \cdots + a_k) = \sum_{i=1}^{k-1} |a_{i+1} - a_i - 1| \; + \; a_k - 1$$

A *smoothest* partition of n is the partition of n with minimum roughness. Intuitively, the smoothest partition is the one closest to a descending arithmetic series $k + \cdots + 3 + 2 + 1$, which is the only partition that has roughness 0. For example, the smoothest partitions of 7 are $4 + 2 + 1$ and $3 + 2 + 1 + 1$:

$$\begin{array}{ccc} \rho(1+1+1+1+1+1+1) = 6 & \rho(3+1+1+1+1) = 4 & \rho(4+1+1+1) = 4 \\ \rho(2+1+1+1+1+1) = 4 & \rho(3+2+1+1) = 1 & \rho(4+2+1) = 1 \\ \rho(2+2+1+1+1) = 3 & \rho(3+2+2) = 2 & \rho(4+3) = 2 \\ \rho(2+2+2+1) = 2 & \rho(3+3+1) = 2 & \rho(7) = 7 \end{array}$$

Describe and analyze an algorithm to compute, given a positive integer n, a smoothest partition of n.

CS 473G: Combinatorial Algorithms, Fall 2005
Homework 3

Due Tuesday, October 18, 2005, at midnight

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

Starting with Homework 1, homeworks may be done in teams of up to three people. Each team turns in just one solution, and every member of a team gets the same grade.

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Staple this sheet to the top of your homework.

---

1. Consider the following greedy approximation algorithm to find a vertex cover in a graph:

```
GreedyVertexCover(G):
  C ← ∅
  while G has at least one edge
      v ← vertex in G with maximum degree
      G ← G \ v
      C ← C ∪ v
  return C
```

In class we proved that the approximation ratio of this algorithm is $O(\log n)$; your task is to prove a matching lower bound. Specifically, prove that for any integer n, there is a graph G with n vertices such that GreedyVertexCover(G) returns a vertex cover that is $\Omega(\log n)$ times larger than optimal.

2. Prove that for *any* constant k and *any* graph coloring algorithm A, there is a graph G such that $\boldsymbol{A(G) > OPT(G) + k}$, where $A(G)$ is the number of colors generated by algorithm A for graph G, and $OPT(G)$ is the optimal number of colors for G.

[Note: This does not contradict the possibility of a constant **factor** *approximation algorithm.]*

3. Let R be a set of rectangles in the plane, with horizontal and vertical edges. A *stabbing set* for R is a set of points S such that every rectangle in R contains at least one point in S. The *rectangle stabbing* problem asks, given a set R of rectangles, for the smallest stabbing set S.

   (a) Prove that the rectangle stabbing problem is NP-hard.

   (b) Describe and analyze an efficient approximation algorithm for the rectangle stabbing problem. Give bounds on the approximation ratio of your algorithm.

4. Consider the following approximation scheme for coloring a graph G.

```
TREECOLOR(G):
  T ← any spanning tree of G
  Color the tree T with two colors
  c ← 2

  for each edge (u, v) ∈ G \ T
      T ← T ∪ {(u, v)}

      if color(u) = color(v)    ⟨⟨Try recoloring u with an existing color⟩⟩
          for i ← 1 to c
              if no neighbor of u in T has color i
                  color(u) ← i

      if color(u) = color(v)    ⟨⟨Try recoloring v with an existing color⟩⟩
          for i ← 1 to c
              if no neighbor of v in T has color i
                  color(v) ← i

      if color(u) = color(v)    ⟨⟨Give up and use a new color⟩⟩
          c ← c + 1
          color(u) ← c

  return c
```

   (a) Prove that this algorithm correctly colors any bipartite graph.

   (b) Prove an upper bound C on the number of colors used by this algorithm. Give a sample graph and run that requires C colors.

   (c) Does this algorithm approximate the minimum number of colors up to a constant factor? In other words, is there a constant α such that $\textsc{TreeColor}(G) < \alpha \cdot OPT(G)$ for any graph G? Justify your answer.

5. In the *bin packing* problem, we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. It's not hard to show that this problem is NP-Hard; this question asks you to analyze a few common approximation algorithms. In each case, the input is an array $W[1..n]$ of weights, and the output is the number of bins used.

```
NextFit(W[1..n]):
  b ← 0
  Total[0] ← ∞

  for i ← 1 to n
      if Total[b] + W[i] > 1
          b ← b + 1
          Total[b] ← W[i]
      else
          Total[b] ← Total[b] + W[i]

  return b
```

```
FirstFit(W[1..n]):
  b ← 0

  for i ← 1 to n
      j ← 1; found ← False

      while j ≤ b and found = False
          if Total[j] + W[i] ≤ 1
              Total[j] ← Total[j] + W[i]
              found ← True
          j ← j + 1

      if found = False
          b ← b + 1
          Total[b] = W[i]

  return b
```

(a) Prove that NextFit uses at most twice the optimal number of bins.

(b) Prove that FirstFit uses at most twice the optimal number of bins.

(c) Prove that if the weight array W is initially sorted in decreasing order, then FirstFit uses at most $(4 \cdot OPT + 1)/3$ bins, where OPT is the optimal number of bins. The following facts may be useful (but you need to prove them if your proof uses them):

- In the packing computed by FirstFit, every item with weight more than $1/3$ is placed in one of the first OPT bins.
- FirstFit places at most $OPT - 1$ items outside the first OPT bins.

CS 473G: Combinatorial Algorithms, Fall 2005
Homework 4

Due Thursday, October 27, 2005, at midnight

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

Homeworks may be done in teams of up to three people. Each team turns in just one solution; every member of a team gets the same grade.

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Staple this sheet to the top of your solution to problem 1.

If you are an I2CS student, print "(I2CS)" next to your name. Teams that include both on-campus and I2CS students can have up to four members. Any team containing both on-campus and I2CS students automatically receives 3 points of extra credit.

> For the rest of the semester, unless specifically stated otherwise, you may assume that the function RANDOM(m) returns an integer chosen uniformly at random from the set $\{1, 2, , \ldots, m\}$ in $O(1)$ time. For example, a fair coin flip is obtained by calling RANDOM(2).

1. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

   What is the *exact* expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. *[Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?]*

2. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

   - MAKEQUEUE: Return a new priority queue containing the empty set.
   - FINDMIN(Q): Return the smallest element of Q (if any).
   - DELETEMIN(Q): Remove the smallest element in Q (if any).
   - INSERT(Q, x): Insert element x into Q, if it is not already there.
   - DECREASEKEY(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
   - DELETE(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
   - MELD(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

   A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

```
MELD(Q1, Q2):
  if Q1 is empty return Q2
  if Q2 is empty return Q1

  if key(Q1) > key(Q2)
      swap Q1 ↔ Q2

  with probability 1/2
      left(Q1) ← MELD(left(Q1), Q2)
  else
      right(Q1) ← MELD(right(Q1), Q2)

  return Q1
```

   (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of MELD(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if each left/right choice is made with equal probability?]*

   (b) Prove that MELD(Q_1, Q_2) runs in $O(\log n)$ time with high probability.

   (c) Show that each of the other meldable priority queue operations cab be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

3. Let $M[1..n][1..n]$ be an $n \times n$ matrix in which every row and every column is sorted. Such an array is called *totally monotone*. No two elements of M are equal.

   (a) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, compute the number of elements of M smaller than $M[i][j]$ and larger than $M[i'][j']$.

   (b) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, return an element of M chosen uniformly at random from the elements smaller than $M[i][j]$ and larger than $M[i'][j']$. Assume the requested range is always non-empty.

   (c) Describe and analyze a randomized algorithm to compute the median element of M in $O(n \log n)$ expected time.

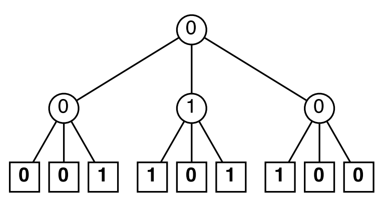
4. Let $X[1..n]$ be an array of n distinct real numbers, and let $N[1..n]$ be an array of indices with the following property: If $X[i]$ is the largest element of X, then $X[N[i]]$ is the smallest element of X; otherwise, $X[N[i]]$ is the smallest element of X that is larger than $X[i]$.

   For example:

i	1	2	3	4	5	6	7	8	9
$X[i]$	83	54	16	31	45	99	78	62	27
$N[i]$	6	8	9	5	2	3	1	7	4

   Describe and analyze a randomized algorithm that determines whether a given number x appears in the array X in $O(\sqrt{n})$ expected time. **Your algorithm may not modify the arrays X and $Next$.**

5. A *majority tree* is a complete ternary tree with depth n, where every leaf is labeled either 0 or 1. The *value* of a leaf is its label; the *value* of any internal node is the majority of the values of its three children. Consider the problem of computing the value of the root of a majority tree, given the sequence of 3^n leaf labels as input. For example, if $n = 2$ and the leaves are labeled $0, 0, 1, 1, 0, 1, 1, 0, 0$, the root has value 0.



A majority tree with depth $n = 2$.

   (a) Prove that *any* deterministic algorithm that computes the value of the root of a majority tree *must* examine every leaf. *[Hint: Consider the special case $n = 1$. Recurse.]*

   (b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. *[Hint: Consider the special case $n = 1$. Recurse.]*

*6. *[Extra credit]* In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0, 1]$, but in practice, computers only have access to random *bits*. This problem asks you to analyze a modification of treaps that takes this limitation into account.

Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1 \mathinner{..} \infty]$ of random bits, which is interpreted as the rational number

$$\mathit{priority}(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or 'reveal') new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LargerPriority(v, w):
  for i ← 1 to ∞
      if i > ℓ_v
          ℓ_v ← i;  π_v[i] ← RandomBit
      if i > ℓ_w
          ℓ_w ← i;  π_w[i] ← RandomBit
      if π_v[i] > π_w[i]
          return v
      else if π_v[i] < π_w[i]
          return w
```

Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LargerPriority during these insertions.

(a) Prove that $E[L] = \Theta(n)$.

(b) Prove that $E[\ell_v] = \Theta(1)$ for any node v. *[Hint: This is equivalent to part (a). Why?]*

(c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. *[Hint: Why doesn't this contradict part (b)?]*

CS 473G: Combinatorial Algorithms, Fall 2005
Homework 5

Due Thursday, November 17, 2005, at midnight

(because you *really* don't want homework due over Thanksgiving break)

Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:
Name:	
Net ID:	Alias:

Homeworks may be done in teams of up to three people. Each team turns in just one solution; every member of a team gets the same grade.

Neatly print your name(s), NetID(s), and the alias(es) you used for Homework 0 in the boxes above. Attach this sheet (or the equivalent information) to the top of your solution to problem 1.

If you are an I2CS student, print "(I2CS)" next to your name. Teams that include both on-campus and I2CS students can have up to four members. Any team containing both on-campus and I2CS students automatically receives 3 points of extra credit.

Problems labeled Ƴ are likely to require techniques from next week's lectures on cuts, flows, and matchings. See also Chapter 7 in Kleinberg and Tardos, or Chapter 26 in CLRS.

Ƴ 1. Suppose you are asked to construct the minimum spanning tree of a graph G, but you are not completely sure of the edge weights. Specifically, you have a *conjectured* weight $\tilde{w}(e)$ for every edge e in the graph, but you also know that up to k of these conjectured weights are wrong. With the exception of one edge e whose true weight you know exactly, you don't know which edges are wrong, or even how they're wrong; the true weights of those edges could be larger or smaller than the conjectured weights. Given this unreliable information, it is of course impossible to reliably construct the true minimum spanning tree of G, but it is still possible to say something about your special edge.

Describe and analyze an efficient algorithm to determine whether a specific edge e, whose *actual* weight is known, is *definitely not* in the minimum spanning tree of G under the stated conditions. The input consists of the graph G, the conjectured weight function $\tilde{w} : E(G) \to \mathbb{R}$, the positive integer k, and the edge e.

2. Most classical minimum-spanning-tree algorithms use the notions of 'safe' and 'useless' edges described in the lecture notes, but there is an alternate formulation. Let G be a weighted undirected graph, where the edge weights are distinct. We say that an edge e is *dangerous* if it is the longest edge in some cycle in G, and *useful* if it does not lie in any cycle in G.

   (a) Prove that the minimum spanning tree of G contains every useful edge.

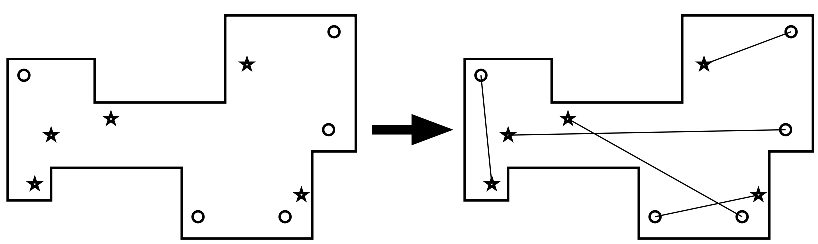
   (b) Prove that the minimum spanning tree of G does not contain any dangerous edge.

   (c) Describe and analyze an efficient implementation of the "anti-Kruskal" MST algorithm: Examine the edges of G in *decreasing* order; if an edge is dangerous, remove it from G. *[Hint: It won't be as fast as the algorithms you saw in class.]*

♈ 3. The UIUC Computer Science department has decided to build a mini-golf course in the basement of the Siebel Center! The playing field is a closed polygon bounded by m horizontal and vertical line segments, meeting at right angles. The course has n *starting points* and n *holes*, in one-to-one correspondence. It is always possible hit the ball along a straight line directly from each starting point to the corresponding hole, without touching the boundary of the playing field. (Players are not allowed to bounce golf balls off the walls; too much glass.) The n starting points and n holes are all at distinct locations.

   Sadly, the architect's computer crashed just as construction was about to begin. Thanks to the herculean efforts of their sysadmins, they were able to recover the *locations* of the starting points and the holes, but all information about which starting points correspond to which holes was lost!

   Describe and analyze an algorithm to compute a one-to-one correspondence between the starting points and the holes that meets the straight-line requirement, or to report that no such correspondence exists. The input consists of the x- and y-coordinates of the m corners of the playing field, the n starting points, and the n holes. Assume you can determine in constant time whether two line segments intersect, given the x- and y-coordinates of their endpoints.



A minigolf course with five starting points ($\star$) and five holes ($\circ$), and a legal correspondence between them.

♈ 4. Let $G = (V, E)$ be a directed graph where the in-degree of each vertex is equal to its out-degree. Prove or disprove the following claim: For any two vertices u and v in G, the number of mutually edge-disjoint paths from u to v is equal to the number of mutually edge-disjoint paths from v to u.

5. You are given a set of n boxes, each specified by its height, width, and depth. The order of the dimensions is unimportant; for example, a $1 \times 2 \times 3$ box is exactly the same as a $3 \times 1 \times 2$ box of a $2 \times 1 \times 3$ box. You can nest box A inside box B if and only if A can be rotated so that it has strictly smaller height, strictly smaller width, and strictly smaller depth than B.

   (a) Design and analyze an efficient algorithm to determine the largest sequence of boxes that can be nested inside one another. *[Hint: Model the nesting relationship as a graph.]*

   ♈ (b) Describe and analyze an efficient algorithm to nest all n boxes into as few groups as possible, where each group consists of a nested sequence. You are not allowed to put two boxes side-by-side inside a third box, even if they are small enough to fit.[1] *[Hint: Model the nesting relationship as a **different** graph.]*

6. *[Extra credit]* Prove that Ford's generic shortest-path algorithm (described in the lecture notes) can take exponential time in the worst case when implemented with a stack instead of a heap (like Dijkstra) or a queue (like Bellman-Ford). Specifically, construct for every positive integer n a weighted directed n-vertex graph G_n, such that the stack-based shortest-path algorithm call RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: Towers of Hanoi.]*

[1] Without this restriction, the problem is NP-hard, even for one-dimensional "boxes".

CS 473G: Combinatorial Algorithms, Fall 2005
Homework 6

Practice only; nothing to turn in.

---

1. A small airline, Ivy Air, flies between three cities: Ithaca (a small town in upstate New York), Newark (an eyesore in beautiful New Jersey), and Boston (a yuppie town in Massachusetts). They offer several flights but, for this problem, let us focus on the Friday afternoon flight that departs from Ithaca, stops in Newark, and continues to Boston. There are three types of passengers:

   (a) Those traveling from Ithaca to Newark (god only knows why).
   (b) Those traveling from Newark to Boston (a very good idea).
   (c) Those traveling from Ithaca to Boston (it depends on who you know).

   The aircraft is a small commuter plane that seats 30 passengers. The airline offers three fare classes:

   (a) Y class: full coach.
   (b) B class: nonrefundable.
   (c) M class: nonrefundable, 3-week advanced purchase.

   Ticket prices, which are largely determined by external influences (i.e., competitors), have been set and advertised as follows:

	Ithaca-Newark	Newark-Boston	Ithaca-Boston
Y	300	160	360
B	220	130	280
M	100	80	140

   Based on past experience, demand forecasters at Ivy Air have determined the following upper bounds on the number of potential customers in each of the 9 possible origin-destination/fare-class combinations:

	Ithaca-Newark	Newark-Boston	Ithaca-Boston
Y	4	8	3
B	8	13	10
M	22	20	18

   The goal is to decide how many tickets from each of the 9 origin/destination/fare-class combinations to sell. The constraints are that the place cannot be overbooked on either the two legs of the flight and that the number of tickets made available cannot exceed the forecasted maximum demand. The objective is to maximize the revenue.

   Formulate this problem as a linear programming problem.

2. (a) Suppose we are given a directed graph $G = (V, E)$, a length function $\ell : E \to \mathbb{R}$, and a source vertex $s \in V$. Write a linear program to compute the shortest-path distance from s to every other vertex in V. *[Hint: Define a variable for each vertex representing its distance from s. What objective function should you use?]*

   (b) In the *minimum-cost multicommodity-flow* problem, we are given a directed graph $G = (V, E)$, in which each edge $u \to v$ has an associated nonnegative *capacity* $c(u \to v) \geq 0$ and an associated *cost* $\alpha(u \to v)$. We are given k different commodities, each specified by a triple $K_i = (s_i, t_i, d_i)$, where s_i is the source node of the commodity, t_i is the target node for the commodity i, and d_i is the *demand*: the desired flow of commodity i from s_i to t_i. A *flow* for commodity i is a non-negative function $f_i : E \to \mathbb{R}_{\geq 0}$ such that the total flow into any vertex other than s_i or t_i is equal to the total flow out of that vertex. The *aggregate flow* $F : E \to \mathbb{R}$ is defined as the sum of these individual flows: $F(u \to v) = \sum_{i=1}^{k} f_i(u \to v)$. The aggregate flow $F(u \to v)$ on any edge must not exceed the capacity $c(u \to v)$. The goal is to find an aggregate flow whose total *cost* $\sum_{u \to v} F(u \to v) \cdot \alpha(u \to v)$ is as small as possible. (Costs may be negative!) Express this problem as a linear program.

3. In class we described the duality transformation only for linear programs in canonical form:

Primal (Π)		**Dual (∐)**
max $c \cdot x$ s.t. $Ax \leq b$ $x \geq 0$	$\Longleftrightarrow$	min $y \cdot b$ s.t. $yA \geq c$ $y \geq 0$

Describe precisely how to dualize the following more general linear programming problem:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{d} c_j x_j \\
\text{subject to} \quad & \sum_{j=1}^{d} a_{ij} x_j \leq b_i \quad \text{for each } i = 1 \,..\, p \\
& \sum_{j=1}^{d} a_{ij} x_j = b_i \quad \text{for each } i = p+1 \,..\, p+q \\
& \sum_{j=1}^{d} a_{ij} x_j \geq b_i \quad \text{for each } i = p+q+1 \,..\, n
\end{aligned}
$$

Your dual problem should have one variable for each primal constraint, and the dual of your dual program should be precisely the original linear program.

4. (a) Model the maximum-cardinality bipartite matching problem as a linear programming problem. The input is a bipartite graph $G = (U, V; E)$, where $E \subseteq U \times V$; the output is the largest matching in G. Your linear program should have one variable for every edge.

   (b) Now dualize the linear program from part (a). What do the dual variables represent? What does the objective function represent? What problem is this!?

5. An *integer program* is a linear program with the additional constraint that the variables must take only integer values.

   (a) Prove that deciding whether an integer program has a feasible solution is NP-complete.

   (b) Prove that finding the optimal feasible solution to an integer program is NP-hard.

   [Hint: Almost any NP=hard decision problem can be rephrased as an integer program. Pick your favorite.]

6. Consider the LP formulation of the shortest path problem presented in class:

$$\begin{array}{lll} \text{maximize} & d_t & \\ \text{subject to} & d_s = 0 & \\ & d_v - d_u \le \ell_{u\to v} & \text{for every edge } u \to v \end{array}$$

   Characterize the feasible bases for this linear program in terms of the original weighted graph. What does a simplex pivoting operation represent? What is a locally optimal (*i.e.*, dual feasible) basis? What does a dual pivoting operation represent?

7. Consider the LP formulation of the maximum-flow problem presented in class:

$$\begin{array}{lll} \text{maximize} & \sum_w f_{s\to w} - \sum_u f_{u\to s} & \\ \text{subject to} & \sum_w f_{v\to w} - \sum_u f_{u\to v} = 0 & \text{for every vertex } v \ne s,t \\ & f_{u\to v} \le c_{u\to v} & \text{for every edge } u \to v \\ & f_{u\to v} \ge 0 & \text{for every edge } u \to v \end{array}$$

   Is the Ford-Fulkerson augmenting path algorithm an instance of the simplex algorithm applied to this linear program? Why or why not?

*8. *Helly's theorem* says that for any collection of convex bodies in $\mathbb{R}^n$, if every $n+1$ of them intersect, then there is a point lying in the intersection of all of them. Prove Helly's theorem for the special case that the convex bodies are halfspaces. *[Hint: Show that if a system of inequalities $Ax \ge b$ does not have a solution, then we can select $n+1$ of the inequalities such that the resulting system does not have a solution. Construct a primal LP from the system by choosing a 0 cost vector.]*

You have 90 minutes to answer four of these questions.

Write your answers in the separate answer booklet.

You may take the question sheet with you when you leave.

1. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

   Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

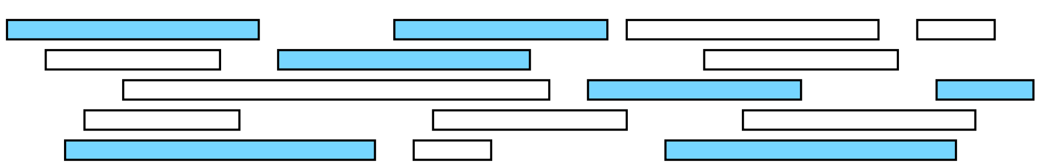
   (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.

   (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

2. Suppose you are given a magical black box that can tell you in constant time whether or not a given graph has a Hamiltonian cycle. Using this magic black box as a subroutine, describe and analyze a *polynomial-time* algorithm to actually compute a Hamiltonian cycle in a given graph, if one exists.

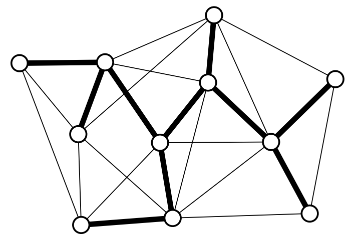
3. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The *size* of a tiling cover is just the number of intervals.

   Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.



A set of intervals. The seven shaded intervals form a tiling path.

4. Prove that the following problem is NP-complete: Given an undirected graph, does it have a spanning tree in which every node has degree at most 3?

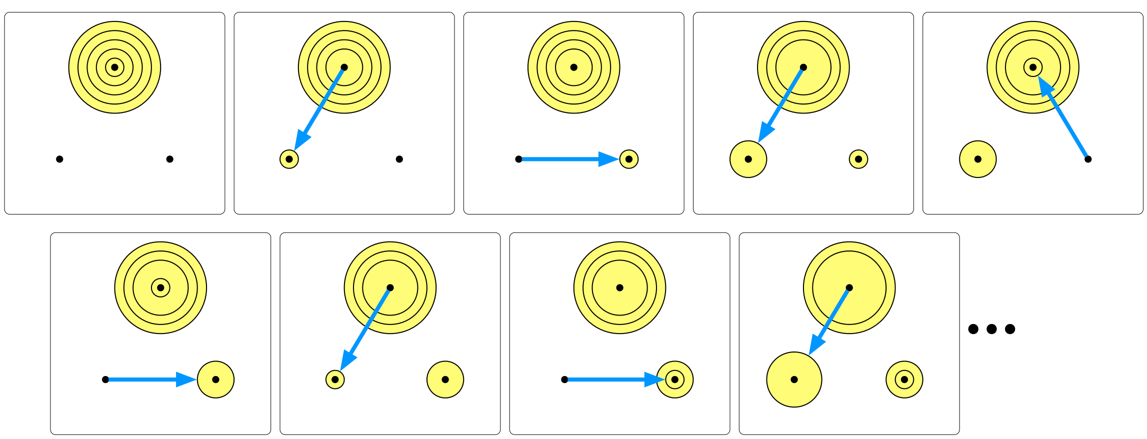


A graph with a spanning tree of maximum degree 3.

5. The *Tower of Hanoi* puzzle, invented by Edouard Lucas in 1883, consists of three pegs and n disks of different sizes. Initially, all n disks are on the same peg, stacked in order by size, with the largest disk on the bottom and the smallest disk on top. In a single move, you can move the topmost disk on any peg to another peg; however, you are never allowed to place a larger disk on top of a smaller one. Your goal is to move all n disks to a different peg.

   (a) Prove that the Tower of Hanoi puzzle can be solved in exactly $2^n - 1$ moves. *[Hint: You've probably seen this before.]*

   (b) Now suppose the pegs are arranged in a circle and you are *only* allowed to move disks *counterclockwise*. How many moves do you need to solve this restricted version of the puzzle? Give a upper bound in the form $O(f(n))$ for some function $f(n)$. Prove your upper bound is correct.



A top view of the first eight moves in a counterclockwise Towers of Hanoi solution

You have 90 minutes to answer four of these questions.

Write your answers in the separate answer booklet.

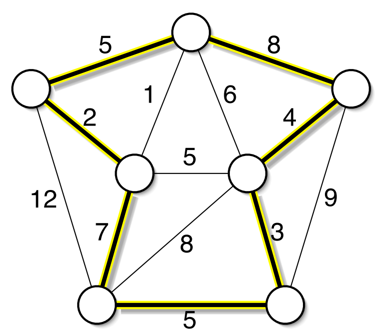
You may take the question sheet with you when you leave.

Chernoff Bounds: If X is the sum of independent indicator variables and $\mu = \mathrm{E}[X]$, then the following inequalities hold for any $\delta > 0$:

$$\Pr[X < (1-\delta)\mu] < \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} \qquad \Pr[X > (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$$

1. Describe and analyze an algorithm that randomly shuffles an array $X[1..n]$, so that each of the $n!$ possible permutations is equally likely, in $O(n)$ time. (Assume that the subroutine RANDOM(m) returns an integer chosen uniformly at random from the set $\{1, 2, , \ldots, m\}$ in $O(1)$ time.)

2. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-complete.



A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

3. A sequence of numbers $\langle a_1, a_2, a_3, \ldots a_n\rangle$ is *oscillating* if $a_i < a_{i+1}$ for every *odd* index i and $a_i > a_{i+1}$ for every *even* index i. Describe and analyze an efficient algorithm to compute the longest oscillating subsequence in a sequence of n integers.

4. This problem asks you to how to efficiently modify a maximum flow if one of the edge capacities changes. Specifically, you are given a directed graph $G = (V, E)$ with capacities $c : E \to \mathbb{Z}_+$, and a maximum flow $F : E \to \mathbb{Z}$ from some vertex s to some other vertex t in G. Describe and analyze efficient algorithms for the following operations:

   (a) INCREMENT(e) — Increase the capacity of edge e by 1 and update the maximum flow F.

   (b) DECREMENT(e) — Decrease the capacity of edge e by 1 and update the maximum flow F.

   Both of your algorithms should be significantly faster than recomputing the maximum flow from scratch.

5.

6. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges.

   (a) Prove that it is NP-hard to compute the most interesting 3-coloring of a graph. *[Hint: There is a one-line proof. Use one of the NP-hard problems described in class.]*

   (b) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G. Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$. *[Hint: There is a one-line proof.]*

   (c) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G. Suppose we assign each vertex in G a *random* color from the set $\{\text{red}, \text{green}, \text{blue}\}$. Prove that the expected number of interesting edges is at least $\frac{2}{3}wow(G)$.

7.

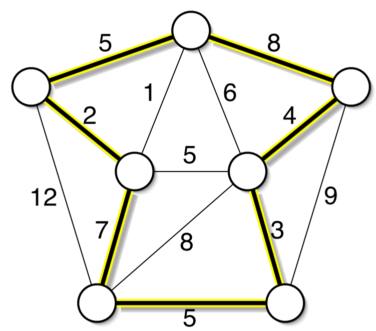
You have 180 minutes to answer six of these questions.

Write your answers in the separate answer booklet.

You may take the question sheet with you when you leave.

1. Describe and analyze an algorithm that randomly shuffles an array $X[1..n]$, so that each of the $n!$ possible permutations is equally likely, in $O(n)$ time. (Assume that the subroutine $\text{Random}(m)$ returns an integer chosen uniformly at random from the set $\{1, 2, , \ldots, m\}$ in $O(1)$ time.)

2. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-complete.



A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

3. Suppose you are given a directed graph $G = (V, E)$ with capacities $c : E \to \mathbb{Z}_+$ *and* a maximum flow $F : E \to \mathbb{Z}$ from some vertex s to some other vertex t in G. Describe and analyze efficient algorithms for the following operations:

   (a) $\text{Increment}(e)$ — Increase the capacity of edge e by 1 and update the maximum flow F.

   (b) $\text{Decrement}(e)$ — Decrease the capacity of edge e by 1 and update the maximum flow F.

   Both of your algorithms should be significantly faster than recomputing the maximum flow from scratch.

4. Suppose you are given an undirected graph G and two vertices s and t in G. Two paths from s to t are *vertex-disjoint* if the only vertices they have in common are s and t. Describe and analyze an efficient algorithm to compute the maximum number of vertex-disjoint paths between s and t in G. *[Hint: Reduce this to a more familiar problem on a suitable directed graph G'.]*

5. A sequence of numbers $\langle a_1, a_2, a_3, \ldots a_n \rangle$ is *oscillating* if $a_i < a_{i+1}$ for every *odd* index i and $a_i > a_{i+1}$ for every *even* index i. For example, the sequence $\langle 2, 7, 1, 8, 2, 8, 1, 8, 3 \rangle$ is oscillating. Describe and analyze an efficient algorithm to compute the longest oscillating subsequence in a sequence of n integers.

6. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem we saw in class is a special case.

   (a) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G. Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$.

   (b) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G. Suppose we assign each vertex in G a *random* color from the set $\{\text{red}, \text{green}, \text{blue}\}$. Prove that the expected number of interesting edges is at least $\frac{2}{3} wow(G)$.

7. It's time for the 3rd Quasi-Annual Champaign-Urbana Ice Motorcycle Demolition Derby Race-O-Rama and Spaghetti Bake-Off! The main event is a competition between two teams of n motorcycles in a huge square ice-covered arena. All of the motorcycles have spiked tires so that they can ride on the ice. Each motorcycle drags a long metal chain behind it. Whenever a motorcycle runs over a chain, the chain gets caught in the tire spikes, and the motorcycle crashes. Two motorcycles can also crash by running directly into each other. All the motorcycle start simultaneously. Each motorcycle travels in a straight line at a constant speed until it either crashes or reaches the opposite wall—no turning, no braking, no speeding up, no slowing down. The Vicious Abscissas start at the south wall of the arena and ride directly north (vertically). Hell's Ordinates start at the west wall of the arena and ride directly east (horizontally). If any motorcycle completely crosses the arena, that rider's entire team wins the competition.

   Describe and analyze an efficient algorithm to decide which team will win, given the starting position and speed of each motorcycle.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 0

Due Friday, September 1, 2006 at noon in 3229 Siebel Center

Name:	
Net ID:	Alias:

☐ I understand the Homework Instructions and FAQ.

- Neatly print your full name, your NetID, and an alias of your choice in the boxes above, and submit this page with your solutions. We will list homework and exam grades on the course web site by alias. For privacy reasons, your alias should not resemble your name, your NetID, your university ID number, or (God forbid) your Social Security number. Please use the same alias for every homework and exam.

  Federal law forbids us from publishing your grades, even anonymously, without your explicit permission. **By providing an alias, you grant us permission to list your grades on the course web site; if you do not provide an alias, your grades will not be listed.**

- Please carefully read the Homework Instructions and FAQ on the course web page, and then check the box above. This page describes what we expect in your homework solutions—start each numbered problem on a new sheet of paper, write your name and NetID on every page, don't turn in source code, analyze and prove everything, use good English and good logic, and so on—as well as policies on grading standards, regrading, and plagiarism. **See especially the policies regarding the magic phrases "I don't know" and "and so on".** If you have *any* questions, post them to the course newsgroup or ask in lecture.

- This homework tests your familiarity with prerequisite material—basic data structures, big-Oh notation, recurrences, discrete probability, and most importantly, induction—to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Each numbered problem is worth 10 points; not all subproblems have equal weight.

#	1	2	3	4	5	6^*	Total
Score							
Grader							

Please put your answers to problems 1 and 2 on the same page.

1. Sort the functions listed below from asymptotically smallest to asymptotically largest, indicating ties if there are any. **Do not turn in proofs**, but you should probably do them anyway, just for practice.

   To simplify your answers, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

   $$\begin{array}{cccccccc}
   \lg n & \ln n & \sqrt{n} & n & n \lg n & n^2 & 2^n & n^{1/n} \\
   n^{1+1/\lg n} & \lg^{1000} n & 2^{\sqrt{\lg n}} & (\sqrt{2})^{\lg n} & \lg^{\sqrt{2}} n & n^{\sqrt{2}} & (1+\frac{1}{n})^n & n^{1/1000} \\
   H_n & H_{\sqrt{n}} & 2^{H_n} & H_{2^n} & F_n & F_{n/2} & \lg F_n & F_{\lg n}
   \end{array}$$

   In case you've forgotten:

   - $\lg n = \log_2 n \neq \ln n = \log_e n$
   - $\lg^3 n = (\lg n)^3 \neq \lg \lg \lg n$.
   - The harmonic numbers: $H_n = \sum_{i=1}^{n} 1/i \approx \ln n + 0.577215\ldots$
   - The Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$

2. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Proofs are *not* required; just give us the list of answers. **Don't turn in proofs**, but you should do them anyway, just for practice. Assume reasonable but nontrivial base cases. **If your solution requires specific base cases, state them.** Extra credit will be awarded for more exact solutions.

   (a) $A(n) = 2A(n/4) + \sqrt{n}$

   (b) $B(n) = 3B(n/3) + n/\lg n$

   (c) $C(n) = \dfrac{2C(n-1)}{C(n-2)}$ *[Hint: This is easy!]*

   (d) $D(n) = D(n-1) + 1/n$

   (e) $E(n) = E(n/2) + D(n)$

   (f) $F(n) = 4F\left(\left\lceil \dfrac{n-8}{2} \right\rceil + \left\lfloor \dfrac{3n}{\log_\pi n} \right\rfloor\right) + 6\dbinom{n+5}{2} - 42n \lg^7 n + \sqrt{13n-6} + \dfrac{\lg \lg n + 1}{\lg n \lg \lg \lg n}$

   (g) $G(n) = 2G(n-1) - G(n-2) + n$

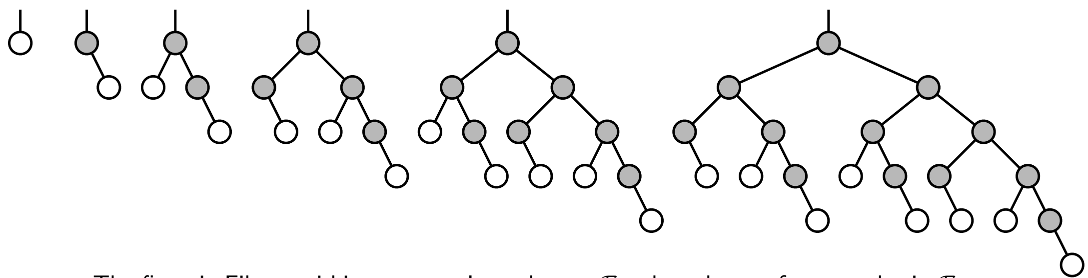
   (h) $H(n) = 2H(n/2) - 2H(n/4) + 2^n$

   (i) $I(n) = I(n/2) + I(n/4) + I(n/6) + I(n/12) + n$

   ★(j) $J(n) = \sqrt{n} \cdot J(2\sqrt{n}) + n$
   [Hint: First solve the secondary recurrence $j(n) = 1 + j(2\sqrt{n})$.]

3. The nth *Fibonacci binary tree* $\mathcal{F}_n$ is defined recursively as follows:

   - $\mathcal{F}_1$ is a single root node with no children.
   - For all $n \geq 2$, $\mathcal{F}_n$ is obtained from $\mathcal{F}_{n-1}$ by adding a right child to every leaf and adding a left child to every node that has only one child.



The first six Fibonacci binary trees. In each tree $\mathcal{F}_n$, the subtree of gray nodes is $\mathcal{F}_{n-1}$.

   (a) Prove that the number of leaves in $\mathcal{F}_n$ is precisely the nth Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$.

   (b) How many nodes does $\mathcal{F}_n$ have? For full credit, give an *exact*, closed-form answer in terms of Fibonacci numbers, and prove your answer is correct.

   (c) Prove that the left subtree of $\mathcal{F}_n$ is a copy of $\mathcal{F}_{n-2}$.

4. Describe and analyze a data structure that stores set of n records, each with a numerical *key* and a numerical *priority*, such that the following operation can be performed quickly:

   RANGETOP(a, z) : return the highest-priority record whose key is between a and z.

   For example, if the (key, priority) pairs are

$$(3,1),\ (4,9),\ (9,2),\ (6,3),\ (5,8),\ (7,5),\ (1,4),\ (0,7),$$

   then RANGETOP$(2, 8)$ would return the record with key 4 and priority 9 (the second record in the list).

   You may assume that no two records have equal keys or equal priorities, and that no record has a key equal to a or z. Analyze both the size of your data structure and the running time of your RANGETOP algorithm. For full credit, your data structure must be as small as possible *and* your RANGETOP algorithm must be as fast as possible.

   [Hint: How would you compute the number *of keys between a and z? How would you solve the problem if you knew that a is always $-\infty$?]*

5. Penn and Teller agree to play the following game. Penn shuffles a standard deck[1] of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3♣), at which point the remaining undrawn cards instantly burst into flames.

   The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.[2] To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

   (a) What is the expected number of cards that Teller draws?

   (b) What is the expected *maximum* value among the cards Teller gives to Penn?

   (c) What is the expected *minimum* value among the cards Teller gives to Penn?

   (d) What is the expected number of cards that Teller gives to Penn?

   Full credit will be given only for *exact* answers (with correct proofs, of course).

*6. **[Extra credit]**[3]

   Lazy binary is a variant of standard binary notation for representing natural numbers where we allow each "bit" to take on one of three values: 0, 1, or 2. Lazy binary notation is defined inductively as follows.

   - The lazy binary representation of zero is 0.
   - Given the lazy binary representation of any non-negative integer n, we can construct the lazy binary representation of $n + 1$ as follows:
     (a) increment the rightmost digit;
     (b) if any digit is equal to 2, replace the rightmost 2 with 0 and increment the digit immediately to its left.

   Here are the first several natural numbers in lazy binary notation:

   0, 1, 10, 11, 20, 101, 110, 111, 120, 201, 210, 1011, 1020, 1101, 1110, 1111, 1120, 1201, 1210, 2011, 2020, 2101, 2110, 10111, 10120, 10201, 10210, 11011, 11020, 11101, 11110, 11111, 11120, 11201, 11210, 12011, 12020, 12101, 12110, 20111, 20120, 20201, 20210, 21011, 21020, 21101, 21110, 101111, 101120, 101201, 101210, 102011, 102020, 102101, 102110, ...

   (a) Prove that in any lazy binary number, between any two 2s there is at least one 0, and between two 0s there is at least one 2.

   (b) Prove that for any natural number N, the sum of the digits of the lazy binary representation of N is exactly $\lfloor \lg(N + 1) \rfloor$.

[1] In a standard deck of 52 cards, each card has a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$ and a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$, and every possible suit-value pair appears in the deck exactly once. Actually, to make the game more interesting, Penn and Teller normally use razor-sharp ninja throwing cards.

[2] Specifically, he hurls them from the opposite side of the stage directly into the back of Penn's right hand.

[3] The "I don't know" rule does not apply to extra credit problems. There is no such thing as "partial extra credit".

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 1

Due Tuesday, September 12, 2006 in 3229 Siebel Center

Starting with this homework, groups of up to three students can submit or present a single joint solution. If your group is submitting a written solution, please remember to **print the names, NetIDs, and aliases of *every* group member on *every* page**. Please remember to submit **separate, individually stapled** solutions to each of the problems.

---

1. Recall from lecture that a *subsequence* of a sequence A consists of a (not necessarily contiguous) collection of elements of A, arranged in the same order as they appear in A. If B is a subsequence of A, then A is a *supersequence* of B.

   (a) Describe and analyze a **simple** recursive algorithm to compute, given two sequences A and B, the length of the *longest common subsequence* of A and B. For example, given the strings `ALGORITHM` and `ALTRUISTIC`, your algorithm would return 5, the length of the longest common subsequence `ALRIT`.

   (b) Describe and analyze a **simple** recursive algorithm to compute, given two sequences A and B, the length of a *shortest common supersequence* of A and B. For example, given the strings `ALGORITHM` and `ALTRUISTIC`, your algorithm would return 14, the length of the shortest common supersequence `ALGTORUISTHIMC`.

   (c) Let $|A|$ denote the length of sequence A. For any two sequences A and B, let $\operatorname{lcs}(A, B)$ denote the length of the longest common subsequence of A and B, and let $\operatorname{scs}(A, B)$ denote the length of the shortest common supersequence of A and B.
   Prove that $|\boldsymbol{A}|+|\boldsymbol{B}| = \mathbf{lcs}(\boldsymbol{A}, \boldsymbol{B}) + \mathbf{scs}(\boldsymbol{A}, \boldsymbol{B})$ for all sequences A and B. *[Hint: There is a simple non-inductive proof.]*

   In parts (a) and (b), we are *not* looking for the most efficient algorithms, but for algorithms with simple and correct recursive structure.

2. You are a contestant on a game show, and it is your turn to compete in the following game. You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni™, to which you are hopelessly addicted.

   (a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

   (b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

   *(c) **[Extra credit]**[1] Prove that your solution to part (b) is asymptotically optimal.

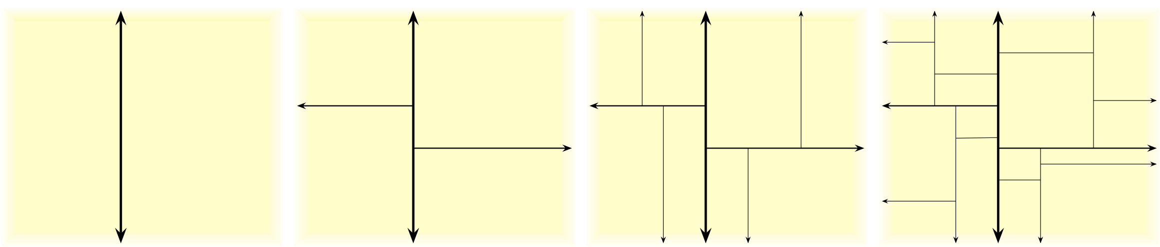
---

[1] The "I don't know" rule does not apply to extra credit problems. There is no such thing as "partial extra credit".

3. A kd-tree is a rooted binary tree with three types of nodes: horizontal, vertical, and leaf. Each vertical node has a *left* child and a *right* child; each horizontal node has a *high* child and a *low* child. The non-leaf node types alternate: non-leaf children of vertical nodes are horizontal and vice versa. Each non-leaf node v stores a real number p_v called its *pivot value*. Each node v has an associated *region* $R(v)$, defined recursively as follows:

   - $R(\mathit{root})$ is the entire plane.
   - If v is is a horizontal node, the horizontal line $y = p_v$ partitions $R(v)$ into $R(\mathit{high}(v))$ and $R(\mathit{low}(v))$ in the obvious way.
   - If v is is a vertical node, the vertical line $x = p_v$ partitions $R(v)$ into $R(\mathit{left}(v))$ and $R(\mathit{right}(v))$ in the obvious way.

   Thus, each region $R(v)$ is an axis-aligned rectangle, possibly with one or more sides at infinity. If v is a leaf, we call $R(v)$ a *leaf box*.



The first four levels of a typical kd-tree.

   Suppose T is a perfectly balanced kd-tree with n leaves (and thus with depth exactly $\lg n$).

   (a) Consider the horizontal line $y = t$, where $t \neq p_v$ for all nodes v in T. *Exactly* how many leaf boxes of T does this line intersect? *[Hint: The parity of the root node matters.]* Prove your answer is correct. A correct $\Theta(\cdot)$ bound is worth significant partial credit.

   (b) Describe and analyze an efficient algorithm to compute, given T and an arbitrary horizontal line ℓ, the number of leaf boxes of T that lie *entirely above* ℓ.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 2

Due Tuesday, September 19, 2006 in 3229 Siebel Center
Remember to turn in in separate, individually stapled solutions to each of the problems.

---

1. You are given an $m \times n$ matrix M in which each entry is a 0 or 1. A *solid block* is a rectangular subset of M in which each entry is 1. Give a correct efficent algorithm to find a solid block in M with maximum area.

$$\begin{array}{c|ccc|c} 1 & 1 & 0 & 1 & 1 \\ \hline 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 0 & 1 & 1 \end{array}$$

   An algorithm that runs in $\Theta(n^c)$ time will earn $19 - 3c$ points.

2. You are a bus driver with a soda fountain machine in the back and a bus full of very hyper students, who are drinking more soda as they ride along the highway. Your goal is to drop the students off as quickly as possible. More specifically, every minute that a student is on your bus, he drinks another ounce of soda. Your goal is to drop the students off quickly, so that in total they drink as little soda as possible.

   You know how many students will get off of the bus at each exit. Your bus begins partway along the highway (probably not at either end), and moves at a constant rate. You must drive the bus along the highway- however you may drive forward to one exit then backward to an exit in the other direction, switching as often as you like (you can stop the bus, drop off students, and turn around instantaneously).

   Give an efficient algorithm to drop the students off so that they drink as little soda as possible. The input to the algorithm should be: the bus route (a list of the exits, together with the travel time between successive exits), the number of students you will drop off at each exit, and the current location of your bus (you may assume it is at an exit).

3. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the ith word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of whitespace between any two words on the same line.

   Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j, then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^{j} P_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 3

Due Wednesday, October 4, 2006 in 3229 Siebel Center

Remember to turn in separate, individually stapled solutions to each of the problems.

---

1. Consider a perfect tree of height h, where every non-leaf node has 3 children. (Therefore, each of the 3^h leaves is at distance h from the root.) Every leaf has a boolean value associated with it - either 0 or 1. Every internal node gets the boolean value assigned to the majority of its children. Given the values assigned to the leaves, we want to find an algorithm that computes the value (0 or 1) of the root.

   It is not hard to find a (deterministic) algorithm that looks at every leaf and correctly determines the value of the root, but this takes $O(3^h)$ time. Describe and analyze a *randomized* algorithm that, on average, looks at asymptotically fewer leaves. That is, the expected number of leaves your algorithm examines should be $o(3^h)$.

2. We define a *meldable heap* to be a binary tree of elements, each of which has a priority, such that the priority of any node is less than the priority of its parent. (Note that the heap does **not** have to be balanced, and that the element with greatest priority is the root.) We also define the priority of a heap to be the priority of its root.

   The *meld* operation takes as input two (meldable) heaps and returns a single meldable heap H that contains all the elements of both input heaps. We define *meld* as follows:

   - Let H_1 be the input heap with greater priority, and H_2 the input heap with lower priority. (That is, the priority of $root(H_1)$ is greater than the priority of $root(H_2)$.) Let H_L be the left subtree of $root(H_1)$ and H_R be the right subtree of $root(H_1)$.
   - We set $root(H) = root(H_1)$.
   - We now flip a coin that comes up either "Left" or "Right" with equal probability.
     - If it comes up "Left", we set the left subtree of $root(H)$ to be H_L, and the right subtree of $root(H)$ to be $meld(H_R, H_2)$ (defined recursively).
     - If the coin comes up "Right", we set the right subtree of $root(H)$ to be H_R, and the left subtree of $root(H)$ to be $meld(H_L, H_2)$.
   - As a base case, melding any heap H_1 with an empty heap gives H_1.

   (a) Analyze the expected running time of $meld(H_a, H_b)$ if H_a is a (meldable) heap with n elements, and H_b is a (meldable) heap with m elements.

   (b) Describe how to perform each of the following operations using only melds, and give the running time of each.
   - $DeleteMax(H)$, which deletes the element with greatest priority.
   - $Insert(H, x)$, which inserts the element x into the heap H.
   - $Delete(H, x)$, which - given a pointer to element x in heap H - returns the heap with x deleted.

3. Randomized Selection. Given an (unsorted) array of n distinct elements and an integer k, SELECTION is the problem of finding the kth smallest element in the array. One easy solution is to sort the array in increasing order, and then look up the kth entry, but this takes $\Theta(n \log n)$ time. The randomized algorithm below attempts to do better, at least on average.

```
QuickSelect(Array A, n, k)
pivot ← Random(1, n)
S ← {x | x ∈ A, x < A[pivot]}
s ← |S|
L ← {x | x ∈ A, x > A[pivot]}
if (k = s + 1)
   return A[pivot]
else if (k ≤ s)
   return QuickSelect(S, s, k)
else
   return QuickSelect(L, n − (s + 1), k − (s + 1))
```

Here we assume that Random(a, b) returns an integer chosen uniformly at random from a to b (inclusive of a and b). The pivot position is randomly chosen; S is the set of elements smaller than the pivot element, and L the set of elements larger than the pivot. The sets S and L are found by comparing every other element of A to the pivot. We partition the elements into these two 'halves', and recurse on the appropriate half.

(a) Write a recurrence relation for the expected running time of QuickSelect.

(b) Given any two elements $x, y \in A$, what is the probability that x and y will be compared?

(c) Either from part (a) or part (b), find the expected running time of QuickSelect.

4. **[Extra Credit]**: In the previous problem, we found a $\Theta(n)$ algorithm for selecting the kth smallest element, but the constant hidden in the $\Theta(\cdot)$ notation is somewhat large. It is easy to find the *smallest* element using at most n comparisons; we would like to be able to extend this to larger k. Can you find a randomized algorithm that uses $n + \Theta(k \log k \log n)$[1] expected comparisons? (Note that there is no constant multiplying the n.)

   Hint: While scanning through a random permutation of n elements, how many times does the smallest element seen so far change? (See HBS 0.) How many times does the kth smallest element so far change?

[1] There is an algorithm that uses $n + \Theta(k \log(n/k))$ comparisons, but this is even harder.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 4

Due Tuesday, October 10, 2006 in 3229 Siebel Center

Remember to submit **separate, individually stapled** solutions to each of the problems.

---

1. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array $A[1..n]$ that stores the height of n buildings on a city block, indexed from west to east. Building i has a good view of Lake Michigan if every building to the east of i is shorter than i. We present an algorithm that computes which buildings have a good view of Lake Michigan. Use the taxation method of amortized analysis to bound the amortized time spent in each iteration of the for loop. What is the total runtime?

```
GOODVIEW(A[1..n]):
  Initialize a stack S
  for i = 1 to n
      while (S not empty and A[i] ≥ A[S.top])
          POP(S)
      PUSH(S, i)
  return S
```

2. Design and analyze a simple data structure that maintains a list of integers and supports the following operations.

   (a) CREATE(): creates and returns a new list L

   (b) PUSH(L, x): appends x to the end of L

   (c) POP(L): deletes the last entry of L and returns it

   (d) LOOKUP(L, k): returns the kth entry of L

   Your solution may use these primitive data structures: arrays, balanced binary search trees, heaps, queues, single or doubly linked lists, and stacks. If your algorithm uses anything fancier, you must give an explicit implementation. Your data structure should support all operations in amortized constant time. In addition, your data structure should support LOOKUP() in worst-case $O(1)$ time. At all times, your data structure should use space which is linear in the number of objects it stores.

3. Consider a computer game in which players must navigate through a field of landmines, which are represented as points in the plane. The computer creates new landmines which the players must avoid. A player may ask the computer how many landmines are contained in any simple polygonal region; it is your job to design an algorithm which answers these questions efficiently.

   You have access to an efficient static data structure which supports the following operations.

- CREATES($\{p_1, p_2, \ldots, p_n\}$): creates a new data structure S containing the points $\{p_1, \ldots, p_n\}$. It has a worst-case running time of $T(n)$. Assume that $T(n)/n \geq T(n-1)/(n-1)$, so that the average processing time of elements does not decrease as n grows.
- DUMPS(S): destroys S and returns the set of points that S stored. It has a worst-case running time of $O(n)$, where n is the number of points in S.
- QUERYS(S, R): returns the number of points in S that are contained in the region R. It has a worst-case running time of $Q(n)$, where n is the number of points stored in S.

Unfortunately, the data structure does not support point insertion, which is required in your application. Using the given static data structure, design and analyze a dynamic data structure that supports the following operations.

(a) CREATED(): creates a new data structure D containing no points. It should have a worst-case constant running time.

(b) INSERTD(D, p): inserts p into D. It should run in amortized $O(\log n) \cdot T(n)/n$ time.

(c) QUERYD(D, R): returns the number of points in D that are contained in the region R. It should have a worst-case running time of $O(\log n) \cdot Q(n)$.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 5

Due Tuesday, October 24, 2006 in 3229 Siebel Center
Remember to turn in in separate, individually stapled solutions to each of the problems.

---

1. Makefiles:
   In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called 'make' that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of the other files that must be compiled. Thus a source file must be recompiled if a file on which it depends is changed.

   Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design and analyze an efficient algorithm to recompile only the necessary files. DO NOT worry about the details of parsing a Makefile.

2. Consider a graph G, with n vertices. Show that if any two of the following properties hold for G, then the third property must also hold.

   - G is connected.
   - G is acyclic.
   - G has $n-1$ edges.

3. The weight of a spanning tree is the sum of the weights on the edges of the tree. Given a graph, G, describe an efficient algorithm (the most efficient one you can) to find the k lightest (with least weight) spanning trees of G.

   Analyze the running time of your algorithm. Be sure to prove your algorithm is correct.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 6

Due Wednesday, November 8, 2006 in 3229 Siebel Center

Remember to turn in separate, individually stapled solutions to each of the problems.

---

1. Dijkstra's algorithm can be used to determine shortest paths on graphs with some negative edge weights (as long as there are no negative cycles), but the worst-case running time is much worse than the $O(E+V \log V)$ it takes when the edge weights are all positive. Construct an infinite family of graphs - with negative edge weights - for which the asymptotic running time of Dijkstra's algorithm is $\Omega(2^{|V|})$.

2. It's a cold and rainy night, and you have to get home from Siebel Center. Your car has broken down, and it's too windy to walk, which means you have to take a bus. To make matters worse, there is no bus that goes directly from Siebel Center to your apartment, so you have to change buses some number of times on your way home. Since it's cold outside, you want to spend as little time as possible waiting in bus shelters.

   From a computer in Siebel Center, you can access an online copy of the MTD bus schedule, which lists bus routes and the arrival time of every bus at each stop on its route. Describe an algorithm which, given the schedule, finds a way for you to get home that minimizes the time you spend at bus shelters (the amount of time you spend on the bus doesn't matter). Since Siebel Center is warm and the nearest bus stop is right outside, you can assume that you wait inside Siebel until the first bus you want to take arrives outside. Analyze the efficiency of your algorithm and prove that it is correct.

3. The Floyd-Warshall all-pairs shortest path algorithm computes, for each $u, v \in V$, the shortest path from u to v. However, if the graph has negative cycles, the algorithm fails. Describe a modified version of the algorithm (*with the same asymptotic time complexity*) that correctly returns shortest-path distances, even if the graph contains negative cycles. That is, if there is a path from u to some negative cycle, and a path from that cycle to v, the algorithm should output $dist(u, v) = -\infty$. For any other pair u, v, the algorithm should output the length of the shortest directed path from u to v.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 6

Due at **4 p.m. on** Friday, November 17, 2006 in 3229 Siebel Center

---

Remember to turn in separate, individually stapled solutions to each of the problems.

---

1. Given an undirected graph $G(V, E)$, with three vertices $u, v, w \in V$, you want to know whether there exists a path from u to w via v. (That is, the path from u to w must use v as an intermediate vertex.) Describe an efficient algorithm to solve this problem.

2. *Ad-hoc Networks*, made up of cheap, low-powered wireless devices, are often used on battlefields, in regions that have recently suffered from natural disasters, and in other situtations where people might want to monitor conditions in hard-to-reach areas. The idea is that a large collection of the wireless devices could be dropped into the area from an airplane (for instance), and then they could be configured into an efficiently functioning network.

   Since the devices are cheap and low-powered, they frequently fail, and we would like our networks to be reliable. If a device detects that it is likely to fail, it should transmit the information it has to some other device (called a *backup*) within range of it. The range is limited; we assume that there is a distance d such that two devices can communicate if and only if they are within distance d of each other. To improve reliability, we don't want a device to transmit information to a neighbor that has already failed, and so we require each device v to have at least k backup devices that it could potentially contact, all of which must be within d meters of it. We call this the *backup set* of v. Also, we do not want any device to be in the backup set of too many other devices; if it were, and it failed, a large fraction of our network would be affected.

   The input to our problem is a collection of n devices, and for each pair u, v of devices, the distance between u and v. We are also given the distance d that determines the range of a device, and parameters b and k. Describe an algorithm that determines if, for each device, we can find a backup set of size k, while also requiring that no device appears in the backup set of more than b other devices.

3. **UPDATED:** Given a piece of text T and a pattern P (the 'search string'), an algorithm for the string-matching problem either finds the first occurrence of P in T, or reports that there is none. Modify the Knuth-Morris-Pratt (KMP) algorithm so that it solves the string-matching problem, even if the pattern contains the wildcards '?' and '*'. Here, '?' represents any *single* character of the text, and '*' represents any substring of the text (including the empty substring). For example, the pattern "A?B*?A" matches the text "ABACBCABBCCACBA" starting in position 3 (in three different ways), and position 7 (in two ways). For this input, your algorithm would need to return '3'.

   UPDATE: You may assume that the pattern you are trying to match containst at most 3 blocks of question marks; the usage of '*' wildcards is stll unrestricted. Here, a block refers to a string of consecutive '?'s in the pattern. For example, AAB??ACA???????BB contains 2 blocks of question marks; A?B?C?A?C contains 4 blocks of question marks.

4. In the two-dimensional pattern-matching problem, you are given an $m \times n$ matrix M and a $p \times q$ pattern P. You wish to find all positions (i, j) in M such that the the submatrix of M between rows i and $i + p - 1$ and between columns j and $j + q - 1$ is identical to P. (That is, the $p \times q$ sub-matrix of M below and to the right of position (i, j) should be identical to P.) Describe and analyze an efficient algorithm to solve this problem.[1]

[1]Note that the normal string-matching problem is the special case of the 2-dimensional problem where $m = p = 1$.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 8

Due Wednesday, December 6, 2006 in 3229 Siebel Center

Remember to submit **separate, individually stapled** solutions to each of the problems.

---

1. Given an array $A[1..n]$ of $n \geq 2$ distinct integers, we wish to find the second largest element using as few comparisons as possible.

   (a) Give an algorithm which finds the second largest element and uses at most $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

   *(b) Prove that every algorithm which finds the second largest element uses at least $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

2. Let R be a set of rectangles in the plane. For each point p in the plane, we say that the *rectangle depth* of p is the number of rectangles in R that contain p.

   (a) (Step 1: Algorithm Design) Design and analyze a polynomial-time algorithm which, given R, computes the maximum rectangle depth.

   (b) (Step 2: ???) Describe and analyze a polynomial-time reduction from the maximum rectangle depth problem to the maximum clique problem.

   (c) (Step 3: Profit!) In 2000, the Clay Mathematics Institute described the Millennium Problems: seven challenging open problems which are central to ongoing mathematical research. The Clay Institute established seven prizes, each worth one million dollars, to be awarded to anyone who solves a Millennium problem. One of these problems is the P = NP question. In (a), we developed a polynomial-time algorithm for the maximum rectangle depth problem. In (b), we found a reduction from this problem to an NP-complete problem. We know from class that if we find a polynomial-time algorithm for any NP-complete problem, then we have shown P = NP. Why hasn't Jeff used (a) and (b) to show P = NP and become a millionaire?

3. Let G be a complete graph with integer edge weights. If C is a cycle in G, we say that the *cost* of C is the sum of the weights of edges in C. Given G, the traveling salesman problem (TSP) asks us to compute a Hamiltonian cycle of minimum cost. Given G, the traveling salesman cost problem (TSCP) asks us to compute the cost of a minimum cost Hamiltonian cycle. Given G and an integer k, the traveling salesman decision problem (TSDP) asks us to decide if there is a Hamiltonian cycle in G of cost at most k.

   (a) Describe and analyze a polynomial-time reduction from TSP to TSCP.

   (b) Describe and analyze a polynomial-time reduction from TSCP to TSDP.

   (c) Describe and analyze a polynomial-time reduction from TSDP to TSP.

(d) What can you conclude about the relative computational difficulty of TSP, TSCP, and TSDP?

4. Let G be a graph. A set S of vertices of G is a *dominating set* if every vertex in G is either in S or adjacent to a vertex in S. Show that, given G and an integer k, deciding if G contains a dominating set of size at most k is NP-complete.

1. Probability

   (a) n people have checked their hats with a hat clerk. The clerk is somewhat absent-minded and returns the hats uniformly at random (with no regard for whether each hat is returned to its owner). On average, how many people will get back their own hats?

   (b) Let S be a uniformly random permutation of $\{1, 2, \ldots, n-1, n\}$. As we move from the left to the right of the permutation, let X denote the smallest number seen so far. On average, how many different values will X take?

2. A *tournament* is a directed graph where each pair of distinct vertices u, v has either the edge uv or the edge vu (but not both). A *Hamiltonian path* is a (directed) path that visits each vertex of the (di)graph. Prove that every tournament has a Hamiltonian path.

3. Describe and analyze a data structure that stores a set of n records, each with a numerical *key*, such that the following operation can be performed quickly:

   `Foo`(a): return the sum of the records with keys at least as large as a.

   For example, if the keys are:

   $$3\ 4\ 9\ 6\ 5\ 8\ 7\ 1\ 0$$

   then `Foo`(2) would return 42, since $3, 4, 5, 6, 7, 8, 9$ are all larger than 2 and $3+4+5+6+7+8+9 = 42$.

   You may assume that no two records have equal keys, and that no record has a key equal to a. Analyze both the size of your data structure and the running time of your `Foo` algorithm. Your data structure must be as small as possible and your `Foo` algorithm must be as fast as possible.

1. The Acme Company is planning a company party. In planning the party, each employee is assigned a *fun value* (a positive real number). The goal of the party planners is to maximize the total fun value (sum of the individual fun values) of the employees invited to the party. However, the planners are not allowed to invite both an employee and his direct boss. Given a tree containing the boss/underling structure of Acme, find the invitation list with the highest allowable fun value.

2. An *inversion* in an array A is a pair i, j such that $i < j$ and $A[i] > A[j]$. (In an n-element array, the number of inversions is between 0 and $\binom{n}{2}$.)

   Find an efficient algorithm to count the number of inversions in an n-element array.

3. A *tromino* is a geometric shape made from three squares joined along complete edges. There are only two possible trominoes: the three component squares may be joined in a line or an L-shape.

   (a) Show that it is possible to cover all but one square of a 64 x 64 checkerboard using L-shape trominoes. (In your covering, each tromino should cover three squares and no square should be covered more than once.)

   (b) Show that you can leave *any* single square uncovered.

   (c) Can you cover all but one square of a 64 x 64 checkerboard using *line* trominoes? If so, which squares can you leave uncovered?

1. **Moving on a Checkerboard**

   Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:

   1) the square immediately above
   2) the square that is one up and one to the left (but only if the checker is not already in the leftmost column)
   3) the square that is one up and one to the right (but only if the checker is not already in the rightmost column)

   Each time you move from square x to square y, you receive $p(x, y)$ dollars. You are given a list of the values $p(x, y)$ for each pair (x, y) for which a move from x to y is legal. Do not assume that $p(x, y)$ is positive.

   Give an algorithm that figures out the set of moves that will move the checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible. You algorithm is free to pick any square along the bottom edge as a starting point and any square along the top edge as a destination in order to maximize the number of dollars gathered along the way. What is the running time of your algorithm?

2. **Maximizing Profit**

   You are given lists of values $h_1, h_2, \ldots, h_k$ and $l_1, l_2, \ldots, l_k$. For each i you can choose $j_i = h_i$, $j_i = l_i$, or $j_i = 0$; the only catch is that if $j_i = h_i$ then j_{i-1} must be 0 (except for $i = 1$). Your goal is to maximize $\sum_{i=1}^{k} j_i$.

   Give an efficient algorithm that returns the maximum possible value of $\sum_{i=1}^{k} j_i$.

3. **Maximum alternating subsequence**

   An *alternating sequence* is a sequence $a_1, a_2, \ldots$ such that no three consecutive terms of the sequence satisfy $a_i > a_{i+1} > a_{i+2}$ or $a_i < a_{i+1} < a_{i+2}$.

   Given a sequence, efficiently find the longest alternating subsequence it contains. What is the running time of your algorithm?

1. **Championship Showdown**

   What excitement! The Champaign Spinners and the Urbana Dreamweavers have advanced to meet each other in the World Series of Basketweaving! The World Champions will be decided by a best of $2n - 1$ series of head-to-head weaving matches, and the first to win n matches will take home the coveted Golden Basket (for example, a best-of-7 series requires four match wins, but we will keep the generalized case). We know that for any given match there is a constant probability p that Champaign will win, and a subsequent probability $q = 1 - p$ that Urbana will win.

   Let $P(i, j)$ be the probability that Champaign will win the series given that they still need i more victories, whereas Urbana needs j more victories for the championship. $P(0, j) = 1$, $1 \le j \le n$, because Champaign needs no more victories to win. $P(i, 0) = 0$, $1 \le i \le n$, as Champaign cannot possibly win if Urbana already has. $P(0, 0)$ is meaningless. Champaign wins any particular match with probability p and loses with probability q, so

   $$P(i, j) = p \cdot P(i - 1, j) + q \cdot P(i, j - 1)$$

   for any $i \ge 1$ and $j \ge 1$.

   Create and analyze an $O(n^2)$-time dynamic programming algorithm that takes the parameters n, p, and q and returns the probability that Champaign will win the series (that is, calculate $P(n, n)$).

2. **Making Change**

   Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \cdots < c[n]$. (In the U.S., for example, $n = 6$ and the values are $1, 5, 10, 25, 50$, and 100 cents.) Your beloved benevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.

   Describe and analyze a dynamic programming algorithm to determine, given a target amount A and a sorted array $c[1..n]$ of coin values, the smallest number of coins needed to make A cents in change. You can assume that $c[1] = 1$, so that it is possible to make change for any amount A.

3. **Knapsack**

   You are a thief, who is trying to choose the best collection of treasure (some subset of the n treasures, numbered 1 through n) to steal. The weight of item i is $w_i \in \mathbb{N}$ and the profit is $p_i \in \mathbb{R}$. Let $C \in \mathbb{N}$ be the maximum weight that your knapsack can hold. Your goal is to choose a subset of elements $S \subseteq \{1, 2, \ldots, n\}$ that maximizes your total profit $P(S) = \sum_{i \in S} p_i$, subject to the constraint that the sum of the weights $W(S) = \sum_{i \in S} w_i$ is not more than C.

   Give an algorithm that runs in time $O(Cn)$.

1. **Randomized Edge Cuts**

   We will randomly partition the vertex set of a graph G into two sets S and T. The algorithm is to flip a coin for each vertex and with probability $1/2$, put it in S; otherwise put it in T.

   (a) Show that the expected number of edges with one endpoint in S and the other endpoint in T is exactly half the edges in G.

   (b) Now say the edges have weights. What can you say about the sum of the weights of the edges with one endpoint in S and the other endpoint in T?

2. **Skip Lists**

   A *skip list* is built in layers. The bottom layer is an ordinary sorted linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer $i + 1$ with some fixed probability p.

```
1
1-----4---6
1---3-4---6-----9
1-2-3-4-5-6-7-8-9-10
```

   (a) What is the probability a node reaches height h.

   (b) What is the probability any node is above $c \log n$ (for some fixed value of c)?
   Compute the value explicitly when $p = 1/2$ and $c = 4$.

   (c) To search for an entry x, scan the top layer until you find the last entry y that is less than or equal to x. If $y < x$, drop down one layer and in this new layer (beginning at y) find the last entry that is less than or equal to x. Repeat this process (dropping down a layer, then finding the last entry less than or equal to x) until you either find x or reach the bottom layer and confirm that x is not in the skip list. What is the expected search time?

   (d) Describe an efficient method for insertion. What is the expected insertion time?

3. **Clock Solitaire**

   In a standard deck of 52 cards, put 4 face-down in each of the 12 'hour' positions around a clock, and 4 face-down in a pile in the center. Turn up a card from the center, and look at the number on it. If it's number x, place the card face-up next to the face-down pile for x, and turn up the next card in the face-down pile for x (that is, the face-down pile corresponding to hour x). You win if, for each Ace $\leq x \leq$ Queen, all four cards of value x are turned face-up before all four Kings (the center cards) are turned face-up.

   What is the probability that you win a game of Clock Solitaire?

1. **Simulating Queues with Stacks**

   A *queue* is a first-in-first-out data structure. It supports two operations *push* and *pop*. Push adds a new item to the back of the queue, while pop removes the first item from the front of the queue. A *stack* is a last-in-first-out data structure. It also supports push and pop. As with a queue, push adds a new item to the back of the queue. However, pop removes the last item from the back of the queue (the one most recently added).

   Show how you can simulate a queue by using two stacks. Any sequence of pushes and pops should run in amortized constant time.

2. **Multistacks**

   A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i, we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2}, and so on. To clarify, a user can only push elements onto S_0. All other pushes and pops happen in order to make space to push onto S_0. Moving a single element from one stack to the next takes $O(1)$ time.

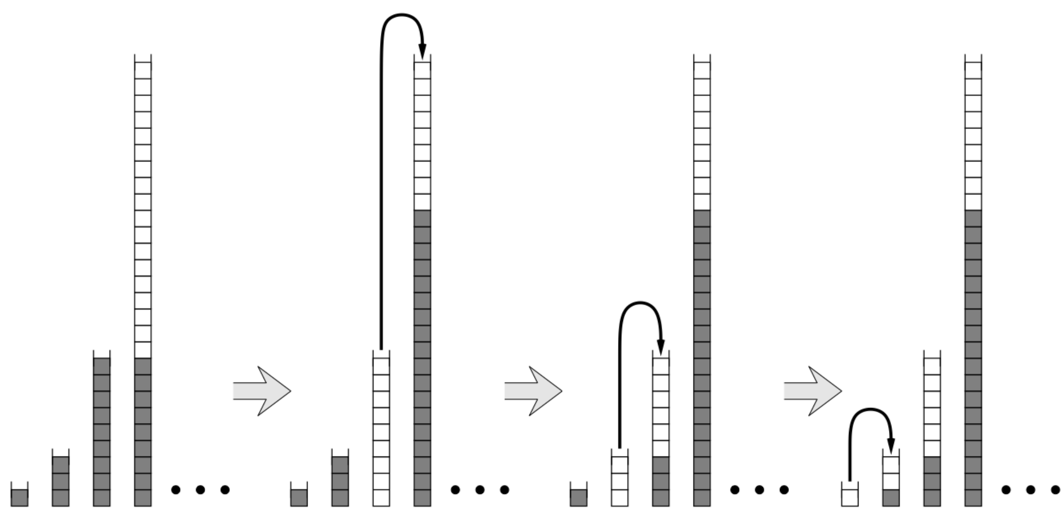


Figure 1. Making room for one new element in a multistack.

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?

   (b) Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack.

3. **Powerhungry function costs**

   A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. Determine the amortized cost of the operation.

1. **Representation of Integers**

   (a) Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers–if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.

   (b) Prove that any integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.

2. **Minimal Dominating Set**

   Suppose you are given a rooted tree T (not necessarily binary). You want to label each node in T with an integer 0 or 1, such that every node either has the label 1 or is adjacent to a node with the label 1 (or both). The *cost* of a labeling is the number of nodes with label 1. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T.

3. **Names in Boxes**

   The names of 100 prisoners are placed in 100 wooden boxes, one name to a box, and the boxes are lined up on a table in a room. One by one, the prisoners are led into the room; each may look in at most 50 boxes, but must leave the room exactly as he found it and is permitted no further communication with the others.

   The prisoners have a chance to plot their strategy in advance, and they are going to need it, because unless *every single prisoner finds his own name* all will subsequently be executed. Find a strategy for them which has probability of success exceeding 30%. You may assume that the names are distributed in the boxes uniformly at random.

   (a) Calculate the probability of success if each prisoner picks 50 boxes uniformly at random.

   *(b) Consider the following strategy.

   The prisoners number themselves 1 to 100. Prisoner i begins by looking in box i. There he finds the name of prisoner j. If $j \neq i$, he continues by looking in box j. As long as prisoner i has not found his name, he continues by looking in the box corresponding to the last name he found.

   Describe the set of permutations of names in boxes for which this strategy will succeed.

   *(c) Count the number of permutations for which the strategy above succeeds. Use this sum to calculate the probability of success. You may find it useful to do this calculation for general n, then set $n = 100$ at the end.

   (d) We assumed that the names were distributed in the boxes uniformly at random. Explain how the prisoners could augment their strategy to make this assumption unnecessary.

1. **Dynamic MSTs**

   Suppose that you already have a minimum spanning tree (MST) in a graph. Now one of the edge weights changes. Give an efficient algorithm to find an MST in the new graph.

2. **Minimum Bottleneck Trees**

   In a graph G, for any pair of vertices u, v, let bottleneck(u, v) be the maximum over all paths p_i from u to v of the minimum-weight edge along p_i. Construct a spanning tree T of G such that for each pair of vertices, their bottleneck in G is the same as their bottleneck in T.

   One way to think about it is to imagine the vertices of the graph as islands, and the edges as bridges. Each bridge has a maximum weight it can support. If a truck is carrying stuff from u to v, how much can the truck carry? We don't care what route the truck takes; the point is that the smallest-weight edge on the route will determine the load.

3. **Eulerian Tours**

   An *Eulerian tour* is a "walk along edges of a graph" (in which successive edges must have a common endpoint) that uses each edge exactly once and ends at the vertex where it starts. A graph is called Eulerian if it has an Eulerian tour.

   Prove that a connected graph is Eulerian iff each vertex has even degree.

1. **Alien Abduction**

   Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road won't be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

   More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.

2. **The Only SSSP Algorithm**

   In the lecture notes, Jeff mentions that all SSSP algorithms are special cases of the following generic SSSP algorithm. Each vertex v in the graph stores two values, which describe a tentative shortest path from s to v.

   - $\text{dist}(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path.
   - $\text{pred}(v)$ is the predecessor of v in the shortest $s \rightsquigarrow v$ path.

   We call an edge *tense* if $\text{dist}(u) + w(u \to v) < \text{dist}(v)$. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

   ```
   Relax(u → v):
     dist(v) ← dist(u) + w(u → v)
     pred(v) ← u
   ```

   If there are no tense edges, our algorithm is finished, and we have our desired shortest path tree. The correctness of the relaxation algorithm follows directly from three simple claims. The first of these is below. Prove it.

   - When the algorithm halts, if $\text{dist}(v) \neq \infty$, then $\text{dist}(v)$ is the total weight of the predecessor chain ending at v:
   $$s \to \cdots \to (pred(pred(v)) \to pred(v) \to v.$$

3. **Can't find a Cut-edge**

   A cut-edge is an edge which when deleted disconnects the graph. Prove or disprove the following. Every 3-regular graph has no cut-edge. (A common approach is induction.)

1. **Max-Flow with vertex capacities**

   In a standard $s-t$ Maximum-Flow Problem, we assume edges have capacities, and there is no limit on how much flow is allowed to pass through a node. In this problem, we consider the variant of Maximum-Flow and Minimum-Cut problems with node capacities.

   More specifically, each node, n_i, has a capacity c_i. The edges have unlimited capacity. Show how you can model this problem as a standard Max-flow problem (where the weights are on the edges).

2. **Emergency evacuation**

   Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be reushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location.

   At the same time, we don't want to overload any hospital by sending too many patients its way. We'd like to distribute the people so that each hospital receives at most $\lceil n/k \rceil$ people.

   Show how to model this problem as a Max-flow problem.

3. **Tracking a Hacker**

   A computer network (with each edge weight 1) is designed to carry traffic from a source s to a destination t. Recently, a computer hacker destroyed some of the edges in the graph. Normally, the maximum $s-t$ flow in G is k. Unfortunately, there is currently no path from s to t. Fortunately, the sysadmins know that the hacker destroyed at most k edges of the graph.

   The sysadmins are trying to diagnose which of the nodes of the graph are no longer reachable. They would like to avoid testing each node. They are using a monitoring tool with the following behavior. If you use the command *ping(v)*, for a given node v, it will tell you whether there is currently a path from s to v (so *ping(t)* will return `False` but *ping(s)* will return `True`).

   Give an algorithm that accomplishes this task using only $O(k \log n)$ pings. (You may assume that any algorithm you wish to run on the original network (before the hacker destroyed edges) runs for free, since you have a model of that network on your computer.)

1. **Updating a maximum flow**

   Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e, a designated source $s \in V$, and a designated sink $t \in V$. You are also given a maximum $s - t$ flow in G, defined by a flow value f_e on each edge e. The flow $\{f_e\}$ is *acyclic*: There is no cycle in G on which all edges carry positive flow.

   Now suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m + n)$, where m is the number of edges in G and n is the number of nodes.

2. **Cooking Schedule**

   You live in a cooperative apartment with n other people. The co-op needs to schedule cooks for the next n days, so that each person cooks one day and each day there is one cook. In addition, each member of the co-op has a list of days they are available to cook (and is unavailable to cook on the other days).

   Because of your superior CS473 skills, the co-op selects you to come up with a schedule for cooking, so that everyone cooks on a day they are available.

   (a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible schedule for the co-op.

   (b) A friend of yours tried to help you out by coming up with a cooking schedule. Unfortunately, when you look at the schedule he created, you notice a big problem. $n - 2$ of the people are scheduled for different nights on which they are available: no problem there. But the remaining two people are assigned to cook on the same night (and no one is assigned to the last night).

   You want to fix your friend's mistake, but without having to recompute everything from scratch. Show that it's possible, using his "almost correct" schedule to decide in $O(n^2)$ time whether there exists a feasible schedule.

3. **Disjoint paths in a digraph**

   Let $G = (V, E)$ be a directed graph, and suppose that for each node v, the number of edges into v is equal to the number of edges out of v. That is, for all v,

   $$|\{(u, v) : (u, v) \in E\}| = |\{(v, w) : (v, w) \in E\}|.$$

   Let x, y be two nodes of G, and suppose that there exist k mutually edge-disjoint paths from x to y. Under these conditions, does it follow that there exist k mutually edge-disjoint paths from y to x. Give a proof or a counterexample with explanation.

1. **String matching: an example**

   (a) Build a finite automata to search for the string "bababoon".

   (b) Use the automata from part (a) to build the prefix function for Knuth-Morris-Pratt.

   (c) Use the automata or the prefix function to search for "bababoon" in the string "babybaboon-buysbananasforotherbabybababoons".

2. **Cooking Schedule Strikes Back**

   You live in a cooperative apartment with n other people. The co-op needs to schedule cooks for the next $5n$ days, so that each person cooks five days and each day there is one cook. In addition, each member of the co-op has a list of days they are available to cook (and is unavailable to cook on the other days).

   Because of your success at headbanging last week, the co-op again asks you to compose a cooking schedule. Unfortunately, you realize that no such schedule is possible Give a schedule for the cooking so that no one has to cook on more than 2 days that they claim to be unavailable.

3. **String matching on Trees**

   You are given a rooted tree T (not necessarily binary), in which each node has a character. You are also given a pattern $P = p_1 p_2 \cdots p_l$. Search for the string as a subtree. In other words, search for a subtree in which p_i is on a child of the node containing p_{i-1} for each $2 \leq i \leq l$.

1. **Self-reductions**

   In each case below assume that you are given a black box which can answer the decision version of the indicated problem. Use a polynomial number of calls to the black box to construct the desired set.

   (a) Independent set: Given a graph G and an integer k, does G have a subset of k vertices that are pairwise nonadjacent?

   (b) Subset sum: Given a multiset (elements can appear more than once) $X = \{x_1, x_2, \ldots, x_k\}$ of positive integers, and a positive integer S does there exist a subset of X with sum exactly S?

2. **Lower Bounds**

   Give adversary arguments to prove the indicated lower bounds for the following problems:

   (a) Searching in a sorted array takes at least $1 + \lfloor \lg_2 n \rfloor$ queries.

   (b) Let M be an $n \times n$ array of real values that is increasing in both rows and columns. Prove that searching for a value requires at least n queries.

3. **k-coloring**

   Show that we can solve the problem of constructing a k-coloring of a graph by using a polynomial number of calls to a black box that determines whether a graph has such a k-coloring. (Hint: Try reducing via an intermediate problem that asks whether a partial coloring of a graph can be extended to a proper k-coloring.)

1. **NP-hardness Proofs: Restriction**
   Prove that each of the following problems is NP-hard. In each part, find a special case of the given problem that is equivalent to a known NP-hard problem.

   (a) **Longest Path**
   Given a graph G and a positive integer k, does G contain a path with k or more edges?

   (b) **Partition into Hamiltonian Subgraphs**
   Given a graph G and a positive integer k, can the vertices of G be partitioned into at most k disjoint sets such that the graph induced by each set has a Hamiltonian cycle?

   (c) **Set Packing**
   Given a collection of finite sets C and a positive integer k, does C contain k disjoint sets?

   (d) **Largest Common Subgraph**
   Given two graphs G_1 and G_2 and a positive integer k, does there exist a graph G_3 such that G_3 is a subgraph of both G_1 and G_2 and G_3 has at least k edges?

2. **Domino Line**
   You are given an unusual set of dominoes; each domino has a number on each end, but the numbers may be arbitarily large and some numbers appear on many dominoes, while other numbers only appear on a few dominoes. Your goal is to form a line using all the dominoes so that adjacent dominoes have the same number on their adjacent halves. Either give an efficient algorithm to solve the problem or show that it is NP-hard.

3. **Set Splitting**
   Given a finite set S and a collection of subsets C is there a partition of S into two sets S_1 and S_2 such that no subset in C is contained entirely in S_1 or S_2? Show that the problem is NP-hard. (Hint: use NAE-3SAT, which is similar to 3SAT except that a satisfying assingment does not allow all 3 variables in a clause to be true.)

You have 120 minutes to answer four of these five questions.

Write your answers in the separate answer booklet.

1. **Multiple Choice.**

Each of the questions on this page has one of the following five answers:

A: $\Theta(1)$	B: $\Theta(\log n)$	C: $\Theta(n)$	D: $\Theta(n \log n)$	E: $\Theta(n^2)$

Choose the correct answer for each question. Each correct answer is worth $+1$ point; each incorrect answer is worth $-\frac{1}{2}$ point; each "I don't know" is worth $+\frac{1}{4}$ point. Your score will be rounded to the nearest *non-negative* integer. You do *not* need to justify your answers; just write the correct letter in the box.

(a) What is $\dfrac{5}{n} + \dfrac{n}{5}$?

(b) What is $\displaystyle\sum_{i=1}^{n} \frac{n}{i}$?

(c) What is $\displaystyle\sum_{i=1}^{n} \frac{i}{n}$?

(d) How many bits are required to represent the nth Fibonacci number in binary?

(e) What is the solution to the recurrence $T(n) = 2T(n/4) + \Theta(n)$?

(f) What is the solution to the recurrence $T(n) = 16T(n/4) + \Theta(n)$?

(g) What is the solution to the recurrence $T(n) = T(n-1) + 1/n^2$?

(h) What is the worst-case time to search for an item in a binary search tree?

(i) What is the worst-case running time of quicksort?

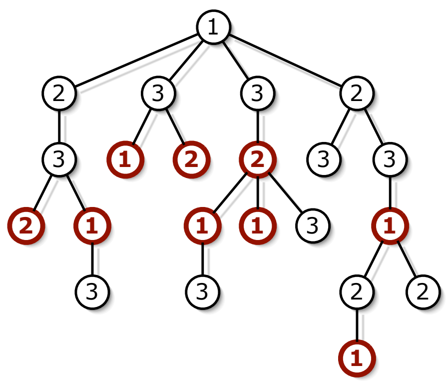
(j) What is the running time of the fastest possible algorithm to solve Sudoku puzzles? A Sudoku puzzle consists of a 9×9 grid of squares, partitioned into nine 3×3 sub-grids; some of the squares contain digits between 1 and 9. The goal of the puzzle is to enter digits into the blank squares, so that each digit between 1 and 9 appears exactly once in each row, each column, and each 3×3 sub-grid. The initial conditions guarantee that the solution is unique.

2							4	
	7		5					
				1		9		
6		4			2			
	8						5	
			9			3		7
		1		4				
					3		8	
	5							6

A Sudoku puzzle. **Don't try to solve this during the exam!**

2. Oh, no! You have been appointed as the gift czar for Giggle, Inc.'s annual mandatory holiday party! The president of the company, who is certifiably insane, has declared that *every* Giggle employee must receive one of three gifts: (1) an all-expenses-paid six-week vacation anywhere in the world, (2) an all-the-pancakes-you-can-eat breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. Corporate regulations prohibit any employee from receiving the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy. How do you decide what gifts everyone gets if you want to minimize the number of people that get fired?

   More formally, suppose you are given a rooted tree T, representing the company hierarchy. You want to label each node in T with an integer 1, 2, or 3, such that every node has a different label from its parent.. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T. (Your algorithm does *not* have to compute the actual best labeling—just its cost.)



A tree labeling with cost 9. Bold nodes have smaller labels than their parents.
This is *not* the optimal labeling for this tree.

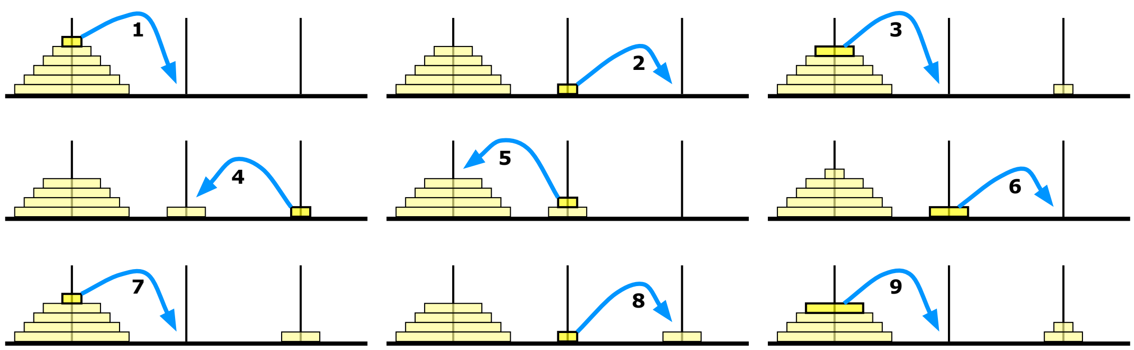
3. Suppose you are given an array $A[1\,..\,n]$ of n distinct integers, sorted in increasing order. Describe and analyze an algorithm to determine whether there is an index i such that $A[i] = i$, in $o(n)$ time. *[Hint: Yes, that's* ***little****-oh of n. What can you say about the sequence $A[i] - i$?]*

4. Describe and analyze a *polynomial-time* algorithm to compute the length of the longest common subsequence of two strings $A[1\,..\,m]$ and $B[1\,..\,n]$. For example, given the strings 'DYNAMIC' and 'PROGRAMMING', your algorithm would return the number 3, because the longest common subsequence of those two strings is 'AMI'. You must give a complete, self-contained solution; don't just refer to HW1.

5. Recall that the Tower of Hanoi puzzle consists of three pegs and n disks of different sizes. Initially, all the disks are on one peg, stacked in order by size, with the largest disk on the bottom and the smallest disk on top. In a single move, you can transfer the highest disk on any peg to a different peg, except that you may never place a larger disk on top of a smaller one. The goal is to move all the disks onto one other peg.

   Now suppose the pegs are arranged in a row, and you are forbidden to transfer a disk directly between the left and right pegs in a single move; every move must involve the middle peg. How many moves suffice to transfer all n disks from the left peg to the right peg under this restriction? **Prove your answer is correct.**

   For full credit, give an *exact* upper bound. A correct upper bound using $O(\cdot)$ notation (with a proof of correctness) is worth 7 points.



The first nine moves in a restricted Towers of Hanoi solution.

1. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, "Get out…while…you…", thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

   You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

   On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can't stand each other's company, so you'll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

   You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger's heart, you wisely decide to leave the park immediately.

   Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along the path is as small as possible. *Be sure to account for* ***both*** *the vertex probabilities* ***and*** *the edge probabilities!*

2. Consider the following solution for the union-find problem, called *union-by-weight*. Each set leader $\overline{x}$ stores the number of elements of its set in the field *weight*$(\overline{x})$. Whenever we UNION two sets, the leader of the *smaller* set becomes a new child of the leader of the *larger* set (breaking ties arbitrarily).

```
MAKESET(x):
    parent(x) ← x
    weight(x) ← 1

FIND(x):
    while x ≠ parent(x)
        x ← parent(x)
    return x

UNION(x, y)
    x̄ ← FIND(x)
    ȳ ← FIND(y)
    if weight(x̄) > weight(ȳ)
        parent(ȳ) ← x̄
        weight(x̄) ← weight(x̄) + weight(ȳ)
    else
        parent(x̄) ← ȳ
        weight(x̄) ← weight(x̄) + weight(ȳ)
```

   Prove that if we use union-by-weight, the *worst-case* running time of FIND is $O(\log n)$.

3. ***Prove or disprove***[1] each of the following statements.

   (a) Let G be an arbitrary undirected graph with arbitrary distinct weights on the edges. The minimum spanning tree of G includes the lightest edge in every cycle in G.

   (b) Let G be an arbitrary undirected graph with arbitrary distinct weights on the edges. The minimum spanning tree of G excludes the heaviest edge in every cycle in G.

4. In Homework 2, you were asked to analyze the following algorithm to find the kth smallest element from an unsorted array. (The algorithm is presented here in iterative form, rather than the recursive form you saw in the homework, but it's exactly the same algorithm.)

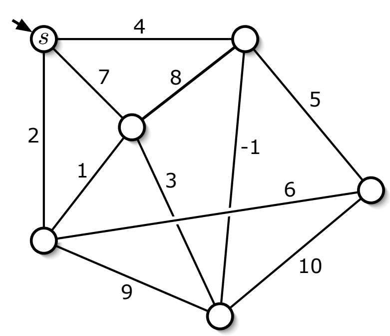
```
QUICKSELECT(A[1 .. n], k):
  i ← 1; j ← n
  while i ≤ j
      r ← PARTITION(A[i .. j], RANDOM(i, j))
      if r = k
          return A[r]
      else if r > k
          j ← r − 1
      else
          i ← r + 1
```

The algorithm relies on two subroutines. RANDOM(i, j) returns an integer chosen uniformly at random from the range $[i \,..\, j]$. PARTITION$(A[i \,..\, j], p)$ partitions the subarray $A[i \,..\, j]$ using the pivot value $A[p]$ and returns the new index of the pivot value in the partitioned array.

What is the *exact* expected number of iterations of the main loop when $k = 1$? ***Prove*** your answer is correct. A correct $\Theta(\cdot)$ bound (with proof) is worth 7 points. You may assume that the input array $A[\,]$ contains n distinct integers.

5. Find the following spanning trees for the weighted graph shown below.

   (a) A breadth-first spanning tree rooted at s.
   (b) A depth-first spanning tree rooted at s.
   (c) A shortest-path tree rooted at s.
   (d) A minimum spanning tree.



You do *not* need to justify your answers; just clearly indicate the edges of each spanning tree. Yes, one of the edges has negative weight.

[1] But not both! If you give us *both* a proof *and* a disproof for the same statement, you will get no credit, even if one of your arguments is correct.

1. A *double-Hamiltonian* circuit in an undirected graph G is a closed walk that visits every vertex in G exactly *twice*, possibly by traversing some edges more than once. ***Prove*** that it is NP-hard to determine whether a given undirected graph contains a double-Hamiltonian circuit.

2. Suppose you are running a web site that is visited by the same set of people every day. Each visitor claims membership in one or more *demographic groups*; for example, a visitor might describe himself as male, 31-40 years old, a resident of Illinois, an academic, a blogger, a Joss Whedon fan[1], and a Sports Racer.[2] Your site is supported by advertisers. Each advertiser has told you which demographic groups should see its ads and how many of its ads you must show each day. Altogether, there are n visitors, k demographic groups, and m advertisers.

   Describe an efficient algorithm to determine, given all the data described in the previous paragraph, whether you can show each visitor exactly *one* ad per day, so that every advertiser has its desired number of ads displayed, and every ad is seen by someone in an appropriate demographic group.

3. Describe and analyze a data structure to support the following operations on an array $X[1\,..\,n]$ as quickly as possible. Initially, $X[i] = 0$ for all i.

   - Given an index i such that $X[i] = 0$, set $X[i]$ to 1.
   - Given an index i, return $X[i]$.
   - Given an index i, return the smallest index $j \geq i$ such that $X[j] = 0$, or report that no such index exists.

   For full credit, the first two operations should run in *worst-case constant* time, and the amortized cost of the third operation should be as small as possible. *[Hint: Use a modified union-find data structure.]*

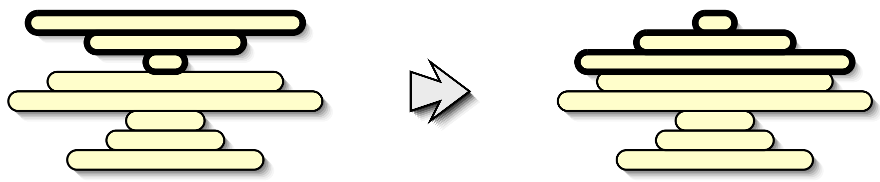
4. The next time you are at a party, one of the guests will suggest everyone play a round of Three-Way Mumbledypeg, a game of skill and dexterity that requires three teams and a knife. The official Rules of Three-Way Mumbledypeg (fixed during the Holy Roman Three-Way Mumbledypeg Council in 1625) require that (1) each team *must* have at least one person, (2) any two people on the same team *must* know each other, and (3) everyone watching the game *must* be on one of the three teams. Of course, it will be a really *fun* party; nobody will want to leave. There will be several pairs of people at the party who don't know each other. The host of the party, having heard thrilling tales of your prowess in all things algorithmic, will hand you a list of which pairs of partygoers know each other and ask you to choose the teams, while he sharpens the knife.

   Either describe and analyze a polynomial time algorithm to determine whether the partygoers can be split into three legal Three-Way Mumbledypeg teams, or prove that the problem is NP-hard.

[1] Har har har! Mine is an evil laugh! Now *die!*

[2] It's Ride the Fire Eagle Danger Day!

5. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.



Flipping the top three pancakes.

   (a) Describe an efficient algorithm to sort an arbitrary stack of n pancakes. *Exactly* how many flips does your algorithm perform in the worst case? (For full credit, your algorithm should perform as few flips as possible; an optimal $\Theta()$ bound is worth three points.)

   (b) Now suppose one side of each pancake is burned. *Exactly* how many flips do you need to sort the pancakes *and* have the burned side of every pancake on the bottom? (For full credit, your algorithm should perform as few flips as possible; an optimal $\Theta()$ bound is worth three points.)

6. Describe and analyze an efficient algorithm to find the length of the longest substring that appears both forward and backward in an input string $T[1 \,..\, n]$. The forward and backward substrings must not overlap. Here are several examples:

   - Given the input string ALGORITHM, your algorithm should return 0.
   - Given the input string RECURSION, your algorithm should return 1, for the substring R.
   - Given the input string REDIVIDE, your algorithm should return 3, for the substring EDI. (The forward and backward substrings must not overlap!)
   - Given the input string DYNAMICPROGRAMMINGMANYTIMES, your algorithm should return 4, for the substring YNAM.

   For full credit, your algorithm should run in $O(n^2)$ time.

7. A *double-Eulerian* circuit in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Describe and analyze a *polynomial-time* algorithm to determine whether a given undirected graph contains a double-Eulerian circuit.

CS 473G: Graduate Algorithms, Spring 2007
Homework 0

Due **<u>in class</u>** at 11:00am, Tuesday, January 30, 2007

Name:	
Net ID:	Alias:

☐ I understand the Course Policies.

- Neatly print your full name, your NetID, and an alias of your choice in the boxes above, and staple this page to your solution to problem 1. We will list homework and exam grades on the course web site by alias. **By providing an alias, you agree to let us list your grades; if you do not provide an alias, your grades will not be listed.** For privacy reasons, your alias should not resemble your name, your NetID, your university ID number, or (God forbid!) your Social Security number. Please use the same alias for every homework and exam.

- Read the Course Policies on the course web site, and then check the box above. Among other things, this page describes what we expect in your homework solutions, as well as policies on grading standards, regrading, extra credit, and plagiarism. In particular:

  – Submit each numbered problem separately, on its own piece(s) of paper. If you need more than one page for a problem, staple just *those* pages together, but keep different problems separate. **Do not staple your entire homework together.**

  – You may use *any* source at your disposal—paper, electronic, or human—but you *must* write your answers in your own words, and you *must* cite every source that you use.

  – Algorithms or proofs containing phrases like "and so on" or "repeat this for all n", instead of an explicit loop, recursion, or induction, are worth zero points.

  – Answering "I don't know" to any homework or exam problem is worth 25% partial credit.

  If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup.

- This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, discrete probability, graphs, and most importantly, induction—to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** The early chapters of Kleinberg and Tardos (or any algorithms textbook) should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks.

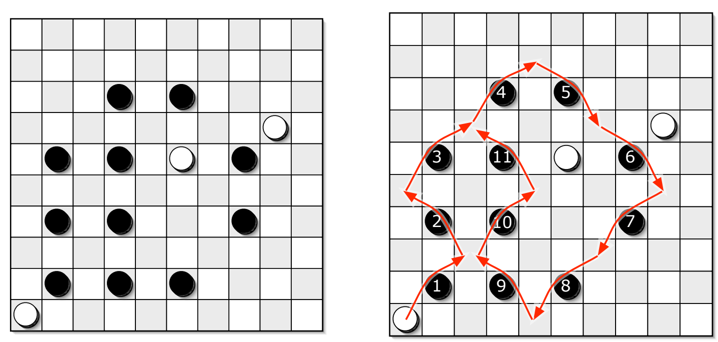
- Every homework will have five problems, each worth 10 points. Stars indicate more challenging problems. Many homeworks will also include an extra-credit problem.

---

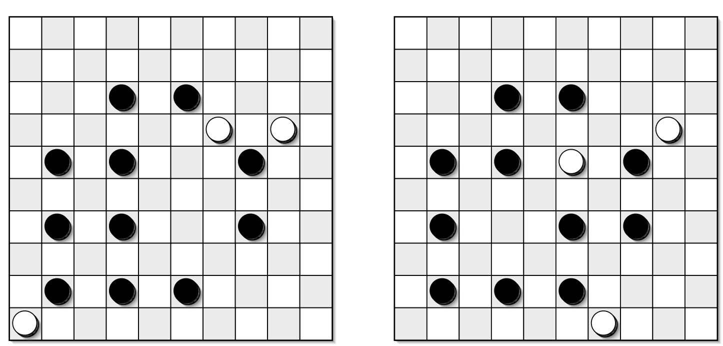
*1. Draughts/checkers is a game played on an $m \times m$ grid of squares, alternately colored light and dark. (The game is usually played on an 8×8 or 10×10 board, but the rules easily generalize to any board size.) Each dark square is occupied by at most one game piece (usually called a *checker* in the U.S.), which is either black or white; light squares are always empty. One player ("White") moves the white pieces; the other ("Black") moves the black pieces.

Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[1] Pieces can be moved in any of the four diagonal directions, either one or two steps at a time. On each turn, a player either *moves* one of her pieces one step diagonally into an empty square, or makes a series of *jumps* with one of her checkers. In a single jump, a piece moves to an empty square two steps away in any diagonal direction, but only if the intermediate square is occupied by a piece of the opposite color; this enemy piece is *captured* and immediately removed from the board. Multiple jumps are allowed in a single turn as long as they are made by the same piece. A player wins if her opponent has no pieces left on the board.

Describe an algorithm[2] that correctly determines whether White can capture every black piece, thereby winning the game, *in a single turn*. The input consists of the width of the board (m), a list of positions of white pieces, and a list of positions of black pieces. For full credit, your algorithm should run in $O(n)$ time, where n is the total number of pieces, but any algorithm that runs in time polynomial in n and m is worth significant partial credit.



White wins in one turn.



White cannot win in one turn from either of these positions.

*[Hint: The greedy strategy—make arbitrary jumps until you get stuck—does **not** always find a winning sequence of jumps even when one exists.]*

[1] Most variants of draughts have 'flying kings', which behave *very* differently than what's described here.
[2] Since you've read the Course Policies, you know what this phrase means.

2. (a) Prove that any positive integer can be written as the sum of distinct powers of 2. *[Hint: "Write the number in binary" is **not** a proof; it just restates the problem.]* For example:

$$\begin{aligned} 16+1 = \quad & 17 \quad &= 2^4+2^0 \\ 16+4+2+1 = \quad & 23 \quad &= 2^4+2^2+2^1+2^0 \\ 32+8+1 = \quad & 42 \quad &= 2^5+2^3+2^1 \end{aligned}$$

   (b) Prove that *any* integer (positive, negative, or zero) can be written as the sum of distinct powers of -2. For example:

$$\begin{aligned} -32+16-2+1 = \quad & -17 \quad &= (-2)^5+(-2)^4+(-2)^1+(-2)^0 \\ 64-32-8-2+1 = \quad & 23 \quad &= (-2)^6+(-2)^5+(-2)^3+(-2)^1+(-2)^0 \\ 64-32+16-8+4-2 = \quad & 42 \quad &= (-2)^6+(-2)^5+(-2)^4+(-2)^3+(-2)^2+(-2)^1 \end{aligned}$$

3. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

   Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left.

4. On their long journey from Denmark to England, Rosencrantz and Guildenstern amuse themselves by playing the following game with a fair coin. First Rosencrantz flips the coin over and over until it comes up tails. Then Guildenstern flips the coin over and over until he gets as many heads in a row as Rosencrantz got on his turn. Here are three typical games:

   Rosencrantz: $\underline{\mathsf{H\ H}}$ T
   Guildenstern: $\mathsf{H\ T}$ $\underline{\mathsf{H\ H}}$

   Rosencrantz: T
   Guildenstern: (no flips)

   Rosencrantz: $\underline{\mathsf{H\ H\ H}}$ T
   Guildenstern: $\mathsf{T\ H\ H\ T\ H\ H\ T\ H\ T\ T}$ $\underline{\mathsf{H\ H\ H}}$

   (a) What is the expected number of flips in one of Rosencrantz's turns?

   (b) Suppose Rosencrantz flips k heads in a row on his turn. What is the expected number of flips in Guildenstern's next turn?

   (c) What is the expected total number of flips (by both Rosencrantz and Guildenstern) in a single game?

Prove that your answers are correct. If you have to appeal to "intuition" or "common sense", your answer is almost certainly wrong! You must give *exact* answers for full credit, but a correct asymptotic bound for part (b) is worth significant credit.

5. (a) **[5 pts]** Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Assume reasonable but nontrivial base cases. If your solution requires a particular base case, say so.

$$A(n) = 3A(n/9) + \sqrt{n}$$

$$B(n) = 4B(n-1) - 4B(n-2)$$

$$C(n) = \frac{\pi\, C(n-1)}{\sqrt{2}\, C(n-2)} \qquad \textit{[Hint: This is easy!]}$$

$$D(n) = \max_{n/4<k<3n/4} \big(D(k) + D(n-k) + n\big)$$

$$E(n) = 2E(n/2) + 4E(n/3) + 2E(n/6) + n^2$$

Do *not* turn in proofs—just a list of five functions—but you should do them anyway, just for practice. *[Hint: On the course web page, you can find a handout describing several techniques for solving recurrences.]*

(b) **[5 pts]** Sort the functions in the box from asymptotically smallest to asymptotically largest, indicating ties if there are any. **Do *not* turn in proofs**—just a sorted list of 16 functions—but you should do them anyway, just for practice.

To simplify your answer, write $f(n) \ll g(n)$ to indicate that $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

n	$\lg n$	$\sqrt{n}$	3^n
$\sqrt{\lg n}$	$\lg \sqrt{n}$	$3^{\sqrt{n}}$	$\sqrt{3^n}$
$3^{\lg n}$	$\lg(3^n)$	$3^{\lg \sqrt{n}}$	$3^{\sqrt{\lg n}}$
$\sqrt{3^{\lg n}}$	$\lg(3^{\sqrt{n}})$	$\lg \sqrt{3^n}$	$\sqrt{\lg(3^n)}$

Recall that $\lg n = \log_2 n$.

CS 473G: Graduate Algorithms, Spring 2007
Homework 1

Due February 6, 2007

Remember to submit **separate, individually stapled** solutions to each of the problems.

---

1. Jeff tries to make his students happy. At the beginning of class, he passes out a questionnaire to students which lists a number of possible course policies in areas where he is flexible. Every student is asked to respond to each possible course policy with one of "strongly favor", "mostly neutral", or "strongly oppose". Each student may respond with "strongly favor" or "strongly oppose" to at most five questions. Because Jeff's students are very understanding, each student is happy if he or she prevails in just one of his or her strong policy preferences. Either describe a polynomial time algorithm for setting course policy to maximize the number of happy students or show that the problem is NP-hard.

2. Consider a variant 3SAT′ of 3SAT which asks, given a formula ϕ in conjunctive normal form in which each clause contains at most 3 literals and each variable appears in at most 3 clauses, is ϕ satisfiable? Prove that 3SAT′ is NP-complete.

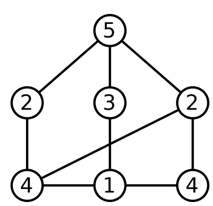
3. For each problem below, either describe a polynomial-time algorithm to solve the problem or prove that the problem is NP-complete.

   (a) A *double-Eulerian* circuit in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G, does G have a double-Eulerian circuit?

   (b) A *double-Hamiltonian* circuit in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G, does G have a double-Hamiltonian circuit?

4. Suppose you have access to a magic black box; if you give it a graph G as input, the black box will tell you, in constant time, if there is a proper 3-coloring of G. Describe a polynomial time algorithm which, given a graph G that is 3-colorable, uses the black box to compute a 3-coloring of G.

5. Let C_5 be the graph which is a cycle on five vertices. A $(5,2)$-coloring of a graph G is a function $f : V(G) \to \{1,2,3,4,5\}$ such that every pair $\{u,v\}$ of adjacent vertices in G is mapped to a pair $\{f(u), f(v)\}$ of vertices in C_5 which are at distance two from each other.



A $(5,2)$-coloring of a graph.

Using a reduction from 5COLOR, prove that the problem of deciding whether a given graph G has a $(5,2)$-coloring is NP-complete.

CS 473G: Graduate Algorithms, Spring 2007
Homework 2

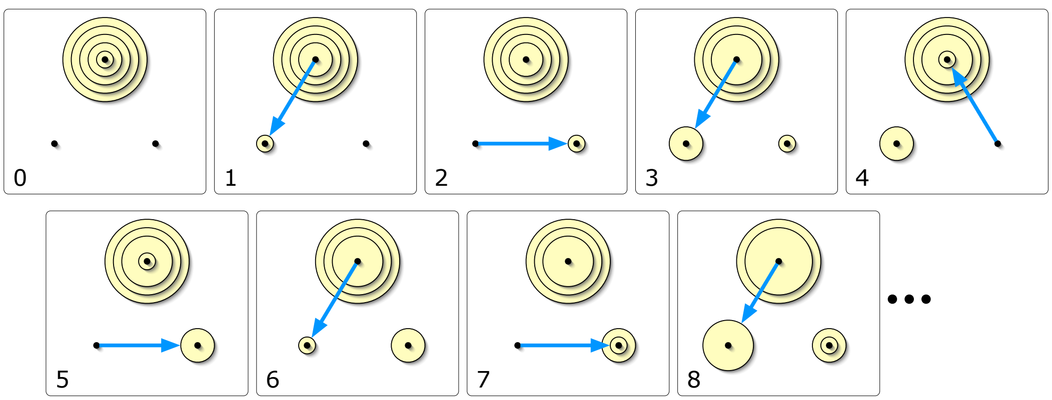
Due Tuesday, February 20, 2007

Remember to submit **separate, individually stapled** solutions to each problem.

As a general rule, a complete full-credit solution to any homework problem should fit into two typeset pages (or five hand-written pages). If your solution is significantly longer than this, you may be including too much detail.

---

1. Consider a restricted variant of the Tower of Hanoi puzzle, where the three needles are arranged in a triangle, and you are required to move each disk *counterclockwise*. Describe an algorithm to move a stack of n disks from one needle to another. *Exactly* how many moves does your algorithm perform? To receive full credit, your algorithm must perform the minimum possible number of moves. *[Hint: Your answer will depend on whether you are moving the stack clockwise or counterclockwise.]*



A top view of the first eight moves in a counterclockwise Towers of Hanoi solution

*2. You find yourself working for The Negation Company ("We Contradict Everything... Not!"), the world's largest producer of multi-bit Boolean inverters. Thanks to a recent mining discovery, the market prices for amphigen and opoterium, the key elements used in AND and OR gates, have plummeted to almost nothing. Unfortunately, the market price of inverton, the essential element required to build NOT gates, has recently risen sharply as natural supplies are almost exhausted. Your boss is counting on you to radically redesign the company's only product in response to these radically new market prices.

Design a Boolean circuit that inverts $n = 2^k - 1$ bits, using only k NOT gates but *any* number of AND and OR gates. The input to your circuit consists of n bits $x_1, x_2, \ldots, x_n$, and the output consists of n bits $y_1, y_2, \ldots, y_n$, where each output bit y_i is the inverse of the corresponding input bit x_i. *[Hint: Solve the case $k = 2$ first.]*

3. (a) Let $X[1..m]$ and $Y[1..n]$ be two arbitrary arrays. A *common supersequence* of X and Y is another sequence that contains both X and Y as subsequences. Give a simple recursive definition for the function *scs*(X, Y), which gives the length of the *shortest* common supersequence of X and Y.

   (b) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Give a simple recursive definition for the function *los*(X), which gives the length of the longest oscillating subsequence of an arbitrary array X of integers.

   (c) Call a sequence $X[1..n]$ of integers *accelerating* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Give a simple recursive definition for the function *lxs*(X), which gives the length of the longest accelerating subsequence of an arbitrary array X of integers.

   Each recursive definition should translate directly into a recursive algorithm, *but you do **not** need to analyze these algorithms.* We are looking for correctness and *simplicity*, not algorithmic efficiency. Not yet, anyway.

4. Describe an algorithm to solve 3SAT in time $O(\phi^n \operatorname{poly}(n))$, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$. *[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]*

5. (a) Describe an algorithm that determines whether a given set of n integers contains two distinct elements that sum to zero, in $O(n \log n)$ time.

   (b) Describe an algorithm that determines whether a given set of n integers contains *three* distinct elements that sum to zero, in $O(n^2)$ time.

   (c) Now suppose the input set X contains n integers between $-10000n$ and $10000n$. Describe an algorithm that determines whether X contains three ***distinct*** elements that sum to zero, in $O(n \log n)$ time.

   For example, if the input set is $\{-10, -9, -7, -3, 1, 3, 5, 11\}$, your algorithm for part (a) should return TRUE, because $(-3) + 3 = 0$, and your algorithms for parts (b) and (c) should return FALSE, even though $(-10) + 5 + 5 = 0$.

CS 473G: Graduate Algorithms, Spring 2007
Homework 3

Due Friday, March 9, 2007

Remember to submit **separate, individually stapled** solutions to each problem.

As a general rule, a complete, full-credit solution to any homework problem should fit into two typeset pages (or five hand-written pages). If your solution is significantly longer than this, you may be including too much detail.

---

1. (a) Let $X[1\,..\,m]$ and $Y[1\,..\,n]$ be two arbitrary arrays. A *common supersequence* of X and Y is another sequence that contains both X and Y as subsequences. Describe and analyze an efficient algorithm to compute the function *scs*(X, Y), which gives the length of the *shortest* common supersequence of X and Y.

   (b) Call a sequence $X[1\,..\,n]$ *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Describe and analyze an efficient algorithm to compute the function *los*(X), which gives the length of the longest oscillating subsequence of an arbitrary array X of integers.

   (c) Call a sequence $X[1\,..\,n]$ of integers *accelerating* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Describe and analyze an efficient algorithm to compute the function *lxs*(X), which gives the length of the longest accelerating subsequence of an arbitrary array X of integers.

   [Hint: Use the recurrences you found in Homework 2. You do not need to prove ***again*** *that these recurrences are correct.]*

2. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \operatorname{poly}(n))$ time. Given an undirected n-vertex graph G with weighted edges, your algorithm should return the weight of the lightest Hamiltonian cycle in G (or ∞ if G has no Hamiltonian cycles).

3. Let G be an arbitrary undirected graph. A set of cycles $\{c_1, \ldots, c_k\}$ in G is *redundant* if it is non-empty and every edge in G appears in an even number of c_i's. A set of cycles is *independent* if it contains no redundant subsets. (In particular, the empty set is independent.) A maximal independent set of cycles is called a *cycle basis* for G.

   (a) Let C be any cycle basis for G. Prove that for any cycle γ in G **that is not an element of C**, there is a subset $A \subseteq C$ such that $A \cup \{\gamma\}$ is redundant. In other words, prove that γ is the 'exclusive or' of some subset of basis cycles.

   Solution: The claim follows directly from the definitions. A cycle basis is a *maximal* independent set, so if C is a cycle basis, then for any cycle $\gamma \notin C$, the larger set $C \cup \{\gamma\}$ cannot be an independent set, so it must contain a redundant subset. On the other hand, if C is a basis, then C is independent, so C contains no redundant subsets. Thus, $C \cup \{\gamma\}$ must have a redundant subset B that contains γ. Let $A = B \setminus \{\gamma\}$. ■

(b) Prove that the set of independent cycle sets form a matroid.

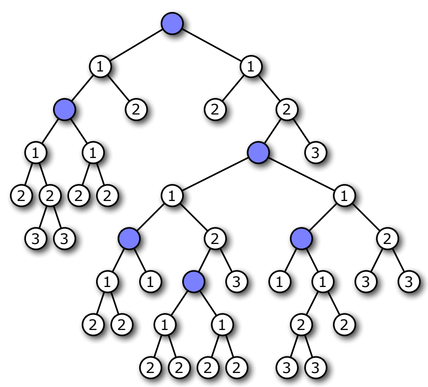
(c) Now suppose each edge of G has a weight. Define the weight of a cycle to be the total weight of its edges, and the weight of a *set* of cycles to be the total weight of all cycles in the set. (Thus, each edge is counted once for every cycle in which it appears.) Describe and analyze an efficient algorithm to compute the minimum-weight cycle basis of G.

4. Let T be a rooted binary tree with n vertices, and let $k \leq n$ be a positive integer. We would like to mark k vertices in T so that every vertex has a nearby marked ancestor. More formally, we define the *clustering cost* of a clustering of any subset K of vertices as

$$\mathit{cost}(K) = \max_{v} \mathit{cost}(v, K),$$

where the maximum is taken over all vertices v in the tree, and

$$\mathit{cost}(v, K) = \begin{cases} 0 & \text{if } v \in K \\ \infty & \text{if } v \text{ is the root of } T \text{ and } v \notin K \\ 1 + \mathit{cost}(\mathit{parent}(v)) & \text{otherwise} \end{cases}$$

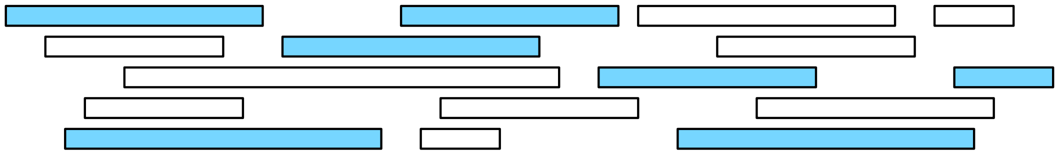


A subset of 5 vertices with clustering cost 3

Describe and analyze a dynamic-programming algorithm to compute the minimum clustering cost of any subset of k vertices in T. For full credit, your algorithm should run in $O(n^2k^2)$ time.

5. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The *size* of a tiling cover is just the number of intervals.

Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X. If you use a greedy algorithm, you must prove that it is correct.



A set of intervals. The seven shaded intervals form a tiling path.

CS 473G: Graduate Algorithms, Spring 2007
Homework 4

Due March 29, 2007

Please remember to submit **separate, individually stapled** solutions to each problem.

---

1. Given a graph G with edge weights and an integer k, suppose we wish to partition the the vertices of G into k subsets $S_1, S_2, \ldots, S_k$ so that the sum of the weights of the edges that cross the partition (*i.e.*, have endpoints in different subsets) is as large as possible.

   (a) Describe an efficient $(1 - 1/k)$-approximation algorithm for this problem.

   (b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for the new problem? Justify your answer.

2. In class, we saw a $(3/2)$-approximation algorithm for the metric traveling salesman problem. Here, we consider computing minimum cost Hamiltonian *paths*. Our input consists of a graph G whose edges have weights that satisfy the triangle inequality. Depending upon the problem, we are also given zero, one, or two endpoints.

   (a) If our input includes zero endpoints, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path.

   (b) If our input includes one endpoint u, describe a $(3/2)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u.

   (c) If our input includes two endpoints u and v, describe a $(5/3)$-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u and ends at v.

3. Consider the greedy algorithm for metric TSP: start at an arbitrary vertex u, and at each step, travel to the closest unvisited vertex.

   (a) Show that the greedy algorithm for metric TSP is an $O(\log n)$-approximation, where n is the number of vertices. *[Hint: Argue that the kth least expensive edge in the tour output by the greedy algorithm has weight at most $\mathrm{OPT}/(n - k + 1)$; try $k = 1$ and $k = 2$ first.]*

   *(b) **[Extra Credit]** Show that the greedy algorithm for metric TSP is no better than an $O(\log n)$-approximation.

4. In class, we saw that the greedy algorithm gives an $O(\log n)$-approximation for vertex cover. Show that our analysis of the greedy algorithm is asymptotically tight by describing, for any positive integer n, an n-vertex graph for which the greedy algorithm produces a vertex cover of size $\Omega(\log n) \cdot \mathrm{OPT}$.

5. Recall the minimum makespan scheduling problem: Given an array $T[1..n]$ of processing times for n jobs, we wish to schedule the jobs on m machines to minimize the time at which the last job terminates. In class, we proved that the greedy scheduling algorithm has an approximation ratio of at most 2.

   (a) Prove that for any set of jobs, the makespan of the greedy assignment is at most $(2-1/m)$ times the makespan of the optimal assignment.

   (b) Describe a set of jobs such that the makespan of the greedy assignment is exactly $(2-1/m)$ times the makespan of the optimal assignment.

   (c) Describe an efficient algorithm to solve the minimum makespan scheduling problem *exactly* if every processing time $T[i]$ is a power of two.

CS 473G: Graduate Algorithms, Spring 2007
Homework 5

Due Thursday, April 17, 2007

Please remember to submit **separate, individually stapled** solutions to each problem.

Unless a problem specifically states otherwise, you can assume the function $\textsc{Random}(k)$, which returns an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, k\}$, in $O(1)$ time. For example, to perform a fair coin flip, you would call $\textsc{Random}(2)$.

---

1. Suppose we want to write an efficient function $\textsc{RandomPermutation}(n)$ that returns a permutation of the integers $\langle 1, \ldots, n\rangle$ chosen uniformly at random.

   (a) What is the expected running time of the following $\textsc{RandomPermutation}$ algorithm?

   ```
   RANDOMPERMUTATION(n):
     for i ← 1 to n
         π[i] ← EMPTY
     for i ← 1 to n
         j ← RANDOM(n)
         while (π[j] ≠ EMPTY)
             j ← RANDOM(n)
         π[j] ← i
     return π
   ```

   (b) Consider the following partial implementation of $\textsc{RandomPermutation}$.

   ```
   RANDOMPERMUTATION(n):
     for i ← 1 to n
         A[i] ← RANDOM(n)
     π ← SOMEFUNCTION(A)
     return π
   ```

   Prove that if the subroutine $\textsc{SomeFunction}$ is deterministic, then this algorithm cannot be correct. *[Hint: There is a one-line proof.]*

   (c) Describe and analyze an $\textsc{RandomPermutation}$ algorithm whose expected worst-case running time is $O(n)$.

   *(d) **[Extra Credit]** Describe and analyze an $\textsc{RandomPermutation}$ algorithm that uses only fair coin flips; that is, your algorithm can't call $\textsc{Random}(k)$ with $k > 2$. Your algorithm should run in $O(n \log n)$ time with high probability.

2. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

   - MAKEQUEUE: Return a new priority queue containing the empty set.
   - FINDMIN(Q): Return the smallest element of Q (if any).
   - DELETEMIN(Q): Remove the smallest element in Q (if any).
   - INSERT(Q, x): Insert element x into Q, if it is not already there.
   - DECREASEKEY(Q, x, y): Replace an element $x \in Q$ with a smaller element y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q that contains x.
   - DELETE(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q that contains x.
   - MELD(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

   A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

```
MELD(Q1, Q2):
  if Q1 is empty, return Q2
  if Q2 is empty, return Q1
  if key(Q1) > key(Q2)
      swap Q1 ↔ Q2
  with probability 1/2
      left(Q1) ← MELD(left(Q1), Q2)
  else
      right(Q1) ← MELD(right(Q1), Q2)
  return Q1
```

   (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of MELD(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if each left/right choice is made with equal probability?]*

   (b) Prove that MELD(Q_1, Q_2) runs in $O(\log n)$ time with high probability.

   (c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

3. Prove that GUESSMINCUT returns the *second* smallest cut in its input graph with probability $\Omega(1/n^2)$. (The second smallest cut could be significantly larger than the minimum cut.)

4. A *heater* is a sort of dual treap, in which the priorities of the nodes are given by the user, but their search keys are random (specifically, independently and uniformly distributed in the unit interval $[0, 1]$).

   (a) Prove that for any r, the node with the rth smallest *priority* has expected depth $O(\log r)$.

   (b) Prove that an n-node heater has depth $O(\log n)$ with high probability.

   (c) Describe algorithms to perform the operations INSERT and DELETEMIN in a heater. What are the expected worst-case running times of your algorithms?

   You may assume all priorities and keys are distinct. *[Hint: Cite the relevant parts (but only the relevant parts!) of the treap analysis instead of repeating them.]*

5. Let n be an arbitrary positive integer. Describe a set $\mathcal{T}$ of binary search trees with the following properties:

   - Every tree in $\mathcal{T}$ has n nodes, which store the search keys $1, 2, 3, \ldots, n$.
   - For any integer k, if we choose a tree uniformly at random from $\mathcal{T}$, the expected depth of node k in that tree is $O(\log n)$.
   - Every tree in $\mathcal{T}$ has depth $\Omega(\sqrt{n})$.

   (This is why we had to prove via Chernoff bounds that the maximum depth of an n-node treap is $O(\log n)$ with high probability.)

★6. **[Extra Credit]** Recall that F_k denotes the kth Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_k = F_{k-1} + F_{k-2}$ for all $k \geq 2$. Suppose we are building a hash table of size $m = F_k$ using the hash function

$$h(x) = (F_{k-1} \cdot x) \bmod F_k$$

Prove that if the consecutive integers $0, 1, 2, \ldots, F_k - 1$ are inserted in order into an initially empty table, each integer is hashed into one of the largest contiguous empty intervals in the table. Among other things, this implies that there are no collisions.

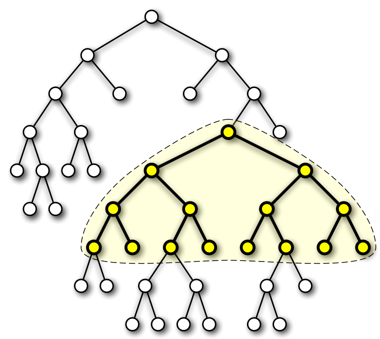
For example, when $m = 13$, the hash table is filled as follows.

0												
0								1				
0			2					1				
0			2					1			3	
0			2			4		1			3	
0	5		2			4		1			3	
0	5		2			4		1	6		3	
0	5		2	7		4		1	6		3	
0	5		2	7		4		1	6		3	8
0	5		2	7		4	9	1	6		3	8
0	5	10	2	7		4	9	1	6		3	8
0	5	10	2	7		4	9	1	6	11	3	8
0	5	10	2	7	12	4	9	1	6	11	3	8

You have 90 minutes to answer four of these questions.
Write your answers in the separate answer booklet.
You may take the question sheet with you when you leave.

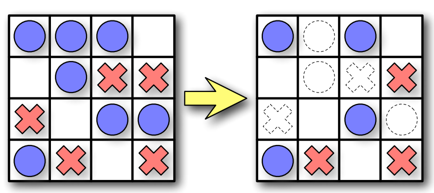
1. Recall that a binary tree is *complete* if every internal node has two children and every leaf has the same depth. An *internal subtree* of a binary tree is a connected subgraph, consisting of a node and some (possibly all or none) of its descendants.

   Describe and analyze an algorithm that computes the depth of the *largest complete internal subtree* of a given n-node binary tree. For full credit, your algorithm should run in $O(n)$ time.

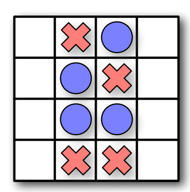


The largest complete internal subtree in this binary tree has depth 3.

2. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

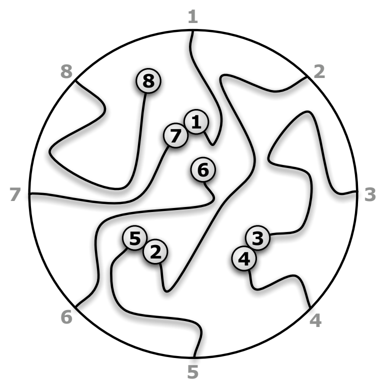
   Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

3. Suppose you are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k. Describe and analyze an algorithm to find the kth largest element in the union of A and B in $O(\log n)$ time. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 21], \qquad B[1..8] = [2, 4, 5, 8, 14, 17, 19, 20], \qquad k = 10,$$

   your algorithm should return 13. You can assume that the arrays contain no duplicates. *[Hint: What can you learn from comparing one element of A to one element of B?]*

4. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.



The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates.
The organizers must pay $M[3,4] + M[2,5] + M[1,7]$.

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1 \,..\, n, 1 \,..\, n]$ posted on the wall behind the Round Table, where $M[i,j] = M[j,i]$ is the reward to be paid if snails i and j meet.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

5. SUBSETSUM and PARTITION are two closely-related NP-hard problems.

- SUBSETSUM: Given a set X of positive integers and an integer t, determine whether there is a subset of X whose elements sum to t.
- PARTITION: Given a set X of positive integers, determine whether X can be partitioned into two subsets whose elements sum to the same value.

(a) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.

(b) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.

Don't forget to ***prove*** that your reductions are correct.

You have 120 minutes to answer four of these questions.
Write your answers in the separate answer booklet.
You may take the question sheet with you when you leave.

1. Consider the following algorithm for finding the smallest element in an unsorted array:

```
RandomMin(A[1..n]):
  min ← ∞
  for i ← 1 to n in random order
     if A[i] < min
        min ← A[i]   (⋆)
  return min
```

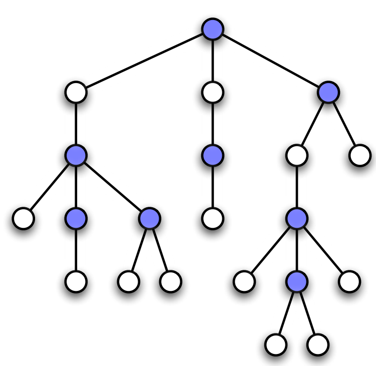
   (a) **[1 pt]** In the worst case, how many times does RandomMin execute line $(\star)$?

   (b) **[3 pts]** What is the probability that line $(\star)$ is executed during the last iteration of the for loop?

   (c) **[6 pts]** What is the *exact* expected number of executions of line $(\star)$? (A correct $\Theta()$ bound is worth 4 points.)

2. Describe and analyze an efficient algorithm to find the size of the smallest vertex cover of a given tree. That is, given a tree T, your algorithm should find the size of the smallest subset C of the vertices, such that every edge in T has at least one endpoint in C.

   The following hint may be helpful. Suppose C is a vertex cover that contains a leaf ℓ. If we remove ℓ from the cover and insert its parent, we get another vertex cover of the same size as C. Thus, there is a minimum vertex cover that includes none of the leaves of T (except when the tree has only one or two vertices).



A tree whose smallest vertex cover has size 8.

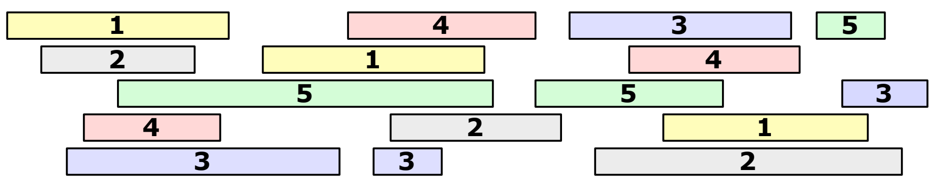
3. A *dominating set* for a graph G is a subset D of the vertices, such that every vertex in G is either in D or has a neighbor in D. The MINDOMINATINGSET problem asks for the size of the smallest dominating set for a given graph.

   Recall the MINSETCOVER problem from lecture. The input consists of a *ground set* X and a collection of subsets $S_1, S_2, \ldots, S_k \subseteq X$. The problem is to find the minimum number of subsets S_i that completely cover X. This problem is NP-hard, because it is a generalization of the vertex cover problem.

   (a) **[7 pts]** Describe a polynomial-time reduction from MINDOMINATINGSET to MINSETCOVER.

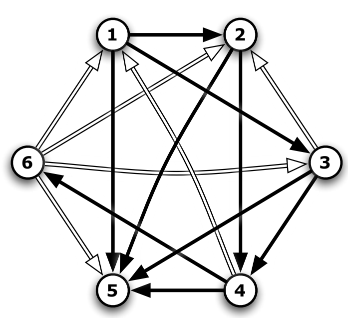
   (b) **[3 pts]** Describe a polynomial-time $O(\log n)$-approximation algorithm for MINDOMINATINGSET. *[Hint: There is a two-line solution.]*

4. Let X be a set of n intervals on the real line. A *proper coloring* of X assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Describe and analyze an efficient algorithm to compute the minimum number of colors needed to properly color X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, where $L[i]$ and $R[i]$ are the left and right endpoints of the ith interval. As usual, if you use a greedy algorithm, you must prove that it is correct.



A proper coloring of a set of intervals using five colors.

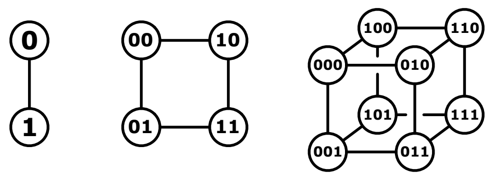
5. The *linear arrangement problem* asks, given an n-vertex directed graph as input, for an ordering $v_1, v_2, \ldots, v_n$ of the vertices that maximizes the number of *forward edges*: directed edges $v_i \to v_j$ such that $i < j$. Describe and analyze an efficient 2-approximation algorithm for this problem.



A directed graph with six vertices with nine forward edges (black) and six backward edges (white)

You have 180 minutes to answer six of these questions.
Write your answers in the separate answer booklet.

1. The d-dimensional hypercube is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if and only if their labels differ in exactly one bit.



The 1-dimensional, 2-dimensional, and 3-dimensional hypercubes.

   (a) *[8 pts]* Recall that a Hamiltonian cycle is a closed walk that visits each vertex in a graph exactly once. ***Prove*** that for all $d \geq 2$, the d-dimensional hypercube has a Hamiltonian cycle.

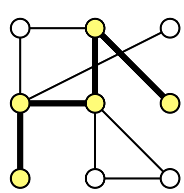
   (b) *[2 pts]* Recall that an Eulerian circuit is a closed walk that traverses each edge in a graph exactly once. Which hypercubes have an Eulerian circuit? *[Hint: This is very easy.]*

2. The University of Southern North Dakota at Hoople has hired you to write an algorithm to schedule their final exams. Each semester, USNDH offers n different classes. There are r different rooms on campus and t different time slots in which exams can be offered. You are given two arrays $E[1..n]$ and $S[1..r]$, where $E[i]$ is the number of students enrolled in the ith class, and $S[j]$ is the number of seats in the jth room. At most one final exam can be held in each room during each time slot. Class i can hold its final exam in room j only if $E[i] < S[j]$.

   Describe and analyze an efficient algorithm to assign a room and a time slot to each class (or report correctly that no such assignment is possible).

3. What is the *exact* expected number of leaves in an n-node treap? (The answer is obviously at most n, so no partial credit for writing "$O(n)$".) *[Hint: What is the probably that the node with the kth largest key is a leaf?]*

4. A *tonian path* in a graph G is a simple path in G that visits more than half of the vertices of G. (Intuitively, a tonian path is "most of a Hamiltonian path".) ***Prove*** that it is NP-hard to determine whether or not a given graph contains a tonian path.



A tonian path.

5. A *palindrome* is a string that reads the same forwards and backwards, like x, pop, noon, redivider, or amanaplanacatahamayakayamahatacanalpanama. Any string can be broken into sequence of palindromes. For example, the string bubbaseesabanana ('Bubba sees a banana.') can be broken into palindromes in several different ways; for example,

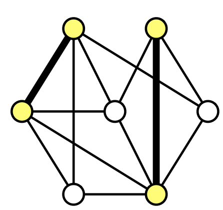
   bub + baseesab + anana
   b + u + bb + a + sees + aba + nan + a
   b + u + bb + a + sees + a + b + anana
   b + u + b + b + a + s + e + e + s + a + b + a + n + a + n + a

   Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string bubbaseesabanana, your algorithm would return the integer 3.

6. Consider the following modification of the 2-approximation algorithm for minimum vertex cover that we saw in class. The only real change is that we compute a set of edges instead of a set of vertices.

```
APPROXMINMAXMATCHING(G):
  M ← ∅
  while G has at least one edge
      (u, v) ← any edge in G
      G ← G \ {u, v}
      M ← M ∪ {(u, v)}
  return M
```

   (a) *[2 pts]* ***Prove*** that the output graph M is a *matching*—no pair of edges in M share a common vertex.
   (b) *[2 pts]* ***Prove*** that M is a *maximal* matching—M is not a proper subgraph of another matching in G.
   (c) *[6 pts]* ***Prove*** that M contains at most twice as many edges as the *smallest* maximal matching in G.



The smallest maximal matching in a graph.

7. Recall that in the standard maximum-flow problem, the flow through an edge is limited by the capacity of that edge, but there is no limit on how much flow can pass through a vertex. Suppose each vertex v in our input graph has a capacity $c(v)$ that limits the total flow through v, in addition to the usual edge capacities. Describe and analyze an efficient algorithm to compute the maximum (s,t)-flow with these additional constraints. *[Hint: Reduce to the standard max-flow problem.]*

CS 573: Graduate Algorithms, Fall 2008
Homework 0

Due **<u>in class</u>** at 12:30pm, Wednesday, September 3, 2008

Name:	
Net ID:	Alias:

☐ I understand the course policies.

- Each student must submit their own solutions for this homework. For all future homeworks, groups of up to three students may submit a single, common solution.

- Neatly print your full name, your NetID, and an alias of your choice in the boxes above, and staple this page to the front of your homework solutions. We will list homework and exam grades on the course web site by alias.

  Federal privacy law and university policy forbid us from publishing your grades, even anonymously, without your explicit written permission. ***By providing an alias, you grant us permission to list your grades on the course web site. If you do not provide an alias, your grades will not be listed.*** For privacy reasons, your alias should not resemble your name, your NetID, your university ID number, or (God forbid) your Social Security number.

- Please carefully read the course policies linked from the course web site. If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup. Once you understand the policies, please check the box at the top of this page. In particular:
  - You may use any source at your disposal—paper, electronic, or human—but you ***must*** write your solutions in your own words, and you ***must*** cite every source that you use.
  - Unless explicitly stated otherwise, ***every*** homework problem requires a proof.
  - Answering "I don't know" to any homework or exam problem is worth 25% partial credit.
  - Algorithms or proofs containing phrases like "and so on" or "repeat this for all n", instead of an explicit loop, recursion, or induction, will receive 0 points.

- This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, discrete probability, graphs, and most importantly, induction—to help you identify gaps in your background knowledge. **You are responsible for filling those gaps.** The early chapters of any algorithms textbook should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks. If you need help, please ask in office hours and/or on the course newsgroup.

---

1. (a) **[5 pts]** Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Assume reasonable but nontrivial base cases. If your solution requires a particular base case, say so.

$$A(n) = 4A(n/8) + \sqrt{n}$$
$$B(n) = B(n/3) + 2B(n/4) + B(n/6) + n$$
$$C(n) = 6C(n-1) - 9C(n-2)$$
$$D(n) = \max_{n/3<k<2n/3} \big(D(k) + D(n-k) + n\big)$$
$$E(n) = \big(E(\sqrt{n})\big)^2 \cdot n$$

(b) **[5 pts]** Sort the functions in the box from asymptotically smallest to asymptotically largest, indicating ties if there are any. **Do *not* turn in proofs**—just a sorted list of 16 functions—but you should do them anyway, just for practice. We use the notation $\lg n = \log_2 n$.

n	$\lg n$	$\sqrt{n}$	3^n
$\sqrt{\lg n}$	$\lg\sqrt{n}$	$3^{\sqrt{n}}$	$\sqrt{3^n}$
$3^{\lg n}$	$\lg(3^n)$	$3^{\lg\sqrt{n}}$	$3^{\sqrt{\lg n}}$
$\sqrt{3^{\lg n}}$	$\lg(3^{\sqrt{n}})$	$\lg\sqrt{3^n}$	$\sqrt{\lg(3^n)}$

2. Describe and analyze a data structure that stores set of n records, each with a numerical *key* and a numerical *priority*, such that the following operation can be performed quickly:

- RANGETOP(a, z) : return the highest-priority record whose key is between a and z.

For example, if the (key, priority) pairs are

$$(3,1),\ (4,9),\ (9,2),\ (6,3),\ (5,8),\ (7,5),\ (1,10),\ (0,7),$$

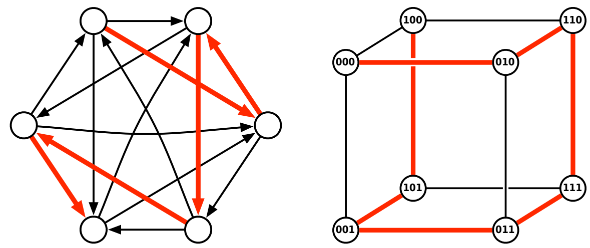
then RANGETOP$(2,8)$ would return the record with key 4 and priority 9 (the second in the list).

Analyze both the size of your data structure and the running time of your RANGETOP algorithm. For full credit, your space and time bounds must both be as small as possible. You may assume that no two records have equal keys or equal priorities, and that no record has a or z as its key. *[Hint: How would you compute the number of keys between a and z? How would you solve the problem if you knew that a is always $-\infty$?]*

3. A *Hamiltonian path* in G is a path that visits every vertex of G exactly once. In this problem, you are asked to prove that two classes of graphs always contain a Hamiltonian path.

   (a) **[5 pts]** A *tournament* is a directed graph with exactly one edge between each pair of vertices. (Think of the nodes in a round-robin tournament, where edges represent games, and each edge points from the loser to the winner.) Prove that every tournament contains a *directed* Hamiltonian path.

   (b) **[5 pts]** Let d be an arbitrary non-negative integer. The d-dimensional *hypercube* is the graph defined as follows. There are 2^d vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if and only if their labels differ in exactly one bit. Prove that the d-dimensional hypercube contains a Hamiltonian path.



Hamiltonian paths in a 6-node tournament and a 3-dimensional hypercube.

4. Penn and Teller agree to play the following game. Penn shuffles a standard deck[1] of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3♣), at which point the remaining undrawn cards instantly burst into flames.

   The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.[2] To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

   (a) What is the expected number of cards that Teller draws?

   (b) What is the expected *maximum* value among the cards Teller gives to Penn?

   (c) What is the expected *minimum* value among the cards Teller gives to Penn?

   (d) What is the expected number of cards that Teller gives to Penn?

   Full credit will be given only for *exact* answers (with correct proofs, of course). *[Hint: Let* $13 = n$*.]*

[1]In a standard deck of playing cards, each card has a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$ and a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$; each of the 52 possible suit-value pairs appears in the deck exactly once. Actually, to make the game more interesting, Penn and Teller normally use razor-sharp ninja throwing cards.

[2]Specifically, he hurls them from the opposite side of the stage directly into the back of Penn's right hand. Ouch!

5. (a) The ***Fibonacci numbers F_n*** are defined by the recurrence $F_n = F_{n-1} + F_{n-2}$, with base cases $F_0 = 0$ and $F_1 = 1$. Here are the first several Fibonacci numbers:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

Prove that any non-negative integer can be written as the sum of distinct, non-consecutive Fibonacci numbers. That is, if the Fibonacci number F_i appears in the sum, it appears exactly once, and its neighbors F_{i-1} and F_{i+1} do not appear at all. For example:

$$17 = F_7 + F_4 + F_2, \qquad 42 = F_9 + F_6, \qquad 54 = F_9 + F_7 + F_5 + F_3 + F_1.$$

(b) The Fibonacci sequence can be extended backward to negative indices by rearranging the defining recurrence: $F_n = F_{n+2} - F_{n+1}$. Here are the first several negative-index Fibonacci numbers:

F_{-10}	F_{-9}	F_{-8}	F_{-7}	F_{-6}	F_{-5}	F_{-4}	F_{-3}	F_{-2}	F_{-1}
−55	34	−21	13	−8	5	−3	2	−1	1

Prove that $F_{-n} = -F_n$ if and only if n is even.

(c) Prove that *any* integer—positive, negative, or zero—can be written as the sum of distinct, non-consecutive Fibonacci numbers *with negative indices*. For example:

$$17 = F_{-7} + F_{-5} + F_{-2}, \qquad -42 = F_{-10} + F_{-7}, \qquad 54 = F_{-9} + F_{-7} + F_{-5} + F_{-3} + F_{-1}.$$

[Hint: Zero is both non-negative and even. Don't use weak induction!]

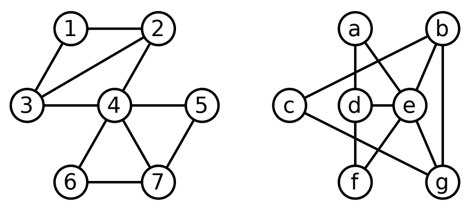
CS 573: Graduate Algorithms, Fall 2008
Homework 1

Due at 11:59:59pm, Wednesday, September 17, 2008

For this and all future homeworks, groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and alias (if any) of every group member on the first page of your submission.

---

1. Two graphs are said to be ***isomorphic*** if one can be transformed into the other just by relabeling the vertices. For example, the graphs shown below are isomorphic; the left graph can be transformed into the right graph by the relabeling $(1,2,3,4,5,6,7) \mapsto (c,g,b,e,a,f,d)$.



Two isomorphic graphs.

Consider the following related decision problems:

- GraphIsomorphism: Given two graphs G and H, determine whether G and H are isomorphic.
- EvenGraphIsomorphism: Given two graphs G and H, such that every vertex in G and H has even degree, determine whether G and H are isomorphic.
- SubgraphIsomorphism: Given two graphs G and H, determine whether G is isomorphic to a subgraph of H.

(a) Describe a polynomial-time reduction from EvenGraphIsomorphism to GraphIsomorphism.

(b) Describe a polynomial-time reduction from GraphIsomorphism to EvenGraphIsomorphism.

(c) Describe a polynomial-time reduction from GraphIsomorphism to SubgraphIsomorphism.

(d) Prove that SubgraphIsomorphism is NP-complete.

(e) What can you conclude about the NP-hardness of GraphIsomorphism? Justify your answer.

[Hint: These are all easy!]

2. (a) A *tonian path* in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-complete.

   (b) A *tonian cycle* in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-complete. *[Hint: Use part (a).]*

3. The following variant of 3Sat is called either Exact3Sat or 1in3Sat, depending on who you ask.

   Given a boolean formula in conjunctive normal form with 3 literals per clause, is there an assignment that makes ***exactly one*** literal in each clause True?

   Prove that this problem is NP-complete.

4. Suppose you are given a magic black box that can solve the MAXCLIQUE problem ***in polynomial time***. That is, given an arbitrary graph G as input, the magic black box computes the number of vertices in the largest complete subgraph of G. Describe and analyze a ***polynomial-time*** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

5. A boolean formula in ***exclusive-or conjunctive normal form*** (XCNF) is a conjunction (AND) of several *clauses*, each of which is the *exclusive*-or of several literals. The XCNF-SAT problem asks whether a given XCNF boolean formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-complete.

*6. ***[Extra credit]*** Describe and analyze an algorithm to solve 3SAT in $O(\phi^n \operatorname{poly}(n))$ time, where $\phi = (1+\sqrt{5})/2 \approx 1.618034$. *[Hint: Prove that in each recursive call, either you have just eliminated a pure literal, or the formula has a clause with at most two literals. What recurrence leads to this running time?]*

[0] In class, I asserted that Gaussian elimination was probably discovered by Gauss, in violation of Stigler's Law of Eponymy. In fact, a method very similar to Gaussian elimination appears in the Chinese treatise *Nine Chapters on the Mathematical Art*, believed to have been finalized before 100AD, although some material may predate emperor Qin Shi Huang's infamous 'burning of the books and burial of the scholars' in 213BC. The great Chinese mathematician Liu Hui, in his 3rd-century commentary on *Nine Chapters*, compares two variants of the method and counts the number of arithmetic operations used by each, with the explicit goal of find the more efficient method. This is arguably the earliest recorded *analysis* of any algorithm.

CS 573: Graduate Algorithms, Fall 2008
Homework 2

Due at 11:59:59pm, Wednesday, October 1, 2008

- For this and all future homeworks, groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and alias (if any) of every group member on the first page of your submission.
- We will use the following point breakdown to grade dynamic programming algorithms: 60% for a correct recurrence (including base cases), 20% for correct running time analysis of the memoized recurrence, 10% for correctly transforming the memoized recursion into an iterative algorithm.
- A greedy algorithm *must* be accompanied by a proof of correctness in order to receive *any* credit.

---

1. (a) Let $X[1..m]$ and $Y[1..n]$ be two arbitrary arrays of numbers. A ***common supersequence*** of X and Y is another sequence that contains both X and Y as subsequences. Describe and analyze an efficient algorithm to compute the function $scs(X, Y)$, which gives the length of the *shortest* common supersequence of X and Y.

   (b) Call a sequence $X[1..n]$ of numbers ***oscillating*** if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Describe and analyze an efficient algorithm to compute the function $los(X)$, which gives the length of the longest oscillating subsequence of an arbitrary array X of integers.

   (c) Call a sequence $X[1..n]$ of numbers ***accelerating*** if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Describe and analyze an efficient algorithm to compute the function $lxs(X)$, which gives the length of the longest accelerating subsequence of an arbitrary array X of integers.

2. A *palindrome* is a string that reads the same forwards and backwards, like x, pop, noon, redivider, or amanaplanacatahamayakayamahatacanalpanama. Any string can be broken into sequence of palindromes. For example, the string bubbaseesabanana ('Bubba sees a banana.') can be broken into palindromes in several different ways; for example:

   bub + baseesab + anana
   b + u + bb + a + sees + aba + nan + a
   b + u + bb + a + sees + a + b + anana
   b + u + b + b + a + s + e + e + s + a + b + a + n + a + n + a

   Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string bubbaseesabanana, your algorithm would return the integer 3.

3. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \operatorname{poly}(n))$ time. Given an undirected n-vertex graph G with weighted edges, your algorithm should return the weight of the lightest Hamiltonian cycle in G, or ∞ if G has no Hamiltonian cycles. *[Hint: The obvious recursive algorithm takes $O(n!)$ time.]*

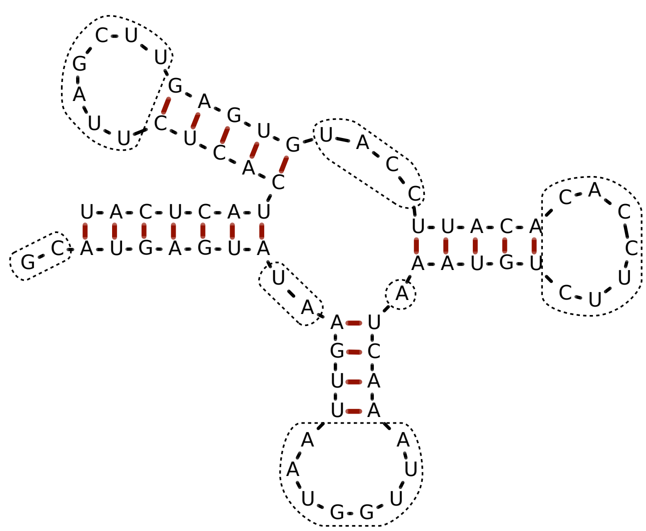
4. Ribonucleic acid (RNA) molecules are long chains of millions of nucleotides or *bases* of four different types: adenine (A), cytosine (C), guanine (G), and uracil (U). The *sequence* of an RNA molecule is a string $b[1..n]$, where each character $b[i] \in \{A, C, G, U\}$ corresponds to a base. In addition to the chemical bonds between adjacent bases in the sequence, hydrogen bonds can form between certain pairs of bases. The set of bonded base pairs is called the *secondary structure* of the RNA molecule.

   We say that two base pairs (i, j) and (i', j') with $i < j$ and $i' < j'$ ***overlap*** if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most base pairs are non-overlapping. Overlapping base pairs create so-called *pseudoknots* in the secondary structure, which are essential for some RNA functions, but are more difficult to predict.

   Suppose we want to predict the best possible secondary structure for a given RNA sequence. We will adopt a drastically simplified model of secondary structure:

   - Each base can be paired with at most one other base.
   - Only A-U pairs and C-G pairs can bond.
   - Pairs of the form $(i, i+1)$ and $(i, i+2)$ cannot bond.
   - Overlapping base pairs cannot bond.

   The last restriction allows us to visualize RNA secondary structure as a sort of fat tree.



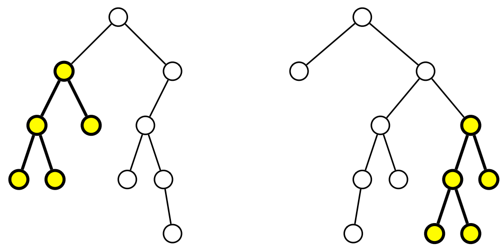
Example RNA secondary structure with 21 base pairs, indicated by heavy red lines.
Gaps are indicated by dotted curves. This structure has score $2^2 + 2^2 + 8^2 + 1^2 + 7^2 + 4^2 + 7^2 = 187$

   (a) Describe and analyze an algorithm that computes the maximum possible *number* of base pairs in a secondary structure for a given RNA sequence.

   (b) A *gap* in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the *score* of a secondary structure to be the sum of the *squares* of the gap lengths.[1] Describe and analyze an algorithm that computes the minimum possible score of a secondary structure for a given RNA sequence.

[1]This score function has absolutely no connection to reality; I just made it up. Real RNA structure prediction requires much more complicated scoring functions.

5. A *subtree* of a (rooted, ordered) binary tree T consists of a node and all its descendants. Design and analyze an efficient algorithm to compute the ***largest common subtree*** of two given binary trees T_1 and T_2; this is the largest subtree of T_1 that is isomorphic to a subtree in T_2. The contents of the nodes are irrelevant; we are only interested in matching the underlying combinatorial structure.



Two binary trees, with their largest common subtree emphasized

*6. ***[Extra credit]*** Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. A ***digital subsequence*** of D is an sequence of positive integers composed in the usual way from disjoint substrings of D. For example, $3, 4, 5, 6, 23, 38, 62, 64, 83, 279$ is an increasing digital subsequence of the first several digits of π:

$$\boxed{3}, 1, \boxed{4}, 1, \boxed{5}, 9, \boxed{6}, \boxed{2,3}, 4, \boxed{3,8}, 4, \boxed{6,2}, \boxed{6,4}, 3, 3, \boxed{8,3}, \boxed{2,7,9}$$

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the previous example has length 10.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D. *[Hint: Be careful about your computational assumptions. How long does it take to compare two k-digit numbers?]*

CS 573: Graduate Algorithms, Fall 2008
Homework 3

Due at 11:59:59pm, Wednesday, October 22, 2008

- Groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and the HW0 alias (if any) of every group member on the first page of your submission.

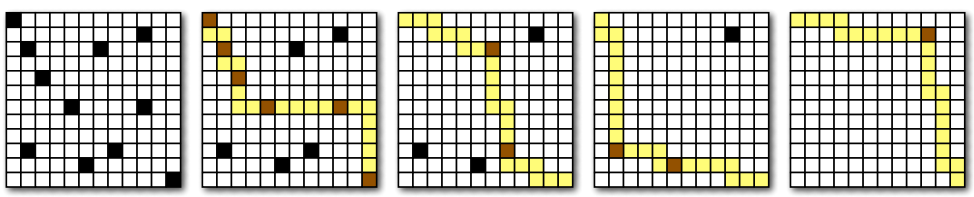
---

1. Consider an $n \times n$ grid, some of whose cells are marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. We want to compute the minimum number of monotone paths that cover all marked cells. The input to our problem is an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \text{True}$ if and only if cell (i, j) is marked.

   One of your friends suggests the following greedy strategy:

   - Find (somehow) one "good" path π that covers the maximum number of marked cells.
   - Unmark the cells covered by π.
   - If any cells are still marked, recursively cover them.

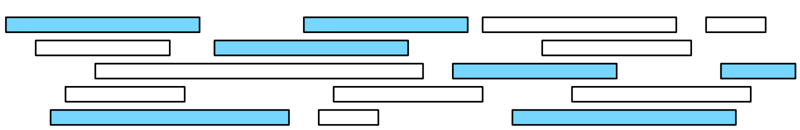
   Does this greedy strategy always compute an optimal solution? If yes, give a proof. If no, give a counterexample.



Greedily covering the marked cells in a grid with four monotone paths.

2. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a ***tiling path*** if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The *size* of a tiling path is just the number of intervals.

   Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X. If you use a greedy algorithm, you must prove that it is correct.



A set of intervals. The seven shaded intervals form a tiling path.

3. Given a graph G with edge weights and an integer k, suppose we wish to partition the vertices of G into k subsets $S_1, S_2, \ldots, S_k$ so that the sum of the weights of the edges that cross the partition (*i.e.*, that have endpoints in different subsets) is as large as possible.

   (a) Describe an efficient $(1-1/k)$-approximation algorithm for this problem. *[Hint: Solve the special case $k = 2$ first.]*

   (b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for this new problem? Justify your answer.

4. Consider the following heuristic for constructing a vertex cover of a connected graph G: ***Return the set of all non-leaf nodes of any depth-first spanning tree.*** (Recall that a depth-first spanning tree is a *rooted* tree; the root is not considered a leaf, even if it has only one neighbor in the tree.)

   (a) Prove that this heuristic returns a vertex cover of G.

   (b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G.

   (c) Prove that for any $\varepsilon > 0$, there is a graph for which this heuristic returns a vertex cover of size at least $(2-\varepsilon)\cdot OPT$.

5. Consider the following greedy approximation algorithm to find a vertex cover in a graph:

```
GreedyVertexCover(G):
  C ← ∅
  while G has at least one edge
      v ← vertex in G with maximum degree
      G ← G \ v
      C ← C ∪ v
  return C
```

   In class we proved that the approximation ratio of this algorithm is $O(\log n)$; your task is to prove a matching lower bound. Specifically, for any positive integer n, describe an n-vertex graph G such that GreedyVertexCover(G) returns a vertex cover that is $\Omega(\log n)$ times larger than optimal. *[Hint: $H_n = \Omega(\log n)$.]*

*6. ***[Extra credit]*** Consider the greedy algorithm for metric TSP: Start at an arbitrary vertex u, and at each step, travel to the closest unvisited vertex.

   (a) Prove that this greedy algorithm is an $O(\log n)$-approximation algorithm, where n is the number of vertices. *[Hint: Show that the kth least expensive edge in the tour output by the greedy algorithm has weight at most $OPT/(n-k+1)$; try $k = 1$ and $k = 2$ first.]*

   *(b) Prove that the greedy algorithm for metric TSP is no better than an $O(\log n)$-approximation. That is, describe an infinite family of weighted graphs that satisfy the triangle inequality, such that the greedy algorithm returns a cycle whose length is $\Omega(\log n)$ times the optimal TSP tour.

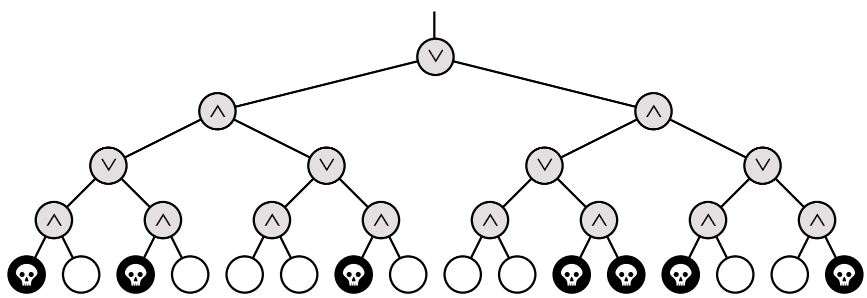
CS 573: Graduate Algorithms, Fall 2008
Homework 4

Due at 11:59:59pm, Wednesday, October 31, 2008

- Groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and the HW0 alias (if any) of every group member on the first page of your submission.
- Unless a problem explicitly states otherwise, you can assume the existence of a function RANDOM(k), which returns an integer uniformly distributed in the range $\{1, 2, \ldots, k\}$ in $O(1)$ time; the argument k must be a positive integer. For example, RANDOM(2) simulates a fair coin flip, and RANDOM(1) *always* returns 1.

---

1. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black leaves stand represent TRUE and FALSE inputs, respectively. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

   (a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

   (b) Unfortunately, Death won't let you even look at every node in the tree. Describe and analyze a *randomized* algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*

   (c) ***[Extra credit]*** Describe and analyze a *randomized* algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. *[Hint: You may not need to change your algorithm at all.]*

2. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

   What is the *exact* expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. *[Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?]*

3. (a) Prove that the expected number of proper descendants of any node in a treap is *exactly* equal to the expected depth of that node.

   (b) Why doesn't the Chernoff-bound argument for depth imply that, with high probability, *every* node in a treap has $O(\log n)$ descendants? The conclusion is obviously bogus—every n-node treap has one node with exactly n descendants!—but what is the flaw in the argument?

   (c) What is the expected number of leaves in an n-node treap? *[Hint: What is the probability that in an n-node treap, the node with kth smallest search key is a leaf?]*

4. The following randomized algorithm, sometimes called "one-armed quicksort", selects the rth smallest element in an unsorted array $A[1..n]$. For example, to find the smallest element, you would call RANDOMSELECT$(A, 1)$; to find the median element, you would call RANDOMSELECT$(A, \lfloor n/2 \rfloor)$. The subroutine PARTITION$(A[1..n], p)$ splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n-1$ comparisons altogether, and returns the new index of the pivot element.

```
RandomSelect(A[1..n], r):
  k ← Partition(A[1..n], Random(n))
  if r < k
      return RandomSelect(A[1..k−1], r)
  else if r > k
      return RandomSelect(A[k+1..n], r−k)
  else
      return A[k]
```

   (a) State a recurrence for the expected running time of RANDOMSELECT, as a function of n and r.

   (b) What is the *exact* probability that RANDOMSELECT compares the ith smallest and jth smallest elements in the input array? The correct answer is a simple function of i, j, and r. *[Hint: Check your answer by trying a few small examples.]*

   (c) Show that for any n and r, the expected running time of RANDOMSELECT is $\Theta(n)$. You can use either the recurrence from part (a) or the probabilities from part (b).

   *(d) ***[Extra Credit]*** Find the *exact* expected number of comparisons executed by RANDOMSELECT, as a function of n and r.

5. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

   - MAKEQUEUE: Return a new priority queue containing the empty set.
   - FINDMIN(Q): Return the smallest element of Q (if any).
   - DELETEMIN(Q): Remove the smallest element in Q (if any).
   - INSERT(Q, x): Insert element x into Q, if it is not already there.
   - DECREASEKEY(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
   - DELETE(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
   - MELD(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

   A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

```
MELD(Q1, Q2):
  if Q1 is empty return Q2
  if Q2 is empty return Q1
  if key(Q1) > key(Q2)
      swap Q1 ↔ Q2
  with probability 1/2
      left(Q1) ← MELD(left(Q1), Q2)
  else
      right(Q1) ← MELD(right(Q1), Q2)
  return Q1
```

   (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (*not* just those constructed by the operations listed above), the expected running time of MELD(Q_1, Q_2) is $O(\log n)$, where n is the total number of nodes in both trees. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if each left/right choice is made with equal probability?]*

   (b) Prove that MELD(Q_1, Q_2) runs in $O(\log n)$ time with high probability. *[Hint: You don't need Chernoff bounds, but you might use the identity $\binom{ck}{k} \le (ce)^k$.]*

   (c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ time with high probability.)

*6. ***[Extra credit]*** In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0,1]$, but in practice, computers only have access to random *bits*. This problem asks you to analyze a modification of treaps that takes this limitation into account.

Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1..\infty]$ of random bits, which is interpreted as the rational number

$$priority(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or 'reveal') new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LargerPriority(v, w):
  for i ← 1 to ∞
      if i > ℓ_v
          ℓ_v ← i;  π_v[i] ← RandomBit
      if i > ℓ_w
          ℓ_w ← i;  π_w[i] ← RandomBit
      if π_v[i] > π_w[i]
          return v
      else if π_v[i] < π_w[i]
          return w
```

Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LargerPriority during these insertions.

(a) Prove that $E[L] = \Theta(n)$.

(b) Prove that $E[\ell_v] = \Theta(1)$ for any node v. *[Hint: This is equivalent to part (a). Why?]*

(c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. *[Hint: Why doesn't this contradict part (b)?]*

CS 573: Graduate Algorithms, Fall 2008
Homework 5

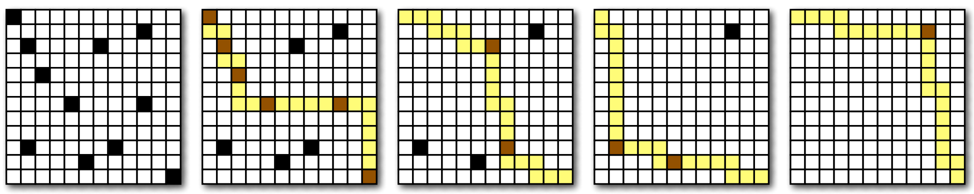
Due at 11:59:59pm, Wednesday, November 19, 2008

- Groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name, NetID, and the HW0 alias (if any) of every group member on the first page of your submission.

---

1. Recall the following problem from Homework 3: You are given an $n \times n$ grid, some of whose cells are marked; the grid is represented by an array $M[1..n, 1..n]$ of booleans, where $M[i,j] = \text{TRUE}$ if and only if cell (i,j) is marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell.

   Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell.



Greedily covering the marked cells in a grid with four monotone paths.

2. Suppose we are given a directed graph $G = (V, E)$, two vertices s an t, and a capacity function $c : V \to \mathbb{R}^+$. A flow f is *feasible* if the total flow into every vertex v is at most $c(v)$:

$$\sum_u f(u \to v) \le c(v) \quad \text{for every vertex } v.$$

   Describe and analyze an efficient algorithm to compute a feasible flow of maximum value.

3. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

   Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

4. *Ad-hoc networks* are made up of cheap, low-powered wireless devices. In principle[1], these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other situations where people might want to monitor conditions in hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be dropped into the area from an airplane (for instance), and then they would somehow automatically configure themselves into an efficiently functioning wireless network.

   The devices can communication only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D.

   We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit the information it has to some other *backup* device within its communication range. To improve reliability, we require each device x to have k potential backup devices, all within distance D of x; we call these k devices the ***backup set*** of x. Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

   So suppose we are given the communication radius D, parameters b and k, and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j. Describe an algorithm that either computes a backup set of size k for each of the n devices, such that that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

5. Let $G = (V, E)$ be a directed graph where for each vertex v, the in-degree and out-degree of v are equal. Let u and v be two vertices G, and suppose G contains k edge-disjoint paths from u to v. Under these conditions, must G also contain k edge-disjoint paths from v to u? Give a proof or a counterexample with explanation.

*6. ***[Extra credit]*** A *rooted tree* is a directed acyclic graph, in which every vertex has exactly one incoming edge, except for the *root*, which has no incoming edges. Equivalently, a rooted tree consists of a root vertex, which has edges pointing to the roots of zero or more smaller rooted trees. Describe a polynomial-time algorithm to compute, given two rooted trees A and B, the largest common rooted subtree of A and B.

   [Hint: Let $LCS(u, v)$ denote the largest common subtree whose root in A is u and whose root in B is v. Your algorithm should compute $LCS(u, v)$ for all vertices u and v using dynamic programming. This would be easy if every vertex had $O(1)$ children, and still straightforward if the children of each node were ordered from left to right and the common subtree had to respect that ordering. But for unordered trees with large degree, you need another trick to combine recursive subproblems efficiently. Don't waste your time trying to reduce the polynomial running time.]

[1]but not really in practice

CS 573: Graduate Algorithms, Fall 2008
Homework 6

Practice only

- This homework is only for practice; do not submit solutions. At least one (sub)problem (or something *very* similar) will appear on the final exam.

---

1. An *integer program* is a linear program with the additional constraint that the variables must take only integer values.

   (a) Prove that deciding whether an integer program has a feasible solution is NP-complete.

   (b) Prove that finding the optimal solution to an integer program is NP-hard.

   [Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]

2. Describe precisely how to dualize a linear program written in general form:

$$\begin{aligned} \text{maximize} \ & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} \ & \sum_{j=1}^{d} a_{ij} x_j \le b_i \quad \text{for each } i = 1 \,..\, p \\ & \sum_{j=1}^{d} a_{ij} x_j = b_i \quad \text{for each } i = p+1 \,..\, p+q \\ & \sum_{j=1}^{d} a_{ij} x_j \ge b_i \quad \text{for each } i = p+q+1 \,..\, n \end{aligned}$$

   Keep the number of dual variables as small as possible. The dual of the dual of any linear program should be *syntactically identical* to the original linear program.

3. Suppose you have a subroutine that can solve linear programs in polynomial time, but only if they are both feasible and bounded. Describe an algorithm that solves *arbitrary* linear programs in polynomial time, using this subroutine as a black box. Your algorithm should return an optimal solution if one exists; if no optimum exists, your algorithm should report that the input instance is UNBOUNDED or INFEASIBLE, whichever is appropriate. *[Hint: Add one constraint to guarantee boundedness; add one variable to guarantee feasibility.]*

4. Suppose you are given a set P of n points in some high-dimensional space $\mathbb{R}^d$, each labeled either *black* or *white*. A *linear classifier* is a d-dimensional vector c with the following properties:

   - If p is a black point, then $p \cdot c > 0$.
   - If p is a while point, then $p \cdot c < 0$.

   Describe an efficient algorithm to find a linear classifier for the given data points, or correctly report that none exists. *[Hint: This is almost trivial, but not quite.]*

Lots more linear programming problems can be found at http://www.ee.ucla.edu/ee236a/homework/problems.pdf. Enjoy!

You have 120 minutes to answer all five questions.
Write your answers in the separate answer booklet.
Please turn in your question sheet and your cheat sheet with your answers.

1. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, several cards are dealt face up in a long row. Then you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. Each card is worth a different number of points. The player that collects the most points wins the game.

   Like most eight-year-olds who haven't studied algorithms, Elmo follows the obvious greedy strategy every time he plays: ***Elmo <u>always</u> takes the card with the higher point value.*** Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

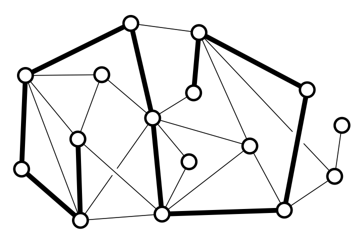
   (a) Describe an initial sequence of cards that allows you to win against Elmo, no matter who moves first, but *only* if you do *not* follow Elmo's greedy strategy.

   (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

   Here is a sample game, where both you and Elmo are using the greedy strategy. Elmo wins 8–7. You cannot win this particular game, no matter what strategy you use.

Initial cards	2	4	5	1	3
Elmo takes the 3	2	4	5	1	~~3~~
You take the 2	~~2~~	4	5	1	
Elmo takes the 4		~~4~~	5	1	
You take the 5			~~5~~	1	
Elmo takes the 1				~~1~~	

2. ***Prove*** that the following problem is NP-hard: Given an undirected graph G, find the longest path in G whose length is a multiple of 5.



This graph has a path of length 10, but no path of length 15.

3. Suppose you are given an array $A[1..n]$ of integers. Describe and analyze an algorithm that finds the largest sum of of elements in a contiguous subarray $A[i..j]$.

   For example, if the array A contains the numbers $[-6, 12, -7, 0, 14, -7, 5]$, your algorithm should return the number 19:

$$\boxed{-6}\,\underbrace{\boxed{12}\,\boxed{-7}\,\boxed{0}\,\boxed{14}}_{19}\,\boxed{-7}\,\boxed{5}$$

4. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, '`bananaananas`' is a shuffle of '`banana`' and '`ananas`' in several different ways:

$$\texttt{banana}{}_{\texttt{ananas}} \qquad \texttt{ban}{}_{\texttt{ana}}{}^{\texttt{ana}}{}_{\texttt{nas}} \qquad \texttt{b}{}_{\texttt{an}}{}^{\texttt{an}}{}_{\texttt{a}}{}^{\texttt{a}}{}_{\texttt{na}}{}^{\texttt{na}}{}_{\texttt{s}}$$

   The strings '`prodgyrnamammiincg`' and '`dyprongarmammicing`' are both shuffles of '`dynamic`' and '`programming`':

$$\texttt{pro}{}^{\texttt{d}}\texttt{g}{}^{\texttt{y}}\texttt{r}{}^{\texttt{nam}}\texttt{ammi}{}^{\texttt{i}}\texttt{n}{}^{\texttt{c}}\texttt{g} \qquad {}^{\texttt{dy}}\texttt{pro}{}^{\texttt{n}}\texttt{g}{}^{\texttt{a}}\texttt{r}{}^{\texttt{m}}\texttt{amm}{}^{\texttt{ic}}\texttt{ing}$$

   Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

5. Suppose you are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B in $\Theta(\log(m+n))$ time. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..5] = [2, 5, 8, 17, 19] \qquad k = 6$$

   your algorithm should return 8. You can assume that the arrays contain no duplicates. An algorithm that works only in the special case $n = m = k$ is worth 7 points.

   [Hint: What can you learn from comparing one element of A to one element of B?]

You have 120 minutes to answer all five questions.
Write your answers in the separate answer booklet.
Please turn in your question sheet and your cheat sheet with your answers.

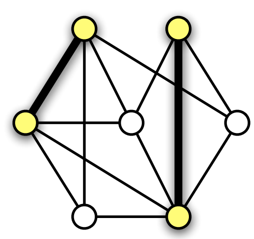
1. Consider the following modification of the 'dumb' 2-approximation algorithm for minimum vertex cover that we saw in class. The only change is that we output a set of edges instead of a set of vertices.

```
ApproxMinMaxMatching(G):
  M ← ∅
  while G has at least one edge
      let (u, v) be any edge in G
      remove u and v (and their incident edges) from G
      add (u, v) to M
  return M
```

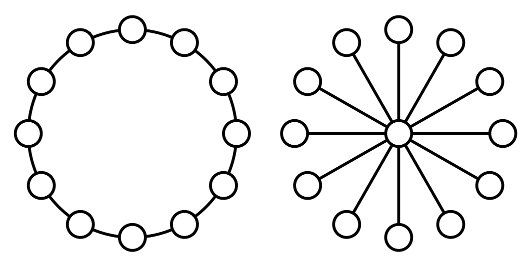
   (a) ***Prove*** that this algorithm computes a *matching*—no two edges in M share a common vertex.

   (b) ***Prove*** that M is a *maximal* matching—M is not a proper subgraph of another matching in G.

   (c) ***Prove*** that M contains at most twice as many edges as the *smallest* maximal matching in G.



The smallest maximal matching in a graph.



A cycle and a star.

2. Consider the following heuristic for computing a small vertex cover of a graph.

   - Assign a random *priority* to each vertex, chosen independently and uniformly from the real interval $[0, 1]$ (just like treaps).
   - Mark every vertex that does *not* have larger priority than *all* of its neighbors.

   For any graph G, let $OPT(G)$ denote the size of the smallest vertex cover of G, and let $M(G)$ denote the number of nodes marked by this algorithm.

   (a) ***Prove*** that the set of vertices marked by this heuristic is *always* a vertex cover.

   (b) Suppose the input graph G is a *cycle*, that is, a connected graph where every vertex has degree 2. What is the expected value of $M(G)/OPT(G)$? ***Prove*** your answer is correct.

   (c) Suppose the input graph G is a *star*, that is, a tree with one central vertex of degree $n-1$. What is the expected value of $M(G)/OPT(G)$? ***Prove*** your answer is correct.

3. Suppose we want to write an efficient function SHUFFLE($A[1..n]$) that randomly permutes the array A, so that each of the $n!$ permutations is equally likely.

   (a) ***Prove*** that the following SHUFFLE algorithm is ***not*** correct. *[Hint: There is a two-line proof.]*

```
SHUFFLE(A[1..n]):
  for i = 1 to n
      swap A[i] ⟷ A[RANDOM(n)]
```

   (b) Describe and analyze a correct SHUFFLE algorithm whose expected running time is $O(n)$.
   Your algorithm may call the function RANDOM(k), which returns an integer uniformly distributed in the range $\{1, 2, \ldots, k\}$ in $O(1)$ time. For example, RANDOM(2) simulates a fair coin flip, and RANDOM(1) *always* returns 1.

4. Let Φ be a legal input for 3SAT—a boolean formula in conjunctive normal form, with exactly three literals in each clause. Recall that an assignment of boolean values to the variables in Φ ***satisfies*** a clause if at least one of its literals is TRUE. The ***maximum satisfiability problem***, sometimes called MAX3SAT, asks for the maximum number of clauses that can be simultaneously satisfied by a single assignment. Solving MAXSAT exactly is clearly also NP-hard; this problem asks about approximation algorithms.

   (a) Let $\mathit{MaxSat}(\Phi)$ denote the maximum number of clauses that can be simultaneously satisfied by one variable assignment. Suppose we randomly assign each variable in Φ to be TRUE or FALSE, each with equal probability. ***Prove*** that the expected number of satisfied clauses is at least $\frac{7}{8}\mathit{MaxSat}(\Phi)$.

   (b) Let $\mathit{MinUnsat}(\Phi)$ denote the *minimum* number of clauses that can be simultaneously *unsatisfied* by a single assignment. ***Prove*** that it is NP-hard to approximate $\mathit{MinUnsat}(\Phi)$ within a factor of $10^{10^{100}}$.

5. Consider the following randomized algorithm for generating biased random bits. The subroutine FAIRCOIN returns either 0 or 1 with equal probability; the random bits returned by FAIRCOIN are mutually independent.

```
ONEINTHREE:
  if FAIRCOIN = 0
      return 0
  else
      return 1 − ONEINTHREE
```

   (a) ***Prove*** that ONEINTHREE returns 1 with probability 1/3.

   (b) What is the *exact* expected number of times that this algorithm calls FAIRCOIN? ***Prove*** your answer is correct.

   (c) Now suppose you are *given* a subroutine ONEINTHREE that generates a random bit that is equal to 1 with probability 1/3. Describe a FAIRCOIN algorithm that returns either 0 or 1 with equal probability, using ONEINTHREE as a subroutine. **Your *only* source of randomness is ONEINTHREE; in particular, you may not use the RANDOM function from problem 3.**

   (d) What is the *exact* expected number of times that your FAIRCOIN algorithm calls ONEINTHREE? ***Prove*** your answer is correct.

You have 180 minutes to answer all seven questions.
Write your answers in the separate answer booklet.
You can keep everything except your answer booklet when you leave.

1. An *integer program* is a linear program with the additional constraint that the variables must take only integer values. ***Prove*** that deciding whether an integer program has a feasible solution is NP-complete. *[Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]*

2. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A *heater* is a priority search tree in which the priorities are given by the user, and the search keys are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is the 'opposite' of a treap.

   The following problems consider an n-node heater T whose node priorities are the integers from 1 to n. We identify nodes in T by their priorities; thus, 'node 5' means the node in T with priority 5. The min-heap property implies that node 1 is the root of T. Finally, let i and j be integers with $1 \le i < j \le n$.

   (a) ***Prove*** that in a random permutation of the $(i+1)$-element set $\{1, 2, \ldots, i, j\}$, elements i and j are adjacent with probability $2/(i+1)$.

   (b) ***Prove*** that node i is an ancestor of node j with probability $2/(i+1)$. *[Hint: Use part (a)!]*

   (c) What is the probability that node i is a *descendant* of node j? *[Hint: **Don't** use part (a)!]*

   (d) What is the *exact* expected depth of node j?

3. The UIUC Faculty Senate has decided to convene a committee to determine whether Chief Illiniwek should become the official ~~mascot~~ *symbol* of the University of Illinois Global Campus. Exactly one faculty member must be chosen from each academic department to serve on this committee. Some faculty members have appointments in multiple departments, but each committee member will represent only one department. For example, if Prof. Blagojevich is affiliated with both the Department of Corruption and the Department of Stupidity, and he is chosen as the Stupidity representative, then someone else must represent Corruption. Finally, University policy requires that any committee on virtual ~~mascots~~ *symbols* must contain the same number of assistant professors, associate professors, and full professors. Fortunately, the number of departments is a multiple of 3.

   Describe an efficient algorithm to select the membership of the Global Illiniwek Committee. Your input is a list of all UIUC faculty members, their ranks (assistant, associate, or full), and their departmental affiliation(s). There are n faculty members and $3k$ departments.

4. Let $\alpha(G)$ denote the number of vertices in the largest independent set in a graph G. ***Prove*** that the following problem is NP-hard: Given a graph G, return *any* integer between $\alpha(G) - 31337$ and $\alpha(G) + 31337$.

5. Let $G = (V, E)$ be a directed graph with capacities $c \colon E \to \mathbb{R}^+$, a source vertex s, and a target vertex t. Suppose someone hands you a function $f : E \to \mathbb{R}$. Describe and analyze a fast algorithm to determine whether f is a maximum (s,t)-flow in G.

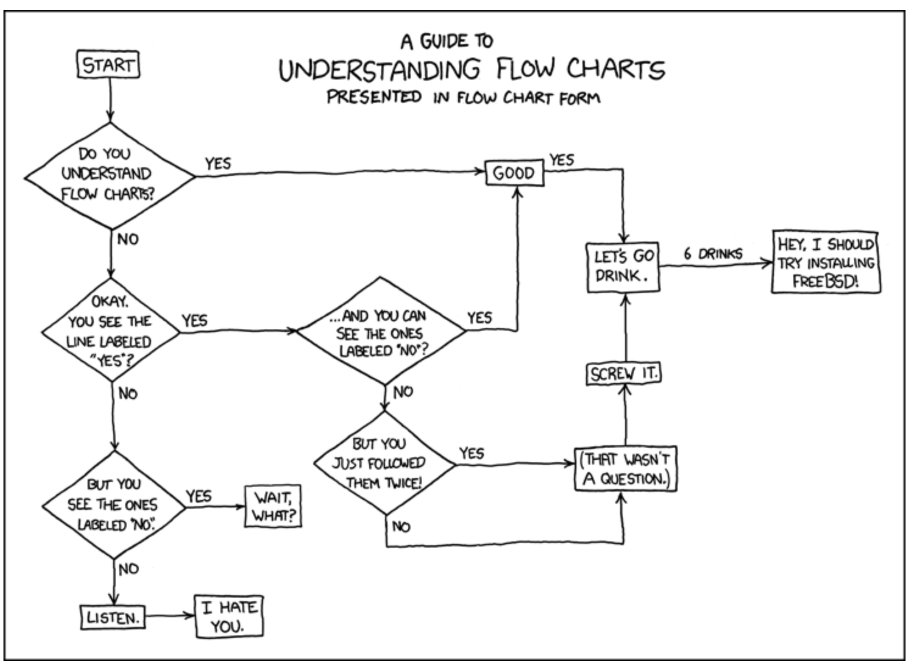
6. For some strange reason, you decide to ride your bicycle 3688 miles from Urbana to Wasilla, Alaska, to join in the annual Wasilla Mining Festival and Helicopter Wolf Hunt. The festival starts exactly 32 days from now, so you need to bike an average of 109 miles each day. Because you are a poor starving student, you can only afford to sleep at campgrounds, which are unfortunately *not* spaced exactly 109 miles apart. So some days you will have to ride more than average, and other days less, but you would like to keep the variation as small as possible. You settle on a formal scoring system to help decide where to sleep; if you ride x miles in one day, your score for that day is $(109 - x)^2$. What is the minimum possible total score for all 32 days?

   More generally, suppose you have D days to travel DP miles, there are n campgrounds along your route, and your score for traveling x miles in one day is $(x - P)^2$. You are given a sorted array $dist[1..n]$ of real numbers, where $dist[i]$ is the distance from your starting location to the ith campground; it may help to also set $dist[0] = 0$ and $dist[n+1] = DP$. Describe and analyze a fast algorithm to compute the minimum possible score for your trip. The running time of your algorithm should depend on the integers D and n, but not on the real number P.

7. Describe and analyze efficient algorithms for the following problems.

   (a) Given a set of n integers, does it contain elements a and b such that $a + b = 0$?

   (b) Given a set of n integers, does it contain elements a, b, and c such that $a + b = c$?



— Randall Munroe, *xkcd*, December 17, 2008 (http://xkcd.com/518/)

CS 473: Undergraduate Algorithms, Spring 2009
Homework 0

Due **in class** at 11:00am, Tuesday, January 27, 2009

- This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, graphs, and most importantly, induction—to help you identify gaps in your background knowledge. **You are responsible for filling those gaps.** The early chapters of any algorithms textbook should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks. If you need help, please ask in office hours and/or on the course newsgroup.
- Each student must submit individual solutions for this homework. For all future homeworks, groups of up to three students may submit a single, common solution.
- Please carefully read the course policies linked from the course web site. If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup. In particular:
  - Submit five separately stapled solutions, one for each numbered problem, with your name and NetID clearly printed on each page. Please do not staple everything together.
  - You may use any source at your disposal—paper, electronic, or human—but you ***must*** write your solutions in your own words, and you ***must*** cite every source that you use.
  - Unless explicitly stated otherwise, ***every*** homework problem requires a proof.
  - Answering "I don't know" to any homework or exam problem (except for extra credit problems) is worth 25% partial credit.
  - Algorithms or proofs containing phrases like "and so on" or "repeat this process for all n", instead of an explicit loop, recursion, or induction, will receive 0 points.

  Write the sentence "I understand the course policies." at the top of your solution to problem 1.

---

1. Professor George O'Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character **&**. Preorder and postorder traversals of the tree visit the nodes in the following order:

   - Preorder: **I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X**
   - Postorder: **H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I**

   (a) List the nodes in George's tree in the order visited by an inorder traversal.

   (b) Draw George's tree.

2. (a) **[5 pts]** Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Assume reasonable but nontrivial base cases. If your solution requires a particular base case, say so. ***Do not submit proofs***—just a list of five functions—but you should do them anyway, just for practice.

$$A(n) = 10A(n/5) + n$$

$$B(n) = 2B\left(\left\lceil \frac{n+3}{4} \right\rceil\right) + 5n^{6/7} - 8\sqrt{\frac{n}{\log n}} + 9\left\lfloor \log^{10} n \right\rfloor - 11$$

$$C(n) = 3C(n/2) + C(n/3) + 5C(n/6) + n^2$$

$$D(n) = \max_{0<k<n} (D(k) + D(n-k) + n)$$

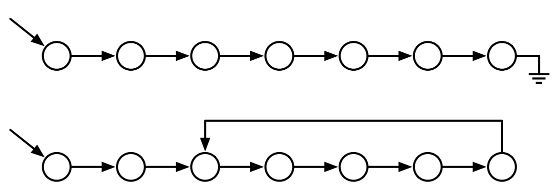
$$E(n) = \frac{E(n-1)\,E(n-3)}{E(n-2)}$$ *[Hint: Write out the first 20 terms.]*

(b) **[5 pts]** Sort the following functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. ***Do not submit proofs***—just a sorted list of 16 functions—but you should do them anyway, just for practice.

Write $f(n) \ll g(n)$ to indicate that $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. We use the notation $\lg n = \log_2 n$.

n	$\lg n$	$\sqrt{n}$	3^n
$\sqrt{\lg n}$	$\lg \sqrt{n}$	$3^{\sqrt{n}}$	$\sqrt{3^n}$
$3^{\lg n}$	$\lg(3^n)$	$3^{\lg \sqrt{n}}$	$3^{\sqrt{\lg n}}$
$\sqrt{3^{\lg n}}$	$\lg(3^{\sqrt{n}})$	$\lg \sqrt{3^n}$	$\sqrt{\lg(3^n)}$

3. Suppose you are given a pointer to the head of singly linked list. Normally, each node in the list has a pointer to the next element, and the last node's pointer is NULL. Unfortunately, your list might have been corrupted (by a bug in *somebody else's* code, of course), so that some node's pointer leads back to an earlier node in the list.



Top: A standard singly-linked list. Bottom: A corrupted singly linked list.

Describe an algorithm[1] that determines whether the linked list is corrupted or not. Your algorithm must not modify the list. For full credit, your algorithm should run in $O(n)$ time, where n is the number of nodes in the list, and use $O(1)$ extra space (not counting the list itself).

[1] Since you understand the course policies, you know what this phrase means. Right?

4. (a) Prove that any integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example:

$$
\begin{aligned}
42 &= 3^4 - 3^3 - 3^2 - 3^1 \\
25 &= 3^3 - 3^1 + 3^0 \\
17 &= 3^3 - 3^2 - 3^0
\end{aligned}
$$

(b) Prove that any integer (positive, negative, or zero) can be written in the form $\sum_i (-2)^i$, where the exponents i are distinct non-negative integers. For example:

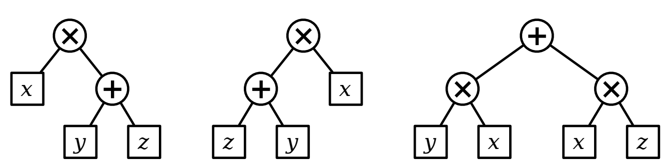
$$
\begin{aligned}
42 &= (-2)^6 + (-2)^5 + (-2)^4 + (-2)^0 \\
25 &= (-2)^6 + (-2)^5 + (-2)^3 + (-2)^0 \\
17 &= (-2)^4 + (-2)^0
\end{aligned}
$$

*[Hint: Don't use weak induction. In fact, **never** use weak induction.]*

5. An ***arithmetic expression tree*** is a binary tree where every leaf is labeled with a variable, every internal node is labeled with an arithmetic operation, and every internal node has exactly two children. For this problem, assume that the only allowed operations are $+$ and $\times$. Different leaves may or may not represent different variables.

Every arithmetic expression tree represents a function, transforming input values for the leaf variables into an output value for the root, by following two simple rules: (1) The value of any $+$-node is the sum of the values of its children. (2) The value of any $\times$-node is the product of the values of its children.

Two arithmetic expression trees are ***equivalent*** if they represent the same function; that is, the same input values for the leaf variables always leads to the same output value at both roots. An arithmetic expression tree is in ***normal form*** if the parent of every $+$-node (if any) is another $+$-node.



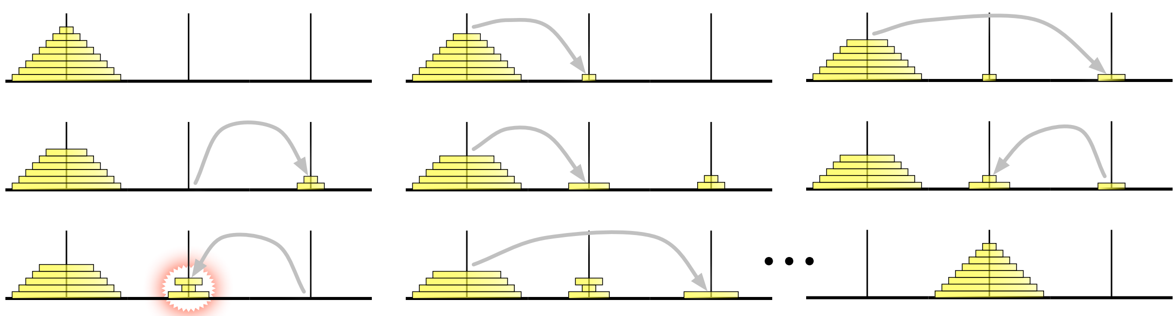
Three equivalent expression trees. Only the third is in normal form.

Prove that for any arithmetic expression tree, there is an equivalent arithmetic expression tree in normal form.

*6. ***[Extra credit]*** You may be familiar with the story behind the famous Tower of Hanoï puzzle:

> At the great temple of Benares, there is a brass plate on which three vertical diamond shafts are fixed. On the shafts are mounted n golden disks of decreasing size. At the time of creation, the god Brahma placed all of the disks on one pin, in order of size with the largest at the bottom. The Hindu priests unceasingly transfer the disks from peg to peg, one at a time, never placing a larger disk on a smaller one. When all of the disks have been transferred to the last pin, the universe will end.

Recently the temple at Benares was relocated to southern California, where the monks are considerably more laid back about their job. At the "Towers of Hollywood", the golden disks have been replaced with painted plywood, and the diamond shafts have been replaced with Plexiglas. More importantly, the restriction on the order of the disks has been relaxed. While the disks are being moved, the bottom disk on any pin must be the *largest* disk on that pin, but disks further up in the stack can be in any order. However, after all the disks have been moved, they must be in sorted order again.



The Towers of Hollywood. The sixth move leaves the disks out of order.

Describe an algorithm that moves a stack of n disks from one pin to the another using the smallest possible number of moves. *Exactly* how many moves does your algorithm perform? *[Hint: The Hollywood monks can bring about the end of the universe considerably faster than their Benaresian counterparts.]*

CS 473: Undergraduate Algorithms, Spring 2009
Homework 1

Due Tuesday, February 3, 2009 at 11:59:59pm.

- Groups of up to three students may submit a single, common solution for this and all future homeworks. Please clearly write every group member's name and NetID on every page of your submission.

1. The traditional Devonian/Cornish drinking song "The Barley Mow" has the following pseudolyrics, where *container*$[i]$ is the name of a container that holds 2^i ounces of beer. One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. (Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.)

```
BarleyMow(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for i ← 1 to n
      "We'll drink it out of the container[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The container[j],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

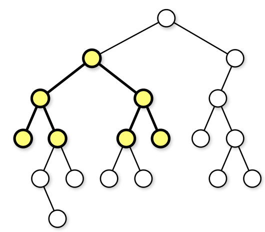
   (a) Suppose each container name *container*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.) *[Hint: Is 'barley-mow' one word or two? Does it matter?]*

   (b) If you want to sing this song for $n > 20$, you'll have to make up your own container names. To avoid repetition, these names will get progressively longer as n increases[1]. Suppose *container*$[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BarleyMow(n)? (Give a tight asymptotic bound.)

   (c) Suppose each time you mention the name of a container, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *container*$[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BarleyMow(n)? (Give an *exact* answer, not just an asymptotic bound.)

[1] "We'll drink it out of the hemisemidemiyottapint, boys!"

2. For this problem, a *subtree* of a binary tree means any connected subgraph; a binary tree is *complete* if every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

3. (a) Describe and analyze a recursive algorithm to reconstruct a binary tree, given its preorder and postorder node sequences (as in Homework 0, problem 1).

   (b) Describe and analyze a recursive algorithm to reconstruct a binary tree, given its preorder and *inorder* node sequences.

CS 473: Undergraduate Algorithms, Spring 2009
Homework 10

Due Tuesday, May 5, 2009 at 11:59:59pm

- Groups of up to three students may submit a single, common solution. Please clearly write every group member's name and NetID on every page of your submission.
- ***This homework is optional.*** If you submit solutions, they will be graded, and your overall homework grade will be the average of ten homeworks (Homeworks 0–10, dropping the lowest). If you do not submit solutions, your overall homework grade will be the average of *nine* homeworks (Homeworks 0–9, dropping the lowest).

---

1. Suppose you are given a magic black box that can determine ***in polynomial time***, given an arbitrary graph G, the number of vertices in the largest complete subgraph of G. Describe and analyze a ***polynomial-time*** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

2. PLANARCIRCUITSAT is a special case of CIRCUITSAT where the input circuit is drawn 'nicely' in the plane — no two wires cross, no two gates touch, and each wire touches only the gates it connects. (Not every circuit can be drawn this way!) As in the general CIRCUITSAT problem, we want to determine if there is an input that makes the circuit output TRUE?

   Prove that PLANARCIRCUITSAT is NP-complete. *[Hint:* XOR.*]*

3. For each problem below, either describe a polynomial-time algorithm or prove that the problem is NP-complete.

   (a) A *double-Eulerian* circuit in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G, does G have a *double-Eulerian* circuit?

   (b) A *double-Hamiltonian* circuit in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G, does G have a *double-Hamiltonian* circuit?

CS 473: Undergraduate Algorithms, Spring 2009
Homework 2

Written solutions due Tuesday, February 10, 2009 at 11:59:59pm.

- Roughly 1/3 of the students will give oral presentations of their solutions to the TAs. ***Please check Compass to check whether you are supposed give an oral presentation for this homework.*** Please see the course web page for further details.
- Groups of up to three students may submit a common solution. Please clearly write every group member's name and NetID on every page of your submission.
- Please start your solution to each numbered problem on a new sheet of paper. Please *don't* staple solutions for different problems together.
- ***For this homework only:*** These homework problems ask you to describe recursive backtracking algorithms for various problems. ***Don't*** use memoization or dynamic programming to make your algorithms more efficient; you'll get to do that on HW3. ***Don't*** analyze the running times of your algorithms. The ***only*** things you should submit for each problem are (1) a description of your recursive algorithm, and (2) a *brief* justification for its correctness.

---

1. A ***basic arithmetic expression*** is composed of characters from the set $\{1, +, \times\}$ and parentheses. Almost every integer can be represented by more than one basic arithmetic expression. For example, all of the following basic arithmetic expression represent the integer 14:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1+1)\times(1+1+1+1+1))+((1+1)\times(1+1))$$
$$(1+1)\times(1+1+1+1+1+1+1)$$
$$(1+1)\times(((1+1+1)\times(1+1))+1)$$

   Describe a recursive algorithm to compute, given an integer n as input, the minimum number of 1's in a basic arithmetic expression whose value is n. The number of parentheses doesn't matter, just the number of 1's. For example, when $n = 14$, your algorithm should return 8, for the final expression above.

2. A sequence $A = \langle a_1, a_2, \ldots, a_n\rangle$ is ***bitonic*** if there is an index i with $1 < i < n$, such that the prefix $\langle a_1, a_2, \ldots, a_i\rangle$ is strictly increasing and the suffix $\langle a_i, a_{i+1}, \ldots, a_n\rangle$ is strictly decreasing. In particular, a bitonic sequence must contain at least three elements.

   Describe a recursive algorithm to compute, given a sequence A, the length of the longest bitonic subsequence of A.

3. A ***palindrome*** is a string that reads the same forwards and backwards, like x, pop, noon, redivider, or amanaplanacatahamayakayamahatacanalpanama. Any string can be broken into sequence of palindromes. For example, the string bubbaseesabanana ('Bubba sees a banana.') can be broken into palindromes in several different ways; for example:

   bub + baseesab + anana
   b + u + bb + a + sees + aba + nan + a
   b + u + bb + a + sees + a + b + anana
   b + u + b + b + a + s + e + e + s + a + b + a + n + a + n + a

   Describe a recursive algorithm to compute the minimum number of palindromes that make up a given input string. For example, given the input string bubbaseesabanana, your algorithm would return the integer 3.

CS 473: Undergraduate Algorithms, Spring 2009
Homework 3

Written solutions due Tuesday, February 17, 2009 at 11:59:59pm.

---

1. Redo Homework 2, but now with dynamic programming!

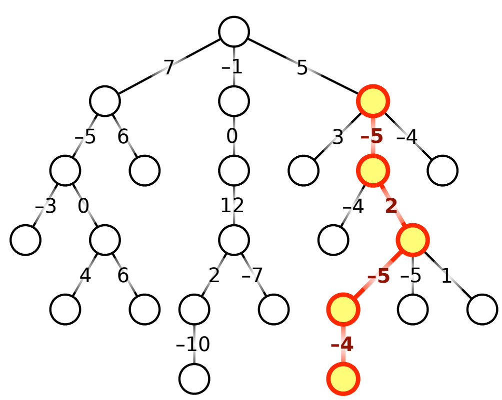
   (a) Describe and analyze an efficient algorithm to compute the minimum number of 1's in a basic arithmetic expression whose value is a given positive integer.

   (b) Describe and analyze an efficient algorithm to compute the length of the longest bitonic subsequence of a given input sequence.

   (c) Describe and analyze an efficient algorithm to compute the minimum number of palindromes that make up a given input string.

   Please see Homework 2 for more detailed descriptions of each problem. ***Solutions for Homework 2 will be posted Friday, after the HW2 oral presentations.*** You may (and should!) use anything from those solutions without justification.

2. Let T be a rooted tree with integer weights on its edges, which could be positive, negative, or zero. Design an algorithm to find the minimum-length path from a node in T down to one of its descendants. The length of a path is the sum of the weights of its edges. For example, given the tree shown below, your algorithm should return the number -12. For full credit, your algorithm should run in $O(n)$ time.



The minimum-weight downward path in this tree has weight -12.

3. Describe and analyze an efficient algorithm to compute the longest common subsequence of *three* given strings. For example, given the input strings `EPIDEMIOLOGIST`, `REFRIGERATION`, and `SUPERCALIFRAGILISTICEXPIALODOCIOUS`, your algorithm should return the number 5, because the longest common subsequence is **`EIEIO`**.

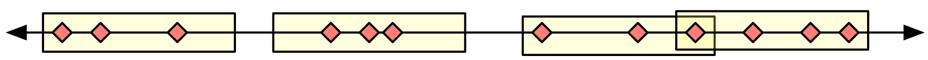
CS 473: Undergraduate Algorithms, Spring 2009
Homework 3½

Practice only

---

1. After graduating from UIUC, you are hired by a mobile phone company to plan the placement of new cell towers along a long, straight, nearly-deserted highway out west. Each cell tower can transmit the same fixed distance from its location. Federal law requires that any building along the highway must be within the broadcast range of at least one tower. On the other hand, your company wants to build as few towers as possble. Given the locations of the buildings, where should you build the towers?

   More formally, suppose you are given a set $X = \{x_1, x_2, \ldots, x_n\}$ of points on the real number line. Describe an algorithm to compute the minimum number of intervals of length 1 that can cover all the points in X. For full credit, your algorithm should run in $O(n \log n)$ time.



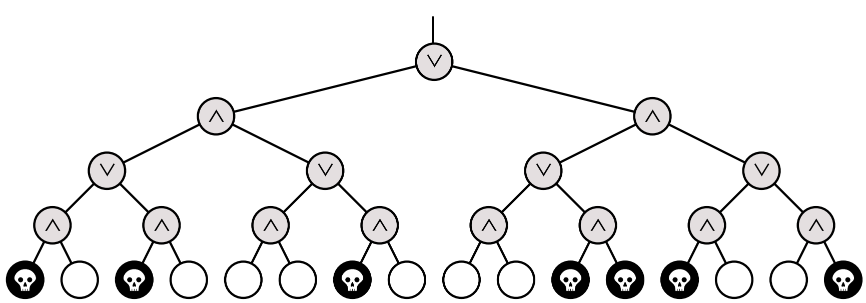
A set of points that can be covered by four unit intervals.

2. (a) The *left spine* of a binary tree is a path starting at the root and following only left-child pointers down to a leaf. What is the expected number of nodes in the left spine of an n-node treap?

   (b) What is the expected number of leaves in an n-node treap? *[Hint: What is the probability that in an n-node treap, the node with kth smallest search key is a leaf?]*

   (c) Prove that the expected number of proper descendants of any node in a treap is exactly equal to the expected depth of that node.

3. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for TRUE and FALSE. If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead.

(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

(b) Unfortunately, Death won't give you enough time to look at every node in the tree. Describe a *randomized* algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*

*(c) Describe a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. *[Hint: You may not need to change your algorithm from part (b) at all!]*

CS 473: Undergraduate Algorithms, Spring 2009
Homework 3

Written solutions due Tuesday, March 2, 2009 at 11:59:59pm.

---

1. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

   - MAKEQUEUE: Return a new priority queue containing the empty set.
   - FINDMIN(Q): Return the smallest element of Q (if any).
   - DELETEMIN(Q): Remove the smallest element in Q (if any).
   - INSERT(Q, x): Insert element x into Q, if it is not already there.
   - DECREASEKEY(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
   - DELETE(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
   - MELD(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

   A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

   ```
   MELD(Q1, Q2):
     if Q1 is empty return Q2
     if Q2 is empty return Q1

     if key(Q1) > key(Q2)
         swap Q1 ↔ Q2

     with probability 1/2
         left(Q1) ← MELD(left(Q1), Q2)
     else
         right(Q1) ← MELD(right(Q1), Q2)

     return Q1
   ```

   (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (*not* just those constructed by the operations listed above), the expected running time of MELD(Q_1, Q_2) is $O(\log n)$, where n is the total number of nodes in both trees. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if each left/right choice is made with equal probability?]*

   (b) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ expected time.)

2. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A *heater* is a priority search tree in which the priorities are given by the user, and the search keys are distributed uniformly and independently at random in the real interval $[0,1]$. Intuitively, a heater is the 'opposite' of a treap.

   The following problems consider an n-node heater T whose node priorities are the integers from 1 to n. We identify nodes in T by their priorities; thus, 'node 5' means the node in T with priority 5. The min-heap property implies that node 1 is the root of T. Finally, let i and j be integers with $1 \le i < j \le n$.

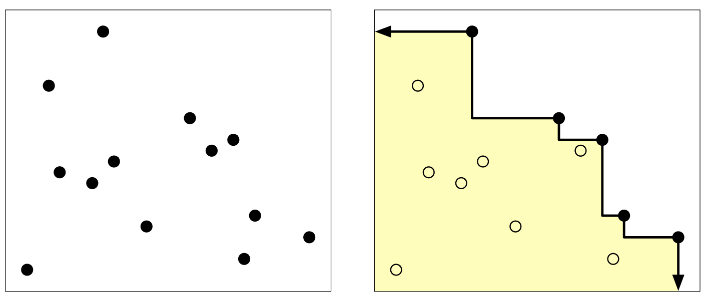
   (a) ***Prove*** that in a random permutation of the $(i+1)$-element set $\{1, 2, \ldots, i, j\}$, elements i and j are adjacent with probability $2/(i+1)$.

   (b) ***Prove*** that node i is an ancestor of node j with probability $2/(i+1)$. *[Hint: Use part (a)!]*

   (c) What is the probability that node i is a *descendant* of node j? *[Hint: **Don't** use part (a)!]*

   (d) What is the *exact* expected depth of node j?

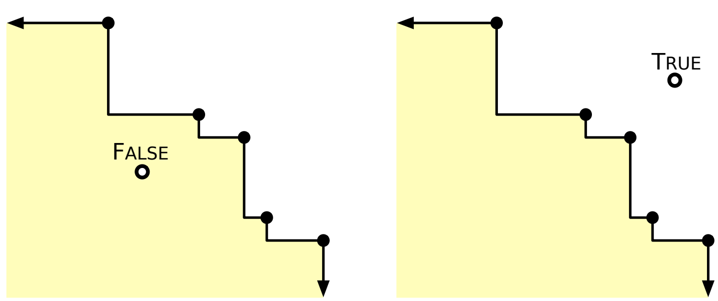
3. Let P be a set of n points in the plane. The *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.



A set of points in the plane and its staircase (shaded).

   (a) Describe an algorithm to compute the staircase of a set of n points in $O(n \log n)$ time.

   (b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm ABOVE?(x, y) that returns TRUE if the point (x, y) is above the staircase, or FALSE otherwise. Your data structure should use $O(n)$ space, and your ABOVE? algorithm should run in $O(\log n)$ time.



Two staircase queries.

CS 473: Undergraduate Algorithms, Spring 2009
Homework 5

Written solutions due Tuesday, March 9, 2009 at 11:59:59pm.

---

1. Remember the difference between stacks and queues? Good.

   (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The *only* access you have to the stacks is through the standard methods PUSH and POP.

   (b) A *quack* is an abstract data type that combines properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

   - **Push:** add a new item to the left end of the list;
   - **Pop:** remove the item on the left end of the list;
   - **Pull:** remove the item on the right end of the list.

   Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any push, pop, or pull operation is $O(1)$. Again, you are *only* allowed to access the stacks through the standard methods PUSH and POP.

2. In a *dirty* binary search tree, each node is labeled either *clean* or *dirty*. The lazy deletion scheme used for scapegoat trees requires us to *purge* the search tree, keeping all the clean nodes and deleting all the dirty nodes, as soon as half the nodes become dirty. In addition, the purged tree should be perfectly balanced.

   Describe and analyze an algorithm to purge an *arbitrary* n-node dirty binary search tree in $O(n)$ time, using at most $O(\log n)$ space (in addition to the tree itself). Don't forget to include the recursion stack in your space bound. An algorithm that uses $\Theta(n)$ additional space in the worst case is worth half credit.

3. Some applications of binary search trees attach a *secondary data structure* to each node in the tree, to allow for more complicated searches. Maintaining these secondary structures usually complicates algorithms for keeping the top-level search tree balanced.

   Let T be an arbitrary binary tree. Suppose every node v in T stores a secondary structure of size $O(size(v))$, which can be built in $O(size(v))$ time, where $size(v)$ denotes the number of descendants of v. Performing a rotation at any node v now requires $O(size(v))$ time, because we have to rebuild one of the secondary structures.

   (a) **[1 pt]** Overall, how much space does this data structure use *in the worst case*?

   (b) **[1 pt]** How much space does this structure use if the primary search tree T is perfectly balanced?

   (c) **[2 pts]** Suppose T is a splay tree. Prove that the *amortized* cost of a splay (and therefore of a search, insertion, or deletion) is $\Omega(n)$. *[Hint: This is easy!]*

(d) **[3 pts]** Now suppose T is a scapegoat tree, and that rebuilding the subtree rooted at v requires $\Theta(\mathit{size}(v) \log \mathit{size}(v))$ time (because we also have to rebuild the secondary structures at every descendant of v). What is the *amortized* cost of inserting a new element into T?

(e) **[3 pts]** Finally, suppose T is a treap. What's the worst-case *expected* time for inserting a new element into T?

CS 473: Undergraduate Algorithms, Spring 2009
Homework 6

Written solutions due Tuesday, March 17, 2009 at 11:59:59pm.

1. Let G be an undirected graph with n nodes. Suppose that G contains two nodes s and t, such that every path from s to t contains more than $n/2$ edges.

   (a) Prove that G must contain a vertex v that lies on *every* path from s to t.

   (b) Describe an algorithm that finds such a vertex v in $O(V + E)$ time.

2. Suppose you are given a graph G with weighted edges and a minimum spanning tree T of G.

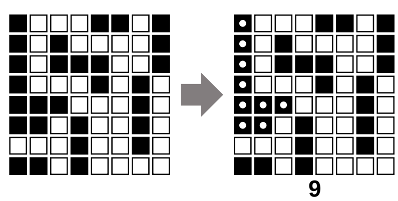
   (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.

   (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

   In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. *[Hint: Consider the cases $e \in T$ and $e \notin T$ separately.]*

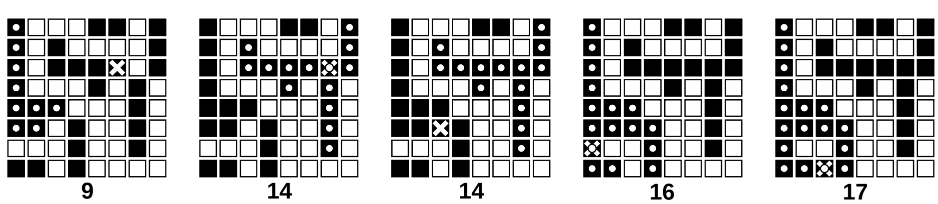
3. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1\,..\,n, 1\,..\,n]$.

   For example, given the bitmap below as input, your algorithm should return the number 9, because the largest conected black component (marked with white dots on the right) contains nine pixels.



   (b) Design and analyze an algorithm BLACKEN(i, j) that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

   For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the BLACKEN algorithm.



   (c) What is the *worst-case* running time of your BLACKEN algorithm?

CS 473: Undergraduate Algorithms, Spring 2009
Homework 6½

Practice only—do not submit solutions

---

1. In class last Tuesday, we discussed Ford's generic shortest-path algorithm—relax arbitrary tense edges until no edge is tense. This problem asks you to fill in part of the proof that this algorithm is correct.

   (a) Prove that after *every* call to RELAX, for every vertex v, either $dist(v) = \infty$ or $dist(v)$ is the total weight of some path from s to v.

   (b) Prove that for every vertex v, when the generic algorithm halts, either $pred(v) = \textsc{Null}$ and $dist(v) = \infty$, or $dist(v)$ is the total weight of the predecessor chain ending at v:

   $$s \to \cdots \to pred(pred(v)) \to pred(v) \to v.$$

2. Describe a modification of Shimbel's shortest-path algorithm that actually computes a negative-weight cycle if any such cycle is reachable from s, or a shortest-path tree rooted at s if there is no such cycle. Your modified algorithm should still run in $O(VE)$ time.

3. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

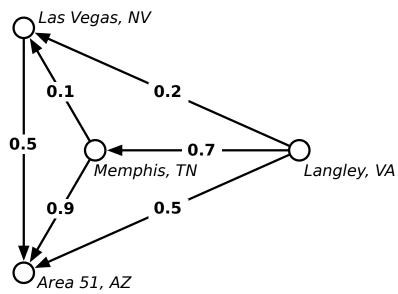
   Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights. *[Hint: Modify the input data and apply Dijkstra's algorithm.]*

CS 473: Undergraduate Algorithms, Spring 2009
Homework 6¾

Practice only—do not submit solutions

1. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

   More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.

   For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted! (That's how they got Elvis, you know.)

2. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero. Suppose the vertices of G are partitioned into k disjoint subsets $V_1, V_2, \ldots, V_k$; that is, every vertex of G belongs to exactly one subset V_i. For each i and j, let $\delta(i, j)$ denote the minimum shortest-path distance between any vertex in V_i and any vertex in V_j:

   $$\delta(i, j) = \min\{dist(u, v) \mid u \in V_i \text{ and } v \in V_j\}.$$

   Describe an algorithm to compute $\delta(i, j)$ for *all* i and j in time $O(VE + kE \log E)$. The output from your algorithm is a $k \times k$ array.

3. Recall[1] that a deterministic finite automaton (DFA) is formally defined as a tuple $M = (\Sigma, Q, q_0, F, \delta)$, where the finite set Σ is the input alphabet, the finite set Q is the set of states, $q_0 \in Q$ is the start state, $F \subseteq Q$ is the set of final (accepting) states, and $\delta : Q \times \Sigma \to Q$ is the transition function. Equivalently, a DFA is a directed (multi-)graph with labeled edges, such that each symbol in Σ is the label of exactly one edge leaving any vertex. There is a special 'start' vertex q_0, and a subset of the vertices are marked as 'accepting states'. Any string in Σ^* describes a unique walk starting at q_0.

   Stephen Kleene[2] proved that the language accepted by any DFA is identical to the language described by some regular expression. This problem asks you to develop a variant of the Floyd-Warshall all-pairs shortest path algorithm that computes a regular expression that is equivalent to the language accepted by a given DFA.

   Suppose the input DFA M has n states, numbered from 1 to n, where (without loss of generality) the start state is state 1. Let $L(i, j, r)$ denote the set of all words that describe walks in M from state i to state j, where every intermediate state lies in the subset $\{1, 2, \ldots, r\}$; thus, the language accepted by the DFA is exactly

$$\bigcup_{q \in F} L(1, q, n).$$

   Let $R(i, j, r)$ be a regular expression that describes the language $L(i, j, r)$.

   (a) What is the regular expression $R(i, j, 0)$?

   (b) Write a recurrence for the regular expression $R(i, j, r)$ in terms of regular expressions of the form $R(i', j', r - 1)$.

   (c) Describe a polynomial-time algorithm to compute $R(i, j, n)$ for all states i and j. (Assume that you can concatenate two regular expressions in $O(1)$ time.)

[1] No, really, you saw this in CS 273/373.

[2] Pronounced 'clay knee', not 'clean' or 'clean-ee' or 'clay-nuh' or 'dimaggio'.

CS 473: Undergraduate Algorithms, Spring 2009
Homework 7

Due Tuesday, April 14, 2009 at 11:59:59pm.

- Groups of up to three students may submit a single, common solution for this and all future homeworks. Please clearly write every group member's name and NetID on every page of your submission.

---

1. A graph is *bipartite* if its vertices can be colored black or white such that every edge joins vertices of two different colors. A graph is d-*regular* if every vertex has degree d. A *matching* in a graph is a subset of the edges with no common endpoints; a matching is *perfect* if it touches every vertex.

   (a) Prove that every regular bipartite graph contains a perfect matching.

   (b) Prove that every d-regular bipartite graph is the union of d perfect matchings.

2. Let $G = (V, E)$ be a directed graph where for each vertex v, the in-degree of v and out-degree of v are equal. Let u and v be two vertices G, and suppose G contains k edge-disjoint paths from u to v. Under these conditions, must G also contain k edge-disjoint paths from v to u? Give a proof or a counterexample with explanation.

3. A flow f is called ***acyclic*** if the subgraph of directed edges with positive flow contains no directed cycles. A flow is *positive* if its value is greater than 0.

   (a) A *path flow* assigns positive values only to the edges of one simple directed path from s to t. Prove that every positive acyclic flow can be written as the sum of a finite number of path flows.

   (b) Describe a flow in a directed graph that *cannot* be written as the sum of path flows.

   (c) A *cycle flow* assigns positive values only to the edges of one simple directed cycle. Prove that every flow can be written as the sum of a finite number of path flows and cycle flows.

   (d) Prove that for any flow f, there is an acyclic flow with the same value as f. (In particular, this implies that some maximum flow is acyclic.)

CS 473: Undergraduate Algorithms, Spring 2009
Homework 8

Due Tuesday, April 21, 2009 at 11:59:59pm.

- Groups of up to three students may submit a single, common solution for this and all future homeworks. Please clearly write every group member's name and NetID on every page of your submission.

1. A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover all the vertices. Describe and analyze an efficient algorithm to find a cycle cover for a given graph, or correctly report that non exists. *[Hint: Use bipartite matching!]*

2. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

   Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

3. *Ad-hoc networks* are made up of cheap, low-powered wireless devices. In principle[1], these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that several simple devices could be distributed randomly in the area of interest (for example, dropped from an airplane), and then they would somehow automatically configure themselves into an efficiently functioning wireless network.

   The devices can communicate only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D.

   We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit all its information to some other *backup* device within its communication range. To improve reliability, we require each device x to have k potential backup devices, all within distance D of x; we call these k devices the ***backup set*** of x. Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

   Suppose we are given the communication distance D, parameters b and k, and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j. Describe and analyze an algorithm that either computes a backup set of size k for each of the n devices, such that that no device appears in more than b backup sets, or correctly reports that no good collection of backup sets exists.

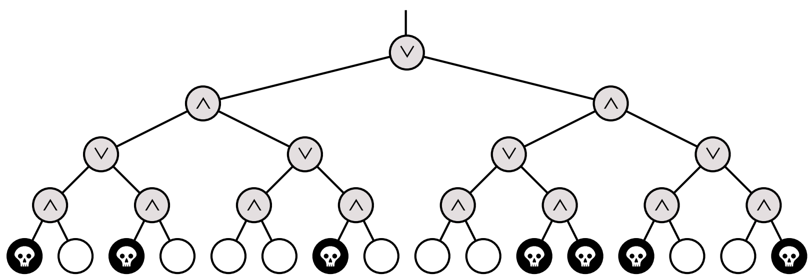
[1] but not so much in practice

CS 473: Undergraduate Algorithms, Spring 2009
Homework 9

Due Tuesday, April 28, 2009 at 11:59:59pm.

- Groups of up to three students may submit a single, common solution for this and all future homeworks. Please clearly write every group member's name and NetID on every page of your submission.

1. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it is white, you will live forever. You move first, so Death gets the last turn.



You can decide whether it is worth playing or not as follows. Imagine that the nodes at even levels (where it is your turn) are Or gates, the nodes at odd levels (where it is Death's turn) are And gates. Each gate gets its input from its children and passes its output to its parent. White and black leaves stand represent True and False inputs, respectively. If the output at the top of the tree is True, then you can win and live forever! If the output at the top of the tree is False, you should challenge Death to a game of Twister instead.

(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy.]*

(b) Prove that *every* deterministic algorithm must examine *every* leaf of the tree in the worst case. Since there are 4^n leaves, this implies that any deterministic algorithm must take $\Omega(4^n)$ time in the worst case. Use an adversary argument; in other words, assume that Death cheats.

(c) ***[Extra credit]*** Describe a *randomized* algorithm that runs in $O(3^n)$ expected time.

2. We say that an array $A[1..n]$ is *k-sorted* if it can be divided into k blocks, each of size n/k, such that the elements in each block are larger than the elements in earlier blocks, and smaller than elements in later blocks. The elements within each block need not be sorted.

For example, the following array is 4-sorted:

1	2	4	3	7	6	8	5	10	11	9	12	15	13	16	14

(a) Describe an algorithm that k-sorts an arbitrary array in time $O(n \log k)$.

(b) Prove that any comparison-based k-sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst case.

(c) Describe an algorithm that completely sorts an already k-sorted array in time $O(n \log(n/k))$.

(d) Prove that any comparison-based algorithm to completely sort a k-sorted array requires $\Omega(n \log(n/k))$ comparisons in the worst case.

In all cases, you can assume that n/k is an integer.

3. UIUC has just finished constructing the new Reingold Building, the tallest dormitory on campus. In order to determine how much insurance to buy, the university administration needs to determine the highest safe floor in the building. A floor is consdered *safe* if ~~a drunk student~~ ***an egg*** can fall from a window on that floor and land without breaking; if the egg breaks, the floor is considered *unsafe*. Any floor that is higher than an unsafe floor is also considered unsafe. The only way to determine whether a floor is safe is to drop an egg from a window on that floor.

   You would like to find the lowest unsafe floor L by performing as few tests as possible; unfortunately, you have only a very limited supply of eggs.

   (a) Prove that if you have only one egg, you can find the lowest unsafe floor with L tests. *[Hint: Yes, this is trivial.]*

   (b) Prove that if you have only one egg, you must perform at least L tests in the worst case. In other words, prove that your algorithm from part (a) is optimal. *[Hint: Use an adversary argument.]*

   (c) Describe an algorithm to find the lowest unsafe floor using *two* eggs and only $O(\sqrt{L})$ tests. *[Hint: Ideally, each egg should be dropped the same number of times. How many floors can you test with n drops?]*

   (d) Prove that if you start with two eggs, you must perform at least $\Omega(\sqrt{L})$ tests in the worst case. In other words, prove that your algorithm from part (c) is optimal.

   *(e) **[Extra credit!]** Describe an algorithm to find the lowest unsafe floor using k eggs, using as few tests as possible, and prove your algorithm is optimal for all values of k.

CS 473: Undergraduate Algorithms, Spring 2009
Head Banging Session 0

January 20 and 21, 2009

---

1. Solve the following recurrences. If base cases are provided, find an *exact* closed-form solution. Otherwise, find a solution of the form $\Theta(f(n))$ for some function f.

   - **Warmup:** You should be able to solve these almost as fast as you can write down the answers.
     (a) $A(n) = A(n-1) + 1$, where $A(0) = 0$.
     (b) $B(n) = B(n-5) + 2$, where $B(0) = 17$.
     (c) $C(n) = C(n-1) + n^2$
     (d) $D(n) = 3\,D(n/2) + n^2$
     (e) $E(n) = 4\,E(n/2) + n^2$
     (f) $F(n) = 5\,F(n/2) + n^2$

   - **Real practice:**
     (a) $A(n) = A(n/3) + 3A(n/5) + A(n/15) + n$
     (b) $B(n) = \min_{0<k<n} (B(k) + B(n-k) + n)$
     (c) $C(n) = \max_{n/4<k<3n/4} (C(k) + C(n-k) + n)$
     (d) $D(n) = \max_{0<k<n} \big(D(k) + D(n-k) + k(n-k)\big)$, where $D(1) = 0$
     (e) $E(n) = 2E(n-1) + E(n-2)$, where $E(0) = 1$ and $E(1) = 2$
     (f) $F(n) = \dfrac{1}{F(n-1)\,F(n-2)}$, where $F(0) = 1$ and $F(2) = 2$
     *(g) $G(n) = n\,G(\sqrt{n}) + n^2$

2. The *Fibonacci numbers* F_n are defined recursively as follows: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for every integer $n \geq 2$. The first few Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$.

   Prove that any non-negative integer can be written as the sum of distinct *non-consecutive* Fibonacci numbers. That is, if any Fibonacci number F_n appears in the sum, then its neighbors F_{n-1} and F_{n+1} do not. For example:

$$\begin{aligned} 88 &= 55 + 21 + 8 + 3 + 1 &&= F_{10} + F_8 + F_6 + F_4 + F_2 \\ 42 &= 34 + 8 &&= F_9 + F_6 \\ 17 &= 13 + 3 + 1 &&= F_7 + F_4 + F_2 \end{aligned}$$

3. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

   Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left. Pretty please.

1. An *inversion* in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward).

   Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time.

2. (a) Prove that the following algorithm actually sorts its input.

```
StoogeSort(A[0..n−1]):
  if n = 2 and A[0] > A[1]
      swap A[0] ⟷ A[1]
  else if n > 2
      m = ⌈2n/3⌉
      StoogeSort(A[0..m−1])
      StoogeSort(A[n−m..n−1])
      StoogeSort(A[0..m−1])
```

   (b) Would StoogeSort still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.

   (c) State a recurrence (including base case(s)) for the number of comparisons executed by StoogeSort.

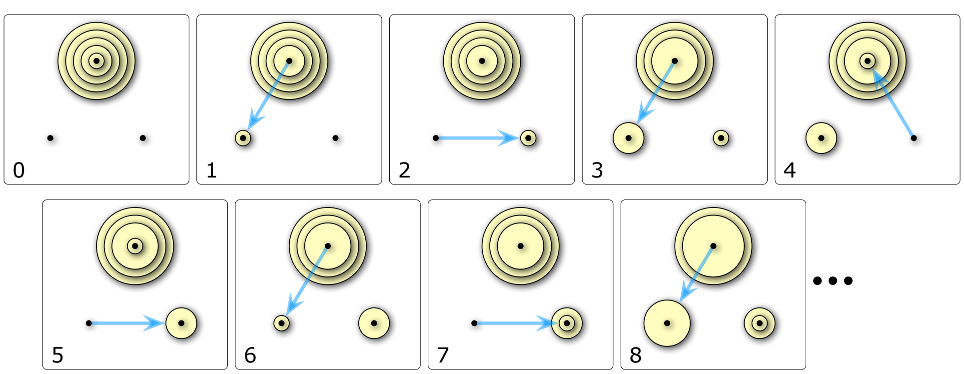
   (d) Solve this recurrence. *[Hint: Ignore the ceiling.]*

   (e) **To think about on your own:** Prove that the number of *swaps* executed by StoogeSort is at most $\binom{n}{2}$.

3. Consider the following restricted variants of the Tower of Hanoi puzzle. In each problem, the needles are numbered 0, 1, and 2, and your task is to move a stack of n disks from needle 1 to needle 2.

   (a) Suppose you are forbidden to move any disk directly between needle 1 and needle 2; *every* move must involve needle 0. Describe an algorithm to solve this version of the puzzle using as few moves as possible. *Exactly* how many moves does your algorithm make?

   (b) Suppose you are only allowed to move disks from needle 0 to needle 2, from needle 2 to needle 1, or from needle 1 to needle 0. Equivalently, Suppose the needles are arranged in a circle and numbered in clockwise order, and you are only allowed to move disks counterclockwise. Describe an algorithm to solve this version of the puzzle using as few moves as possible. *Exactly* how many moves does your algorithm make?



The first eight moves in a counterclockwise Towers of Hanoi solution

★(c) Finally, suppose you are forbidden to move any disk directly from needle 1 to 2, but any other move is allowed. Describe an algorithm to solve this version of the puzzle using as few moves as possible. *Exactly* how many moves does your algorithm make?

[Hint: This version is ***considerably*** *harder than the other two.]*

CS 473: Undergraduate Algorithms, Spring 2009
HBS 10

1. Consider the following problem, called *BOX-DEPTH*: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

   (a) Describe a polynomial-time reduction from *BOX-DEPTH* to *MAX-CLIQUE*.

   (b) Describe and analyze a polynomial-time algorithm for *BOX-DEPTH*. [Hint: $O(n^3)$ time should be easy, but $O(n \log n)$ time is possible.]

   (c) Why don't these two results imply that $P = NP$?

2. Suppose you are given a magic black box that can determine in polynomial time, given an arbitrary weighted graph G, the length of the shortest Hamiltonian cycle in G. Describe and analyze a polynomial-time algorithm that computes, given an arbitrary weighted graph G, the shortest Hamiltonian cycle in G, using this magic black box as a subroutine.

3. Prove that the following problems are NP-complete.

   (a) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 17?

   (b) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

CS 473: Undergraduate Algorithms, Spring 2009
HBS 11

1. You step in a party with a camera in your hand. Each person attending the party has some friends there. You want to have exactly one picture of each person in your camera. You want to use the following protocol to collect photos. At each step, the person that has the camera in his hand takes a picture of one of his/her friends and pass the camera to him/her. Of course, you only like the solution if it finishes when the camera is in your hand. Given the friendship matrix of the people in the party, design a polynomial algorithm that decides whether this is possible, or prove that this decision problem is NP-hard.

2. A boolean formula is in disjunctive normal form (DNF) if it is a disjunctions (OR) of several clauses, each of which is the conjunction (AND) of several literals, each of which is either a variable or its negation. For example:
$$(a \wedge b \wedge c) \vee (\bar{a} \wedge b) \vee (\bar{c} \wedge x)$$
Given a DNF formula give a polynomial algorithm to check whether it is satisfiable or not. Why this does not imply $P = NP$.

3. Prove that the following problems are NP-complete.
   (a) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 17?
   (b) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

CS 473: Undergraduate Algorithms, Spring 2009
HBS 2

1. Consider two horizontal lines l_1 and l_2 in the plane. There are n points on l_1 with x-coordinates $A = a_1, a_2, \ldots, a_n$ and there are n points on l_2 with x-coordinates $B = b_1, b_2, \ldots, b_n$. Design an algorithm to compute, given A and B, a largest set S of non-intersecting line segments subject to the following restrictions:

   (a) Any segment in S connects a_i to b_i for some $i(1 \le i \le n)$.

   (b) Any two segments in S do not intersect.

2. Consider a $2^n x 2^n$ chess board with one (arbitrarily chosen) square removed. Prove that any such chessboard can be tiled without gaps or overlaps by L-shaped pieces of 3 squares each. Can you give an algorithm to do the tiling?

3. Given a string of letters $Y = y_1 y_2 \ldots y_n$, a segmentation of Y is a partition of its letters into contiguous blocks of letters (also called words). Each word has a quality that can be computed by a given oracle (e.g. you can call *quality("meet")* to get the quality of the word "meet"). The quality of a segmentation is equal to the sum over the qualities of its words. Each call to the oracle takes linear time in terms of the argument; that is quality(S) takes $O(|S|)$.

   Using the given oracle, give an algorithm that takes a string Y and computes a segmentation of maximum total quality.

CS 473: Undergraduate Algorithms, Spring 2009
HBS 3

1. Change your recursive solutions for the following problems to efficient algorithms (Hint: use dynamic programming!).

   (a) Consider two horizontal lines l_1 and l_2 in the plane. There are n points on l_1 with x-coordinates $A = a_1, a_2, \ldots, a_n$ and there are n points on l_2 with x-coordinates $B = b_1, b_2, \ldots, b_n$. Design an algorithm to compute, given A and B, a largest set S of non-intersecting line segments subject to the following restrictions:

   i. Any segment in S connects a_i to b_i for some $i (1 \leq i \leq n)$.

   ii. Any two segments in S do not intersect.

   (b) Given a string of letters $Y = y_1 y_2 \ldots y_n$, a segmentation of Y is a partition of its letters into contiguous blocks of letters (also called words). Each word has a quality that can be computed by a given oracle (e.g. you can call *quality("meet")* to get the quality of the word "meet"). The quality of a segmentation is equal to the sum over the qualities of its words. Each call to the oracle takes linear time in terms of the argument; that is quality(S) takes $O(|S|)$.

   Using the given oracle, give an algorithm that takes a string Y and computes a segmentation of maximum total quality.

2. Give a polynomial time algorithm which given two strings A and B returns the longest sequence S that is a subsequence of A and B.

3. Consider a rooted tree T. Assume the root has a message to send to all nodes. At the beginning only the root has the message. If a node has the message, it can forward it to one of its children at each time step. Design an algorithm to find the minimum number of time steps required for the message to be delivered to all nodes.

CS 473: Undergraduate Algorithms, Spring 2009
HBS 3.5

1. Say you are given n jobs to run on a machine. Each job has a start time and an end time. If a job is chosen to be run, then it must start at its start time and end at its end time. Your goal is to accept as many jobs as possible, regardless of the job lengths, subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n. You may assume for simplicity that no two jobs have the same start or end times, but the start time and end time of two jobs can overlap.

2. Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, without knowing the length of the stream in advance. Your algorithm should spend $O(1)$ time per stream element and use $O(1)$ space (not counting the stream itself).

3. Design and analyze an algorithm that return a permutation of the integers $\{1, 2, \ldots, n\}$ chosen uniformly at random.

CS 473: Undergraduate Algorithms, Spring 2009
HBS 4

1. Let x and y be two elements of a set S whose ranks differ by exactly r. Prove that in a treap for S, the expected length of the unique path from x to y is $O(\log r)$

2. Consider the problem of making change for n cents using the least number of coins.

   (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

   (b) Suppose that the available coins have the values $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

   (c) Give a set of 4 coin values for which the greedy algorithm does not yield an optimal solution, show why.

   (d) Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of coin values.

3. A heater is a sort of dual treap, in which the priorities of the nodes are given, but their search keys are generate independently and uniformly from the unit interval $[0,1]$. You can assume all priorities and keys are distinct. Describe algorithms to perform the operations INSERT and DELETEMIN in a heater. What are the expected worst-case running times of your algorithms? In particular, can you express the expected running time of INSERT in terms of the priority rank of the newly inserted item?

CS 473: Undergraduate Algorithms, Spring 2009
HBS 5

1. Recall that the staircase of a set of points consists of the points with no other point both above and to the right. Describe a method to maintain the staircase as new points are added to the set. Specifically, describe and analyze a data structure that stores the staircase of a set of points, and an algorithm $INSERT(x, y)$ that adds the point (x, y) to the set and returns $TRUE$ or $FALSE$ to indicate whether the staircase has changed. Your data structure should use $O(n)$ space, and your INSERT algorithm should run in $O(\log n)$ amortized time.

2. In some applications, we do not know in advance how much space we will require. So, we start the program by allocating a (dynamic) table of some fixed size. Later, as new objects are inserted, we may have to allocate a larger table and copy the previous table to it. So, we may need more than $O(1)$ time for copying. In addition, we want to keep the table size small enough, avoiding a very large table to keep only few items. One way to manage a dynamic table is by the following rules:

   (a) Double the size of the table if an item is inserted into a full table

   (b) Halve the table size if a deletion causes the table to become less than 1/4 full

   Show that, in such a dynamic table we only need $O(1)$ amortized time, per operation.

3. Consider a stack data structure with the following operations:

   - PUSH(x): adds the element x to the top of the stack
   - POP: removes and returns the element that is currently on top of the stack (if the stack is non-empty)
   - SEARCH(x): repeatedly removes the element on top of the stack until x is found or the stack becomes empty

   What is the amortized cost of an operation?

CS 473: Undergraduate Algorithms, Spring 2009
HBS 6

1. Let G be a connected graph and let v be a vertex in G. Show that T is both a DFS tree and a BFS tree rooted at v, then $G = T$.

2. An Euler tour of a graph G is a walk that starts from a vertex v, visits every edge of G exactly once and gets back to v. Prove that G has an Euler tour if and only if all the vertices of G has even degrees. Can you give an efficient algorithm to find an Euler tour of such a graph.

3. You are helping a group of ethnographers analyze some oral history data they have collected by interviewing members of a village to learn about the lives of people lived there over the last two hundred years. From the interviews, you have learned about a set of people, all now deceased, whom we will denote $P_1, P_2, \dots, P_n$. The ethnographers have collected several facts about the lifespans of these people, of one of the following forms:

   (a) P_i died before P_j was born.

   (b) P_i and P_j were both alive at some moment.

   Naturally, the ethnographers are not sure that their facts are correct; memories are not so good, and all this information was passed down by word of mouth. So they'd like you to determine whether the data they have collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they have learned simultaneously hold.

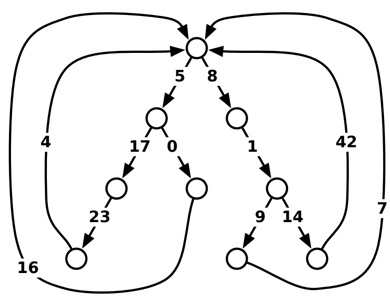
   Describe and analyze and algorithm to answer the ethnographers' problem. Your algorithm should either output possible dates of birth and death that are consistent with all the stated facts, or it should report correctly that no such dates exist.

CS 473: Undergraduate Algorithms, Spring 2009
HBS 6.5

1. (a) Describe and analyze and algorithm to find the *second smallest spanning tree* of a given graph G, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree.

   *(b) Describe and analyze an efficient algorithm to compute, given a weighted undirected graph G and an integer k, the k smallest spanning trees of G.

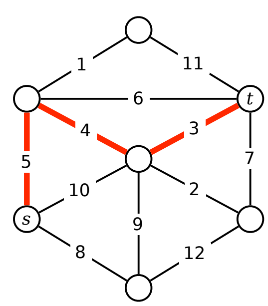
2. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight.



   (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?

   (b) Describe and analyze a faster algorithm.

3. Consider a path between two vertices s and t in an undirected weighted graph G. The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t. (If there are no paths from s to t, the bottleneck distance between s and t is ∞.)



The bottleneck distance between s and t is 5.

Describe and analyze an algorithm to compute the bottleneck distance between *every* pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.

CS 473: Undergraduate Algorithms, Spring 2009
HBS 6.55

1. Suppose you are given a directed graph $G = (V, E)$ with non-negative edge lengths; $\ell(e)$ is the length of $e \in E$. You are interested in the shortest path distance between two given locations/nodes s and t. It has been noticed that the existing shortest path distance between s and t in G is not satisfactory and there is a proposal to add exactly one edge to the graph to improve the situation. The candidate edges from which one has to be chosen is given by $E' = \{e_1, e2, \ldots, e_k\}$ and you can assume that $E \cup E' = \emptyset$. The length of the e_i is $\alpha_i \geq 0$. Your goal is figure out which of these k edges will result in the most reduction in the shortest path distance from s to t. Describe an algorithm for this problem that runs in time $O((m+n)\log n + k)$ where $m = |E|$ and $n = |V|$. Note that one can easily solve this problem in $O(k(m+n)\log n)$ by running Dijkstra's algorithm k times, one for each G_i where G_i is the graph obtained by adding e_i to G.

2. Let G be an undirected graph with non-negative edge weights. Let s and t be two vertices such that the shortest path between s and t in G contains all the vertices in the graph. For each edge e, let $G \backslash e$ be the graph obtained from G by deleting the edge e. Design an $O(E \log V)$ algorithm that finds the shortest path distance between s and t in $G \backslash e$ for all e. [*Note that you need to output E distances, one for each graph $G \backslash e$*]

3. Given a Directed Acyclic Graph (DAG) and two vertices s and t you want to determine if there is an s to t path that includes at least k vertices.

CS 473: Undergraduate Algorithms, Spring 2009
HBS 7

1. Let $G = (V, E)$ be a directed graph with non-negative capacities. Give an efficient algorithm to check whether there is a unique max-flow on G?

2. Let $G = (V, E)$ be a graph and $s, t \in V$ be two specific vertices of G. We call $(S, T = V \setminus S)$ an (s, t)-cut if $s \in S$ and $t \in T$. Moreover, it is a minimum cut if the sum of the capacities of the edges that have one endpoint in S and one endpoint in T equals the maximum (s, t)-flow. Show that, both intersection and union of two min-cuts is a min-cut itself.

3. Let $G = (V, E)$ be a graph. For each edge e let $d(e)$ be a demand value attached to it. A flow is feasible if it sends more than $d(e)$ through e. Assume you have an oracle that is capable of solving the maximum flow problem. Give efficient algorithms for the following problems that call the oracle at most once.

   (a) Find a feasible flow.

   (b) Find a feasible flow of minimum possible value.

CS 473: Undergraduate Algorithms, Spring 2009
HBS 8

1. A box i can be specified by the values of its sides, say (i_1, i_2, i_3). We know all the side lengths are larger than 10 and smaller than 20 (i.e. $10 < i_1, i_2, i_3 < 20$). Geometrically, you know what it means for one box to nest in another: It is possible if you can rotate the smaller so that it fits inside the larger in each dimension. Of course, nesting is recursive, that is if i nests in j and j nests in k then i nests in k. After doing some nesting operations, we say a box is visible if it is not nested in any other one. Given a set of boxes (each specified by the lengthes of their sides) the goal is to find a set of nesting operations to minimize the number of visible boxes. Design and analyze an efficient algorithm to do this.

2. Let the number of papers submitted to a conference be n and the number of available reviewers be m. Each reviewer has a list of papers that he/she can review and each paper should be reviewed by three different persons. Also, each reviewer can review at most 5 papers. Design and analyze an algorithm to make the assignment or decide no feasible assignment exists.

3. Back in the euphoric early days of the Web, people liked to claim that much of the enormous potential in a company like Yahoo! was in the "eyeballs" - the simple fact that it gets millions of people looking at its pages every day. And further, by convincing people to register personal data with the site, it can show each user an extremely targeted advertisement whenever he or she visits the site, in away that TV networks or magazines could not hope to match. So if the user has told Yahoo! that he is a 20-year old computer science major from Cornell University, the site can throw up a banner ad for apartments in Ithaca, NY; on the other hand, if he is a 50-year-old investment banker from Greenwich, Connecticut, the site can display a banner ad pitching Lincoln Town Cars instead.

   But deciding on which ads to show to which people involves some serious computation behind the scenes. Suppose that the managers of a popular Web site have identified k distinct demographic groups $G_1, G_2, \ldots, G_k$. (These groups can overlap; for example G_1 can be equal to all residents of New York State, and G_2 can be equal to all people with a degree in computer science.) The site has contracts with m different advertisers, to show a certain number of copies of their ads to users of the site. Here is what the contract with the i^{th} advertiser looks like:

   (a) For a subset $X_i \subset \{G_1, \ldots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i

   (b) For a number r_i , advertiser i wants its ads shown to at least r_i users each minute.

   Now, consider the problem of designing a good advertising policy - a way to show a single ad to each user of the site. Suppose at a given minute, there are n users visiting the site. Because we have registration information on each of these users, we know that user j(for $j = 1, 2, \ldots, n$) belongs to a subset $U_j \subset \{G_1, \ldots, G_k\}$ of the demographic groups. The problem is: is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute? (That is, for each $i = 1, 2, \ldots, m$, at least r_i of the n users, each belonging to at least one demographic group in X_i , are shown an ad provided by advertiser i.)

   Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

CS 473: Undergraduate Algorithms, Spring 2009
HBS 9

1. Prove that any algorithm to merge two sorted arrays, each of size n, requires at least $2n-1$ comparisons.

2. Suppose you want to determine the largest number in an n-element set $X = \{x_1, x_2, \ldots, x_n\}$, where each element x_i is an integer between 1 and $2^m - 1$. Describe an algorithm that solves this problem in $O(n+m)$ steps, where at each step, your algorithm compares one of the elements x_i with a *constant*. In particular, your algorithm must never actually compare two elements of X! *[Hint: Construct and maintain a nested set of 'pinning intervals' for the numbers that you have not yet removed from consideration, where each interval but the largest is either the upper half or lower half of the next larger block.]*

3. Let P be a set of n points in the plane. The staircase of P is the set of all points in the plane that have at least one point in P both above and to the right. Prove that computing the staircase requires at least $\Omega(n \log n)$ comparisons in two ways,

   (a) Reduction from sorting.

   (b) Directly.

You have 90 minutes to answer four of the five questions.
Write your answers in the separate answer booklet.
You may take the question sheet with you when you leave.

1. Each of these ten questions has one of the following five answers:

   A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$

   Choose the correct answer for each question. Each correct answer is worth $+1$ point; each incorrect answer is worth $-1/2$ point; each "I don't know" is worth $+1/4$ point. Your score will be rounded to the nearest *non-negative* integer.

   (a) What is $\displaystyle\sum_{i=1}^{n} \frac{n}{i}$?

   (b) What is $\displaystyle\sqrt{\sum_{i=1}^{n} i}$?

   (c) How many digits are required to write 3^n in decimal?

   (d) What is the solution to the recurrence $D(n) = D(n/\pi) + \sqrt{2}$?

   (e) What is the solution to the recurrence $E(n) = E(n - \sqrt{2}) + \pi$?

   (f) What is the solution to the recurrence $F(n) = 4F(n/2) + 3n$?

   (g) What is the worst-case time to search for an item in a binary search tree?

   (h) What is the worst-case running time of quicksort?

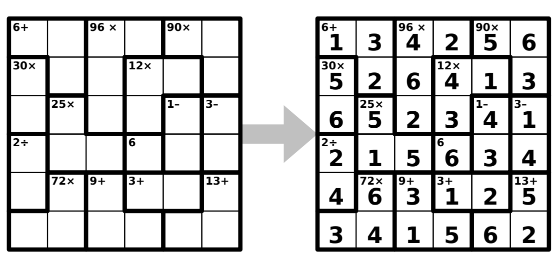
   (i) Let $H[1 .. n, 1 .. n]$ be a fixed array of numbers. Consider the following recursive function:

   $$Glub(i, j) = \begin{cases} 0 & \text{if } i = 0 \\ \infty & \text{if } i > n \text{ or } j = 0 \\ \max\left\{Glub(i-1, j),\ H[i, j] + Glub(i+1, j-1)\right\} & \text{otherwise} \end{cases}$$

   How long does it take to compute $Glub(n, n)$ using dynamic programming?

   (j) What is the running time of the fastest possible algorithm to solve KenKen puzzles?

   A KenKen puzzle is a 6×6 grid, divided into regions called *cages*. Each cage is labeled with a numerical *value* and an arithmetic *operation*: $+$, $-$, $\times$, or $\div$. (The operation can be omitted if the cage consists of a single cell.) The goal is to place an integer between 1 and 6 in each grid cell, so that no number appears twice in any row or column, and the numbers inside each cage can be combined using *only* that cage's operation to obtain that cage's value. The solution is guaranteed to be unique.

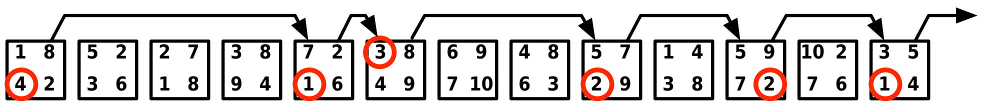


A Kenken puzzle and its solution

2. (a) Suppose $A[1..n]$ is an array of n distinct integers, sorted so that $A[1] < A[2] < \cdots < A[n]$. Each integer $A[i]$ could be positive, negative, or zero. Describe an efficient algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. An algorithm that runs in $\Theta(n)$ time is worth at most 3 points.

   (b) Now suppose $A[1..n]$ is a sorted array of n distinct ***positive*** integers. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy!]*

3. *Moby Selene* is a solitaire game played on a row of n squares. Each square contains four positive integers. The player begins by placing a token on the leftmost square. On each move, the player chooses one of the numbers on the token's current square, and then moves the token that number of squares to the right. The game ends when the token moves past the rightmost square. The object of the game is to make as many moves as possible before the game ends.



A Moby Selene puzzle that allows six moves. (This is ***not*** the longest legal sequence of moves.)

   (a) ***Prove*** that the obvious greedy strategy (always choose the smallest number) does not give the largest possible number of moves for every Moby Selene puzzle.

   (b) Describe and analyze an efficient algorithm to find the largest possible number of legal moves for a given Moby Selene puzzle.

4. Consider the following algorithm for finding the largest element in an unsorted array:

```
RandomMax(A[1..n]):
  max ← ∞
  for i ← 1 to n in random order
      if A[i] > max
          max ← A[i]   (⋆)
  return max
```

   (a) In the worst case, how many times does RandomMax execute line $(\star)$?

   (b) What is the ***exact*** probability that line $(\star)$ is executed during the last iteration of the for loop?

   (c) What is the ***exact*** expected number of executions of line $(\star)$? (A correct $\Theta(\,)$ bound is worth half credit.)

5. *This question is taken directly from HBS 0.* Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

   Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left. Pretty please.

You have 90 minutes to answer four of the five questions.
Write your answers in the separate answer booklet.
You may take the question sheet with you when you leave.

1. Recall that a ***tree*** is a connected graph with no cycles. A graph is ***bipartite*** if we can color its vertices black and white, so that every edge connects a white vertex to a black vertex.

   (a) ***Prove*** that every tree is bipartite.

   (b) Describe and analyze a fast algorithm to determine whether a given graph is bipartite.

2. Describe and analyze an algorithm SHUFFLE($A[1..n]$) that randomly permutes the input array A, so that each of the $n!$ possible permutations is equally likely. You can assume the existence of a subroutine RANDOM(k) that returns a random integer chosen uniformly between 1 and k in $O(1)$ time. For full credit, your SHUFFLE algorithm should run in $O(n)$ time. *[Hint: This problem appeared in HBS 3½.]*

3. Let G be an undirected graph with weighted edges.

   (a) Describe and analyze an algorithm to compute the *maximum* weight spanning tree of G.

   (b) A ***feedback edge set*** of G is a subset F of the edges such that every cycle in G contains at least one edge in F. In other words, removing every edge in F makes G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of G.

   [Hint: Don't reinvent the wheel!]

4. Let $G = (V, E)$ be a connected directed graph with non-negative edge weights, let s and t be vertices of G, and let H be a subgraph of G obtained by deleting some edges. Suppose we want to reinsert exactly one edge from G back into H, so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm to choose the best edge to reinsert. For full credit, your algorithm should run in $O(E \log V)$ time. *[Hint: This problem appeared in HBS 6¾.]*

5. Describe and analyze an efficient data structure to support the following operations on an array $X[1..n]$ as quickly as possible. Initially, $X[i] = 0$ for all i.

   - Given an index i such that $X[i] = 0$, set $X[i]$ to 1.
   - Given an index i, return $X[i]$.
   - Given an index i, return the smallest index $j \geq i$ such that $X[j] = 0$, or report that no such index exists.

   For full credit, the first two operations should run in *worst-case constant* time, and the amortized cost of the third operation should be as small as possible.

You have 180 minutes to answer six of the seven questions.
Write your answers in the separate answer booklet.
You may take the question sheet with you when you leave.

1. SUBSETSUM and PARTITION are two closely related NP-hard problems, defined as follows.

   SUBSETSUM: Given a set X of positive integers and a positive integer k, does X have a subset whose elements sum up to k?

   PARTITION: Given a set Y of positive integers, can Y be partitioned into two subsets whose sums are equal?

   (a) **[2 pts]** ***Prove*** that PARTITION and SUBSETSUM are both in NP.

   (b) **[1 pt]** Suppose you already know that SUBSETSUM is NP-hard. Which of the following arguments could you use to prove that PARTITION is NP-hard? ***You do not need to justify your answer*** — just answer ① or ②.

   ① Given a set X and an integer k, construct a set Y in polynomial time, such that PARTITION(Y) is true if and only if SUBSETSUM(X, k) is true.

   ② Given a set Y, construct a set X and an integer k in polynomial time, such that PARTITION(Y) is true if and only if SUBSETSUM(X, k) is true.

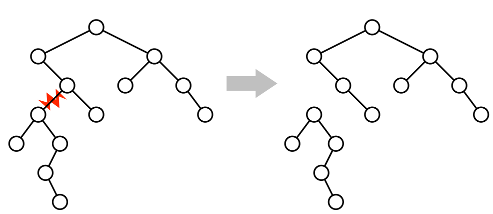
   (c) **[3 pts]** Describe and analyze a polynomial-time reduction from PARTITION to SUBSETSUM. ***You do not need to prove that your reduction is correct.***

   (d) **[4 pts]** Describe and analyze a polynomial-time reduction from SUBSETSUM to PARTITION. ***You do not need to prove that your reduction is correct.***

2. (a) **[4 pts]** For any node v in a binary tree, let $size(v)$ denote the number of nodes in the subtree rooted at v. Let k be an arbitrary positive number. ***Prove*** that every binary tree with at least k nodes contains a node v such that $k \le size(v) \le 2k$.

   (b) **[2 pts]** Removing any edge from an n-node binary tree T separates it into two smaller binary trees. An edge is called a ***balanced separator*** if both of these subtrees have at least $n/3$ nodes (and therefore at most $2n/3$ nodes). ***Prove*** that every binary tree with more than one node has a balanced separator. *[Hint: Use part (a).]*

   (c) **[4 pts]** Describe and analyze an algorithm to find a balanced separator in a given binary tree. *[Hint: Use part (a).]*



Removing a balanced separator from a binary tree.

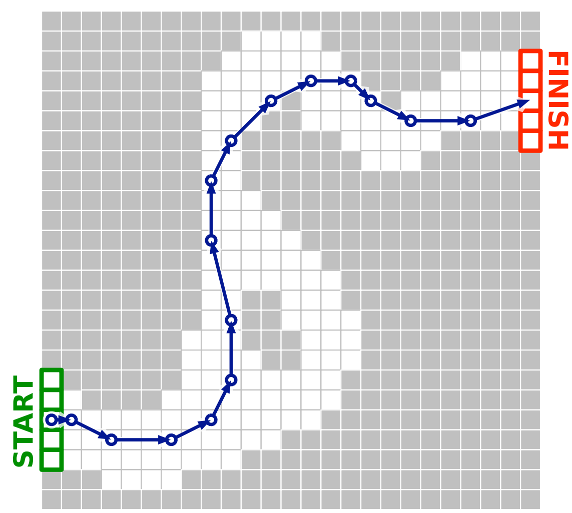
3. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[1] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

   Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. The initial position is a point on the starting line, chosen by the player; the initial velocity is always $(0,0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by ***at most* 1** in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position ***on*** the finish line.

   Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the 'starting line' is the first column, and the 'finish line' is the last column.

   Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack. *[Hint: Build a graph. What are the vertices? What are the edges? What problem is this?]*

velocity	position
$(0,0)$	$(1,5)$
$(1,0)$	$(2,5)$
$(2,-1)$	$(4,4)$
$(3,0)$	$(7,4)$
$(2,1)$	$(9,5)$
$(1,2)$	$(10,7)$
$(0,3)$	$(10,10)$
$(-1,4)$	$(9,14)$
$(0,3)$	$(9,17)$
$(1,2)$	$(10,19)$
$(2,2)$	$(12,21)$
$(2,1)$	$(14,22)$
$(2,0)$	$(16,22)$
$(1,-1)$	$(17,21)$
$(2,-1)$	$(19,20)$
$(3,0)$	$(22,20)$
$(3,1)$	$(25,21)$



A 16-step Racetrack run, on a 25×25 track.

4. A palindrome is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`. Describe and analyze an algorithm to find the length of the longest *subsequence* of a given string that is also a palindrome.

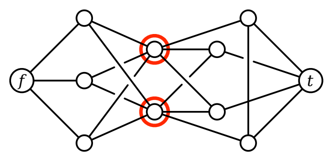
   For example, the longest palindrome subsequence of `MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM` is `MHYMRORMYHM`, so given that string as input, your algorithm should output the number 11.

[1] The actual game is a bit more complicated than the version described here.

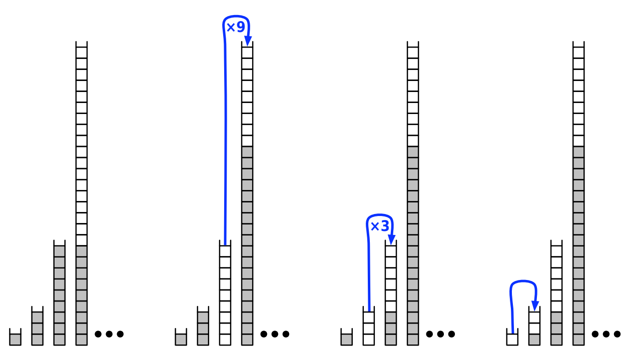
5. The Island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

   Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices f and t represent the stations in Ffarquhar and Tidmouth.

   For example, given the following input graph, your algorithm should return the number 2.



6. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i, we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move all its members to S_{i+2}.) Moving a single element from one stack to the next takes $O(1)$ time.



Making room for one new element in a multistack.

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?

   (b) ***Prove*** that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack.

7. Recall the problem 3Color: Given a graph, can we color each vertex with one of 3 colors, so that every edge touches two different colors? We proved in class that 3Color is NP-hard.

   Now consider the related problem 12Color: Given a graph, can we color each vertex with one of twelve colors, so that every edge touches two different colors? ***Prove*** that 12Color is NP-hard.

You may assume the following problems are NP-hard:

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

PlanarCircuitSat: Given a boolean circuit drawn in the plane so that no two wires cross, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

HamiltonianCycle: Given a graph G, can is there a cycle in G that visits every vertex once?

HamiltonianPath: Given a graph G, can is there a path in G that visits every vertex once?

DoubleHamiltonianCycle: Given a graph G, can is there a closed walk in G that visits every vertex twice?

DoubleHamiltonianPath: Given a graph G, can is there an open walk in G that visits every vertex twice?

MinDegreeSpanningTree: Given an undirected graph G, what is the minimum degree of any spanning tree of G?

MinLeavesSpanningTree: Given an undirected graph G, what is the minimum number of leaves in any spanning tree of G?

TravelingSalesman: Given a graph G with weighted edges, what is the minimum cost of any Hamiltonian path/cycle in G?

LongestPath: Given a graph G with weighted edges and two vertices s and t, what is the length of the longest *simple* path from s to t in G?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of n positive integers, can X be partitioned into $n/3$ three-element subsets, all with the same sum?

Minesweeper: Given a Minesweeper configuration and a particular square x, is it safe to click on x?

Tetris: Given a sequence of N Tetris pieces and a partially filled $n \times k$ board, is it possible to play every piece in the sequence without overflowing the board?

Sudoku: Given an $n \times n$ Sudoku puzzle, does it have a solution?

KenKen: Given an $n \times n$ Ken-Ken puzzle, does it have a solution?

CS 473: Undergraduate Algorithms, Spring 2010
Homework 0

Due Tuesday, January 26, 2009 **in class**

- This homework tests your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, graphs, and most importantly, induction—to help you identify gaps in your background knowledge. **You are responsible for filling those gaps.** The early chapters of any algorithms textbook should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks. If you need help, please ask in office hours and/or on the course newsgroup.

- Each student must submit individual solutions for these homework problems. For all future homeworks, groups of up to three students may submit (or present) a single group solution for each problem.

- Please carefully read the course policies linked from the course web site. If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup. In particular:

  – Submit five separately stapled solutions, one for each numbered problem, with your name and NetID clearly printed on each page. Please do *not* staple everything together.

  – You may use any source at your disposal—paper, electronic, or human—but you ***must*** write your solutions in your own words, and you ***must*** cite every source that you use.

  – Unless explicitly stated otherwise, ***every*** homework problem requires a proof.

  – Answering "I don't know" to any homework or exam problem (except for extra credit problems) is worth 25% partial credit.

  – Algorithms or proofs containing phrases like "and so on" or "repeat this process for all n", instead of an explicit loop, recursion, or induction, will receive 0 points.

---

1. (a) ***Write the sentence "I understand the course policies."***

   (b) **[5 pts]** Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Assume reasonable but nontrivial base cases if none are given. ***Do not submit proofs***—just a list of five functions—but you should do them anyway, just for practice.

   - $A(n) = 3A(n-1) + 1$
   - $B(n) = B(n-5) + 2n - 3$
   - $C(n) = 4C(n/2) + \sqrt{n}$
   - $D(n) = 3D(n/3) + n^2$
   - $E(n) = E(n-1)^2 - E(n-2)^2$, where $E(0) = 0$ and $E(1) = 1$ *[Hint: This is easy!]*

   (c) **[5 pts]** Sort the following functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. ***Do not submit proofs***—just a sorted list of 16 functions—but you should do them anyway, just for practice.

   Write $f(n) \ll g(n)$ to indicate that $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. We use the notation $\lg n = \log_2 n$.

n	$\lg n$	$\sqrt{n}$	5^n
$\sqrt{\lg n}$	$\lg \sqrt{n}$	$5^{\sqrt{n}}$	$\sqrt{5^n}$
$5^{\lg n}$	$\lg(5^n)$	$5^{\lg \sqrt{n}}$	$5^{\sqrt{\lg n}}$
$\sqrt{5^{\lg n}}$	$\lg(5^{\sqrt{n}})$	$\lg \sqrt{5^n}$	$\sqrt{\lg(5^n)}$

2. **[CS 225 Spring 2009]** Suppose we build up a binary search tree by inserting elements one at a time from the set $\{1, 2, 3, \ldots, n\}$, starting with the empty tree. The structure of the resulting binary search tree depends on the order that these elements are inserted; every insertion order leads to a different n-node binary search tree.

   Recall that the *depth* of a leaf ℓ in a binary search tree is the number of *edges* between ℓ and the root, and the depth of a binary tree is the maximum depth of its leaves.

   (a) What is the maximum possible depth of an n-node binary search tree? Give an *exact* answer, and prove that it is correct.

   (b) *Exactly* how many different insertion orders result in an n-node binary search tree with maximum possible depth? Prove your answer is correct. *[Hint: Set up and solve a recurrence. Don't forget to prove that recurrence counts what you want it to count.]*

3. **[CS 173 Spring 2009]** A *binomial tree of order* k is defined recursively as follows:

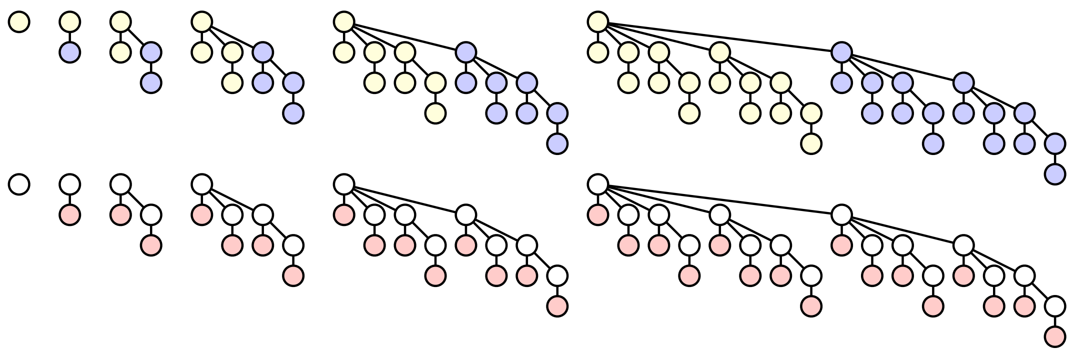
   - A binomial tree of order 0 is a single node.
   - For all $k > 0$, a binomial tree of order k consists of two binomial trees of order $k-1$, with the root of one tree connected as a new child of the root of the other. (See the figure below.)

   Prove the following claims:

   (a) For all non-negative integers k, a binomial tree of order k has exactly 2^k nodes.

   (b) For all positive integers k, attaching a leaf to every node in a binomial tree of order $k-1$ results in a binomial tree of order k.

   (c) For all non-negative integers k and d, a binomial tree of order k has exactly $\binom{k}{d}$ nodes with depth d.



Binomial trees of order 0 through 5.
Top row: the recursive definition. Bottom row: the property claimed in part (b).

4. **[CS 373 Fall 2009]** For any language $L \in \Sigma^*$, let

$$Rotate(L) := \left\{ w \in \Sigma^* \;\middle|\; w = xy \text{ and } yx \in L \text{ for some strings } x, y \in \Sigma^* \right\}$$

   For example, $Rotate(\{\texttt{OOK!}, \texttt{OOKOOK}\}) = \{\texttt{OOK!}, \texttt{OK!O}, \texttt{K!OO}, \texttt{!OOK}, \texttt{OOKOOK}, \texttt{OKOOKO}, \texttt{KOOKOO}\}$.

   Prove that if L is a regular language, then $Rotate(L)$ is also a regular language. *[Hint: Remember the power of nondeterminism.]*

5. Herr Professor Doktor Georg von den Dschungel has a 24-node binary tree, in which every node is labeled with a unique letter of the German alphabet, which is just like the English alphabet with four extra letters: **Ä**, **Ö**, **Ü**, and **ß**. (Don't confuse these with **A**, **O**, **U**, and **B**!) Preorder and postorder traversals of the tree visit the nodes in the following order:

   - Preorder: **B K Ü E H L Z I Ö R C ß T S O A Ä D F M N U G**
   - Postorder: **H I Ö Z R L E C Ü S O T A ß K D M U G N F Ä B**

   (a) List the nodes in George's tree in the order visited by an inorder traversal.

   (b) Draw George's tree.

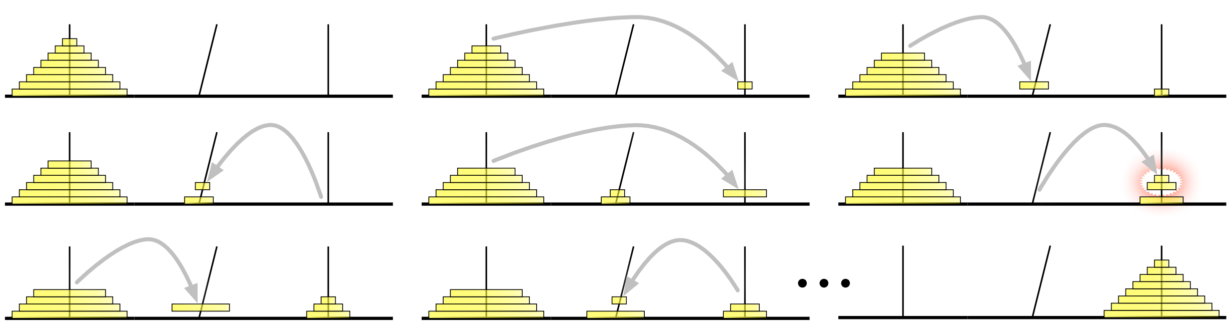
*6. ***[Extra credit]*** You may be familiar with the story behind the famous Tower of Hanoï puzzle, as related by Henri de Parville in 1884:

> In the great temple at Benares beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

A less familiar chapter in the temple's history is its brief relocation to Pisa in the early 13th century. The relocation was organized by the wealthy merchant-mathematician Leonardo Fibonacci, at the request of the Holy Roman Emperor Frederick II, who had heard reports of the temple from soldiers returning from the Crusades. The Towers of Pisa and their attendant monks became famous, helping to establish Pisa as a dominant trading center on the Italian peninsula.

Unfortunately, almost as soon as the temple was moved, one of the diamond needles began to lean to one side. To avoid the possibility of the leaning tower falling over from too much use, Fibonacci convinced the priests to adopt a more relaxed rule: ***Any number of disks on the leaning needle can be moved together to another needle in a single move.*** It was still forbidden to place a larger disk on top of a smaller disk, and disks had to be moved one at a time *onto* the leaning needle or between the two vertical needles.

Thanks to Fibonacci's new rule, the priests could bring about the end of the universe somewhat faster from Pisa then they could than could from Benares. Fortunately, the temple was moved from Pisa back to Benares after the newly crowned Pope Gregory IX excommunicated Frederick II, making the local priests less sympathetic to hosting foreign heretics with strange mathematical habits. Soon afterward, a bell tower was erected on the spot where the temple once stood; it too began to lean almost immediately.



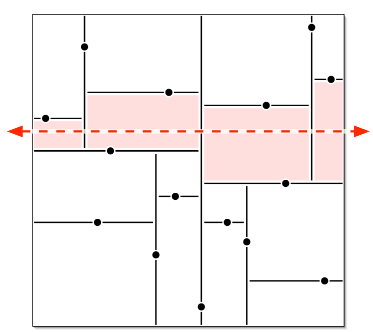
The Towers of Pisa. In the fifth move, two disks are taken off the leaning needle.

Describe an algorithm to transfer a stack of n disks from one *vertical* needle to the other *vertical* needle, using the smallest possible number of moves. *Exactly* how many moves does your algorithm perform?

- For this and all future homeworks, groups of up to three students can submit (or present) a single common solution. Please remember to write the names of all group members on every page.
- **Please fill out the online input survey linked from the course web page no later than Thursday, January 28.** Among other things, this survey asks you to identify the other members of your HW1 group, so that we can partition the class into presentation clusters without breaking up your group. We will announce the presentation clusters on Friday, January 29.
- Students in **Cluster 1** will present their solutions to Jeff or one of the TAs, on Tuesday or Wednesday of the due date (February 2 or February 3), instead of submitting written solutions. **Each homework group in Cluster 1 must sign up for a 30-minute time slot no later than Monday, February 1.** Signup sheets will be posted at 3303 Siebel Center ('The Theory Lab') later this week. Please see the course web page for more details.

---

1. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the points as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points *as evenly as possible* by passing through a median point in the interior of the box (*not* on its boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



A kd-tree for 15 points. The dashed line crosses the four shaded cells.

   (a) How many cells are there, as a function of n? Prove your answer is correct.

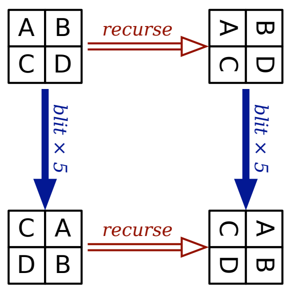
   (b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k.

   (c) Suppose we have n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

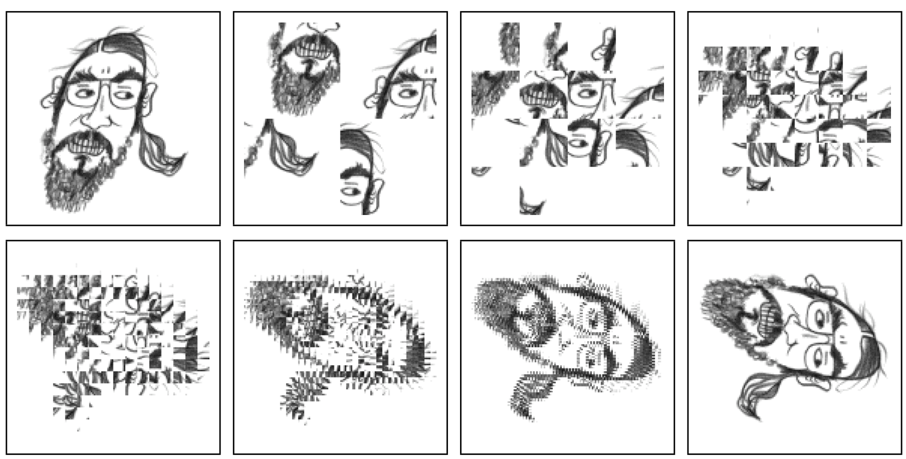
   (d) Describe an analyze an efficient algorithm that counts, given a kd-tree storing n points, the number of points that lie inside a rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

2. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

   Suppose we want to rotate an $n \times n$ pixel map 90° clockwise. One way to do this, at least when n is a power of two, is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. Alternately, we could *first* recursively rotate the blocks and *then* blit them into place.



Two algorithms for rotating a pixel map.
Solid arrows indicate blitting the blocks into place; hollow arrows indicate recursively rotating the blocks.



The first rotation algorithm (blit then recurse) in action.

   (a) Prove that both versions of the algorithm are correct when n is a power of two.

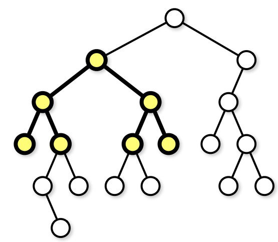
   (b) *Exactly* how many blits does the algorithm perform when n is a power of two?

   (c) Describe how to modify the algorithm so that it works for arbitrary n, not just powers of two. How many blits does your modified algorithm perform?

   (d) What is your algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

   (e) What if a $k \times k$ blit takes only $O(k)$ time?

3. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

CS 473: Undergraduate Algorithms, Spring 2010
Homework 2

Written solutions due Tuesday, February 9, 2010 at noon

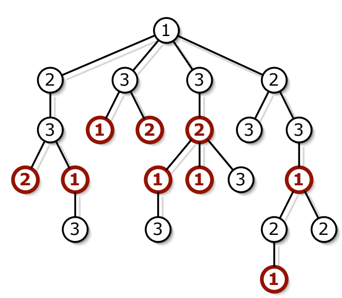
- Roughly 1/3 of the students will give oral presentations of their solutions to the TAs. ***You should have received an email telling you whether you are expected to present this homework.*** Please see the course web page for further details.
- Groups of up to three students may submit a common solution. Please clearly write every group member's name and NetID on every page of your submission. Please start your solution to each numbered problem on a new sheet of paper. Please ***don't*** staple solutions for different problems together.

---

1. A ***palindrome*** is a string that reads the same forwards and backwards, like x, pop, noon, redivider, or "sator arepo tenet opera rotas", Describe and analyze an algorithm to find the length of the longest subsequence of a given string that is also a palindrome. For example, the longest palindrome subsequence of **M**A**H**D**Y**NA**M**ICP**RO**G**R**AMZLET**M**ESHOW**Y**OUT**H**E**M** is MHYMRORMYHM, so given that string as input, your algorithm should output the number 11.

2. Oh, no! You have been appointed as the gift czar for Giggle, Inc.'s annual mandatory holiday party! The president of the company, who is certifiably insane, has declared that every Giggle employee must receive one of three gifts: (1) an all-expenses-paid six-week vacation anywhere in the world, (2) an all-the-pancakes-you-can-eat breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. Corporate regulations prohibit any employee from receiving the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy. How do you decide what gifts everyone gets if you want to minimize the number of people that get fired?

   More formally, suppose you are given a rooted tree T, representing the company hierarchy. You want to label each node in T with an integer 1, 2, or 3, such that every node has a different label from its parent.. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T. (Your algorithm does *not* have to compute the actual best labeling—just its cost.)



A tree labeling with cost 9. Bold nodes have smaller labels than their parents.
This is ***not*** the optimal labeling for this tree.

3. After graduating from UIUC, you have decided to join the Wall Street Bank ***Boole Long Live***. The managing director of the bank, Eloob Egroeg, is a genius mathematician who worships George Boole[1] every morning before leaving for the office. The first day of every hired employee is a 'solve-or-die' day where s/he has to solve one of the problems posed by Eloob within 24 hours. Those who fail to solve the problem are fired immediately!

   Entering into the bank for the first time, you notice that the offices of the employees are organized in a straight row, with a large "T" or "F" written on the door of each office. Furthermore, between each adjacent pair of offices, there is a board marked by one of the symbols $\wedge$, $\vee$, or $\oplus$. When you ask about these arcane symbols, Eloob confirms that T and F represent the boolean values 'true' and 'false', and the symbols on the boards represent the standard boolean operators AND, OR, and XOR. He also explains that these letters and symbols describe whether certain combinations of employees can work together successfully. At the start of any new project, Eloob hierarchically clusters his employees by adding parentheses to the sequence of symbols, to obtain an unambiguous boolean expression. The project is successful if this parenthesized boolean expression evaluates to T.

   For example, if the bank has three employees, and the sequence of symbols on and between their doors is $T \wedge F \oplus T$, there is exactly one successful parenthesization scheme: $(T \wedge (F \oplus T))$. However, if the list of door symbols is $F \wedge T \oplus F$, there is no way to add parentheses to make the project successful.

   Eloob finally poses your solve-or-die question: Describe and algorithm to decide whether a given sequence of symbols can be parenthesized so that the resulting boolean expression evaluates to T. The input to your algorithm is an array $S[0..2n]$, where $S[i] \in \{T, F\}$ when i is even, and $S[i] \in \{\vee, \wedge, \oplus\}$ when i is odd.

[1] 1815-1864, The inventor of Boolean Logic

- For this and all future homeworks, groups of up to three students can submit (or present) a single common solution. Please remember to write the names of all group members on every page.
- Students in **Cluster 3** will present their solutions to Jeff or one of the TAs, on Tuesday or Wednesday of the due date (February 16 or February 17), instead of submitting written solutions. **Each homework group in Cluster 3 must sign up for a 30-minute time slot no later than Monday, February 15.** Signup sheets will be posted at 3304 Siebel Center ('The Theory Lab') later this week. Please see the course web page for more details.

1. You saw in class a correct greedy algorithm for finding the maximum number of non-conflicting courses from a given set of possible courses. This algorithm repeatedly selects the class with the earliest completion time that does not conflict with any previously selected class.

   Below are four alternative greedy algorithms. For each algorithm, either prove that the algorithm constructs an optimal schedule, or give a concrete counterexample showing that the algorithm is suboptimal.

   (a) Choose the course that *ends latest*, discard all conflicting classes, and recurse.

   (b) Choose the course that *starts first*, discard all conflicting classes, and recurse.

   (c) Choose the course with *shortest duration*, discard all conflicting classes, and recurse.

   (d) Choose a course that *conflicts with the fewest other courses* (breaking ties arbitrarily), discard all conflicting classes, and recurse.

2. You have been given the task of designing an algorithm for vending machines that computes the smallest number of coins worth any given amount of money. Your supervisors at The Area 51 Soda Company are anticipating a hostile takeover of earth by an advanced alien race that uses an unknown system of currency. So your algorithm must be as general as possible so that it will work with the alien system, whatever it turns out to be.

   Given a quantity of money x, and a set of coin denominations $b_1, \ldots, b_k$, your algorithm should compute how to make change for x with the fewest number of coins. For example, if you use the US coin denominations (1¢, 5¢, 10¢, 25¢, 50¢, and 100¢), the optimal way to make 17¢ in change uses 4 coins: one dime (10¢), one nickel (5¢), and two pennies (1¢).

   (a) Show that the following greedy algorithm does *not* work for all currency systems: If $x = 0$, do nothing. Otherwise, find the largest denomination $c \leq x$, issue one c-cent coin, and recursively give $x - c$ cents in change.

   (b) Now suppose that the system of currency you are concerned with only has coins in powers of some base b. That is, the coin denominations are $b^0, b^1, b^2, \ldots, b^k$. Show that the greedy algorithm described in part (a) does make optimal change in this currency system.

   (c) Describe and analyze an algorithm that computes optimal change for *any* set of coin denominations. (You may assume the aliens' currency system includes a 1-cent coin, so that making change is always possible.)

3. Suppose you have just purchased a new type of hybrid car that uses fuel extremely efficiently, but can only travel 100 miles on a single battery. The car's fuel is stored in a single-use battery, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and can only be installed by licensed battery-replacement technicians. Thus, even if you decide to replace your battery early, you must still pay full price for the new battery to be installed. Moreover, because these batteries are in high demand, no one can afford to own more than one battery at a time.

   Suppose you are trying to get from San Francisco to New York City on the new Inter-Continental Super-Highway, which runs in a direct line between these two cities. There are several fueling stations along the way; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every fueling station on the ICSH. Given this information, how do you decide the best places to stop for fuel?

   More formally, suppose you are given two arrays $D[1..n]$ and $C[1..n]$, where $D[i]$ is the distance from the start of the highway to the ith station, and $C[i]$ is the cost to replace your battery at the ith station. Assume that your trip starts and ends at fueling stations (so $D[1] = 0$ and $D[n]$ is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

   (a) Describe and analyze a greedy algorithm to find the minimum number of refueling stops needed to complete your trip. Don't forget to prove that your algorithm is correct.

   (b) But what you really want to minimize is the total *cost* of travel. Show that your greedy algorithm in part (a) does *not* produce an optimal solution when extended to this setting.

   (c) Describe a dynamic programming algorithm to compute the locations of the fuel stations you should stop at to minimize the cost of travel.

1. Suppose we want to write an efficient function SHUFFLE(n) that returns a permutation of the set $\{1, 2, \ldots, n\}$ chosen uniformly at random.

   (a) Prove that the following algorithm is ***not*** correct. *[Hint: Consider the case $n = 3$.]*

```
SHUFFLE(n):
  for i ← 1 to n
      π[i] ← i
  for i ← 1 to n
      swap π[i] ↔ π[RANDOM(n)]
  return π[1..n]
```

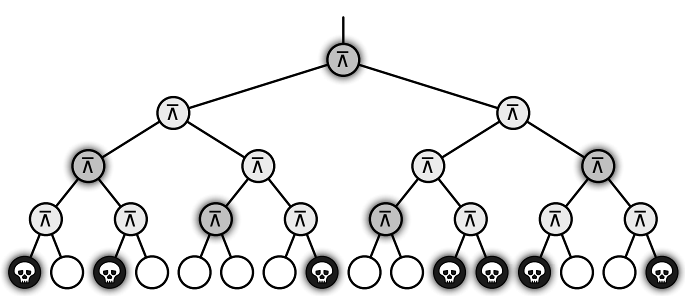
   (b) Consider the following implementation of SHUFFLE.

```
SHUFFLE(n):
  for i ← 1 to n
      π[i] ← NULL
  for i ← 1 to n
      j ← RANDOM(n)
      while (π[j] != NULL)
          j ← RANDOM(n)
      π[j] ← i
  return π[1..n]
```

   Prove that this algorithm is correct. What is its expected running time?

   (c) Describe and analyze an implementation of SHUFFLE that runs in $O(n)$ time. (An algorithm that runs in $O(n)$ *expected* time is fine, but $O(n)$ *worst-case* time is possible.)

2. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it's white, you will live forever. You move first, so Death gets the last turn.



   You can decide whether it's worth playing or not as follows. Imagine that the tree is a Boolean circuit whose inputs are specified at the leaves: white and black represent TRUE and FALSE inputs, respectively. Each internal node in the tree is a NAND gate that gets its input from its children and passes its output to its parent. (Recall that a NAND gate outputs FALSE if and only if both its inputs are TRUE.) If the output at the top of the tree is TRUE, then you can win and live forever! If the output at the top of the tree is FALSE, you should challenge Death to a game of Twister instead. Or maybe Battleship.

(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

(b) Unfortunately, Death won't give you enough time to look at every node in the tree. Describe a *randomized* algorithm that determines whether you can win in $O(3^n)$ expected time. *[Hint: Consider the case $n = 1$.]*

*(c) **[Extra credit]** Describe and analyze a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. *[Hint: You may not need to change your algorithm from part (b) at all!]*

3. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- MakeQueue: Return a new priority queue containing the empty set.
- FindMin(Q): Return the smallest element of Q (if any).
- DeleteMin(Q): Remove the smallest element in Q (if any).
- Insert(Q, x): Insert element x into Q, if it is not already there.
- DecreaseKey(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
- Delete(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
- Meld(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. Meld can be implemented using the following randomized algorithm:

```
MELD(Q1, Q2):
  if Q1 is empty return Q2
  if Q2 is empty return Q1

  if key(Q1) > key(Q2)
        swap Q1 ↔ Q2

  with probability 1/2
        left(Q1) ← MELD(left(Q1), Q2)
  else
        right(Q1) ← MELD(right(Q1), Q2)

  return Q1
```

(a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of Meld(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if each left/right choice is made with equal probability?]*

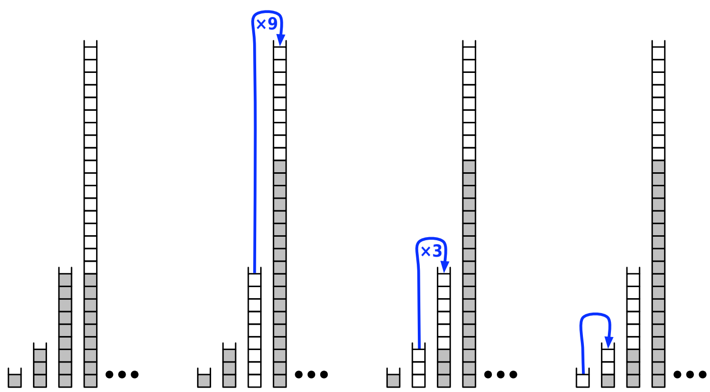
(b) Show that each of the other meldable priority queue operations can be implemented with at most one call to Meld and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ expected time.)

1. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. The user always pushes and pops elements from the smallest stack S_0. However, before any element can be pushed onto any full stack S_i, we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move all its members to S_{i+2}.) Similarly, before any element can be popped from any empty stack S_i, we first pop 3^i elements from S_{i+1} and push them onto S_i to make room. (Thus, if S_{i+1} is already empty, we first recursively fill it by popping elements from S_{i+2}.) Moving a single element from one stack to another takes $O(1)$ time.

   Here is pseudocode for the multistack operations MSPush and MSPop. The internal stacks are managed with the subroutines Push and Pop.

```
MPush(x):
  i ← 0
  while S_i is full
      i ← i + 1
  while i > 0
      i ← i − 1
      for j ← 1 to 3^i
          Push(S_{i+1}, Pop(S_i))
  Push(S_0, x)
```

```
MPop(x):
  i ← 0
  while S_i is empty
      i ← i + 1
  while i > 0
      i ← i − 1
      for j ← 1 to 3^i
          Push(S_i, Pop(S_{i+1}))
  return Pop(S_0)
```



Making room in a multistack, just before pushing on a new element.

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?

   (b) Prove that if the user never pops anything from the multistack, the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack during its lifetime.

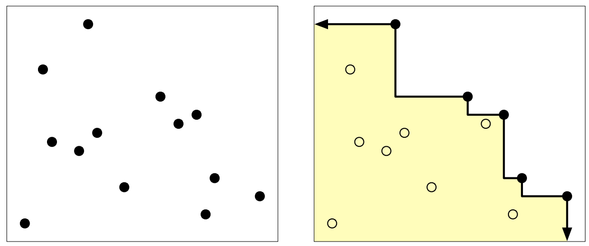
   (c) Prove that in any intermixed sequence of pushes and pops, each push or pop operation takes $O(\log n)$ amortized time, where n is the maximum number of elements in the multistack during its lifetime.

2. Design and analyze a simple data structure that maintains a list of integers and supports the following operations.

   - CREATE( ) creates and returns a new list
   - PUSH(L, x) appends x to the end of L
   - POP(L) deletes the last entry of L and returns it
   - LOOKUP(L, k) returns the kth entry of L

   Your solution may use these primitive data structures: arrays, balanced binary search trees, heaps, queues, single or doubly linked lists, and stacks. If your algorithm uses *anything* fancier, you must give an explicit implementation. Your data structure must support all operations in amortized constant time. In addition, your data structure must support each LOOKUP in *worst-case* $O(1)$ time. At all times, the size of your data structure must be linear in the number of objects it stores.

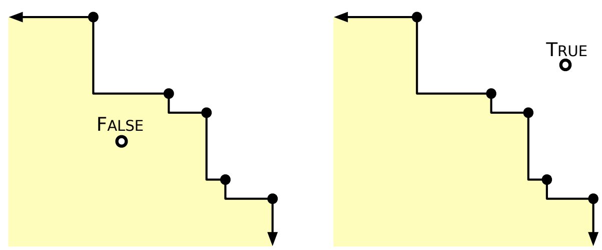
3. Let P be a set of n points in the plane. The *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.



A set of points in the plane and its staircase (shaded).

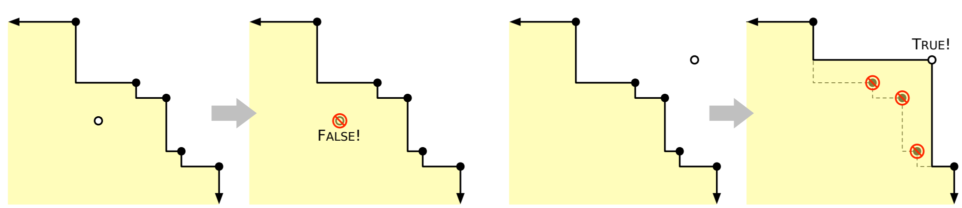
(a) Describe an algorithm to compute the staircase of a set of n points in $O(n \log n)$ time.

(b) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm ABOVE?(x, y) that returns TRUE if the point (x, y) is above the staircase, or FALSE otherwise. Your data structure should use $O(n)$ space, and your ABOVE? algorithm should run in $O(\log n)$ time.



Two staircase queries.

(c) Describe and analyze a data structure that maintains a staircase as new points are inserted. Specifically, your data structure should support a function INSERT(x, y) that adds the point (x, y) to the underlying point set and returns TRUE or FALSE to indicate whether the staircase of the set has changed. Your data structure should use $O(n)$ space, and your INSERT algorithm should run in $O(\log n)$ amortized time.



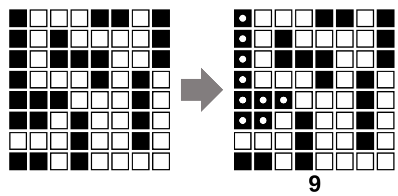
Two staircase insertions.

CS 473: Undergraduate Algorithms, Spring 2010
Homework 6

Written solutions due Tuesday, March 16, 2010 at noon

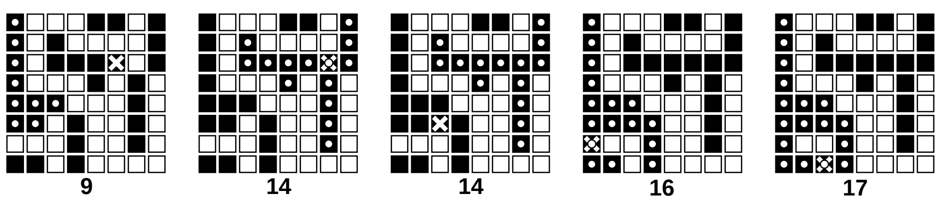
---

1. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$. For example, given the bitmap below as input, your algorithm should return the number 9, because the largest conected black component (marked with white dots on the right) contains nine pixels.



   (b) Design and analyze an algorithm Blacken(i, j) that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

   For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the Blacken algorithm.



   (c) What is the *worst-case* running time of your Blacken algorithm?

2. Suppose you are given a graph G with weighted edges and a minimum spanning tree T of G.

   (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is *increased*.

   (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is *decreased*.

   In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. *[Hint: Consider the cases $e \in T$ and $e \notin T$ separately.]*

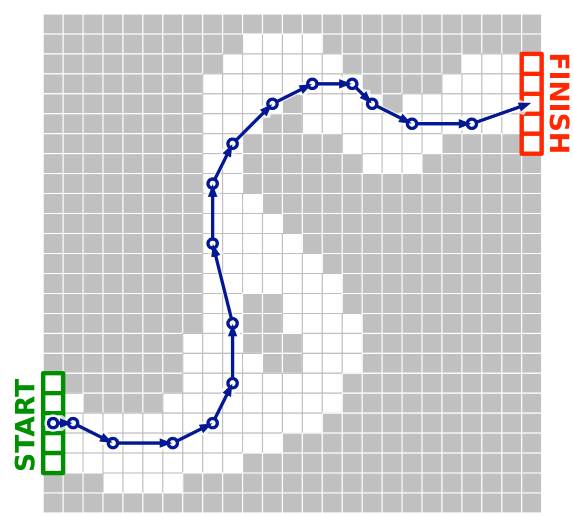
3. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game of uncertain origin that Jeff played on the bus in 5th grade.[1] The game is played using a racetrack drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

   Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. The initial position is an arbitrary point on the starting line, chosen by the player; the initial velocity is always $(0,0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must lie inside the track; otherwise, the car crashes and that player immediately loses the race. The first car that reaches a position ***on*** the finish line is the winner.

   Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the 'starting line' is the first column, and the 'finish line' is the last column.

   Describe and analyze an algorithm to find *the minimum number of steps* required to move a car from the starting line to the finish line according to these rules, given a racetrack bitmap as input. *[Hint: Build a graph. What are the vertices? What are the edges? What problem is this?]*

velocity	position
$(0,0)$	$(1,5)$
$(1,0)$	$(2,5)$
$(2,-1)$	$(4,4)$
$(3,0)$	$(7,4)$
$(2,1)$	$(9,5)$
$(1,2)$	$(10,7)$
$(0,3)$	$(10,10)$
$(-1,4)$	$(9,14)$
$(0,3)$	$(9,17)$
$(1,2)$	$(10,19)$
$(2,2)$	$(12,21)$
$(2,1)$	$(14,22)$
$(2,0)$	$(16,22)$
$(1,-1)$	$(17,21)$
$(2,-1)$	$(19,20)$
$(3,0)$	$(22,20)$
$(3,1)$	$(25,21)$



A 16-step Racetrack run, on a 25×25 track. This is *not* the shortest run on this track.

[1]The actual game Jeff played was a bit more complicated than the version described in this problem. In particular, the track was a freeform curve, and by default, the entire line segment traversed by a car in a single step had to lie entirely inside the track. If a car did run off the track, it started its next turn with velocity zero, at the legal grid point closest to where it first crossed the track boundary.

1. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, "Get out... while... you...", thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

   You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

   On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can't stand each other's company, so you'll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

   You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger's heart, you wisely decide to leave the park immediately.

   Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along the path is as small as possible. *Be sure to account for **both** the vertex probabilities **and** the edge probabilities!*

2. In this problem we will discover how you, too, can be employed by Wall Street and cause a major economic collapse! The *arbitrage* business is a money-making scheme that takes advantage of differences in currency exchange. In particular, suppose that 1 US dollar buys 120 Japanese yen; 1 yen buys 0.01 euros; and 1 euro buys 1.2 US dollars. Then, a trader starting with \$1 can convert his money from dollars to yen, then from yen to euros, and finally from euros back to dollars, ending with \$1.44! The cycle of currencies $\$ \to ¥ \to € \to \$$ is called an ***arbitrage cycle***. Of course, finding and exploiting arbitrage cycles before the prices are corrected requires extremely fast algorithms.

   Suppose n different currencies are traded in your currency market. You are given the matrix $R[1..n, 1..n]$ of exchange rates between every pair of currencies; for each i and j, one unit of currency i can be traded for $R[i, j]$ units of currency j. (Do *not* assume that $R[i, j] \cdot R[j, i] = 1$.)

   (a) Describe an algorithm that returns an array $V[1..n]$, where $V[i]$ is the maximum amount of currency i that you can obtain by trading, starting with one unit of currency 1, assuming there are no arbitrage cycles.

   (b) Describe an algorithm to determine whether the given matrix of currency exchange rates creates an arbitrage cycle.

   (c) Modify your algorithm from part (b) to actually return an arbitrage cycle, if it exists.

3. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero. In this problem, you will develop an algorithm to compute shortest paths between *every* pair of vertices. The output from this algorithm is a two-dimensional array $dist[1..V, 1..V]$, where $dist[i, j]$ is the length of the shortest path from vertex i to vertex j.

   (a) How could we delete some node v from this graph, without changing the shortest-path distance between any other pair of nodes? Describe an algorithm that constructs a directed graph $G' = (V', E')$ with weighted edges, where $V' = V \setminus \{v\}$, and the shortest-path distance between any two nodes in G' is equal to the shortest-path distance between the same two nodes in G. For full credit, your algorithm should run in $O(V^2)$ time.

   (b) Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances from v to every other node, and from every other node to v, in the original graph G. For full credit, your algorithm should run in $O(V^2)$ time.

   (c) Combine parts (a) and (b) into an algorithm that finds the shortest paths between *every* pair of vertices in the graph. For full credit, your algorithm should run in $O(V^3)$ time.

   The lecture notes (along with most algorithms textbooks and Wikipedia) describe a dynamic programming algorithm due to Floyd and Warshall that computes all shortest paths in $O(V^3)$ time. This is *not* that algorithm.

CS 473: Undergraduate Algorithms, Spring 2010
Homework 8

Written solutions due Tuesday, April 20, 2010 in class.

---

1. Suppose you have already computed a maximum (s,t)-flow f in a flow network G with integer capacities. Let k be an arbitrary positive integer, and let e be an arbitrary edge in G whose capacity is at least k.

   (a) Suppose we *increase* the capacity of e by k units. Describe and analyze an algorithm to update the maximum flow.

   (b) Now suppose we *decrease* the capacity of e by k units. Describe and analyze an algorithm to update the maximum flow.

   For full credit, both algorithms should run in $O(Ek)$ time. *[Hint: First consider the case $k = 1$.]*

2. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

   Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

3. A *cycle cover* of a given directed graph $G = (V, E)$ is a set of vertex-disjoint cycles that cover all the vertices. Describe and analyze an efficient algorithm to find a cycle cover for a given graph, or correctly report that none exists. *[Hint: Use ipartite atching!]*

1. We say that an array $A[1..n]$ is *k-sorted* if it can be divided into k blocks, each of size n/k, such that the elements in each block are larger than the elements in earlier blocks, and smaller than elements in later blocks. The elements within each block need not be sorted.

   For example, the following array is 4-sorted:

1	2	4	3	7	6	8	5	10	11	9	12	15	13	16	14

   (a) Describe an algorithm that k-sorts an arbitrary array in time $O(n \log k)$.

   (b) Prove that any comparison-based k-sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst case.

   (c) Describe an algorithm that completely sorts an already k-sorted array in time $O(n \log(n/k))$.

   (d) Prove that any comparison-based algorithm to completely sort a k-sorted array requires $\Omega(n \log(n/k))$ comparisons in the worst case.

   In all cases, you can assume that n/k is an integer and that $n! \approx \left(\frac{n}{e}\right)^n$.

2. Recall the nuts and bolts problem from the first randomized algorithms lecture. You are given n nuts and n bolts of different sizes. Each nut matches exactly one bolt and vice versa. The nuts and bolts are all almost exactly the same size, so we can't tell if one bolt is bigger than the other, or if one nut is bigger than the other. If we try to match a nut with a bolt, however, we will discover either that the nut is too big, the nut is too small, or the nut is just right for the bolt. The goal was to find the matching nut for every bolt.

   Now consider a relaxed version of the problem where the goal is to find the matching nuts for *half* of the bolts, or equivalently, to find $n/2$ matched nut-bolt pairs. (It doesn't matter *which* $n/2$ nuts and bolts are matched.) Prove that any deterministic algorithm to solve this problem must perform $\Omega(n \log n)$ nut-bolt tests in the worst case.

3. UIUC has just finished constructing the new Reingold Building, the tallest dormitory on campus. In order to determine how much insurance to buy, the university administration needs to determine the highest safe floor in the building. A floor is consdered *safe* if ~~a drunk student~~ ***an egg*** can fall from a window on that floor and land without breaking; if the egg breaks, the floor is considered *unsafe*. Any floor that is higher than an unsafe floor is also considered unsafe. The only way to determine whether a floor is safe is to drop an egg from a window on that floor.

   You would like to find the lowest unsafe floor L by performing as few tests as possible; unfortunately, you have only a very limited supply of eggs.

   (a) Prove that if you have only one egg, you can find the lowest unsafe floor with L tests. *[Hint: Yes, this is trivial.]*

   (b) Prove that if you have only one egg, you must perform at least L tests in the worst case. In other words, prove that your algorithm from part (a) is optimal. *[Hint: Use an adversary argument.]*

   (c) Describe an algorithm to find the lowest unsafe floor using *two* eggs and only $O(\sqrt{L})$ tests. *[Hint: Ideally, each egg should be dropped the same number of times. How many floors can you test with n drops?]*

   (d) Prove that if you start with two eggs, you must perform at least $\Omega(\sqrt{L})$ tests in the worst case. In other words, prove that your algorithm from part (c) is optimal.

This homework is practice only. However, there will be at least one NP-hardness problem on the final exam, so working through this homework is *strongly* recommended. Students/groups are welcome to submit solutions for feedback (but not credit) in class on May 4, after which we will publish official solutions.

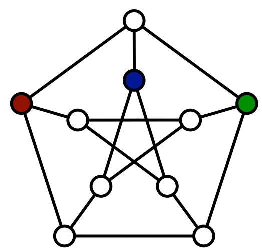
1. Recall that 3SAT asks whether a given boolean formula in conjunctive normal form, with exactly three literals in each clause, is satisfiable. In class we proved that 3SAT is NP-complete, using a reduction from CircuitSAT.

   Now consider the related problem **2SAT**: Given a boolean formula in conjunctive normal form, with exactly *two* literals in each clause, is the formula satisfiable? For example, the following boolean formula is a valid input to 2SAT:

   $$(x \vee y) \wedge (y \vee \overline{z}) \wedge (\overline{x} \vee z) \wedge (\overline{w} \vee y).$$

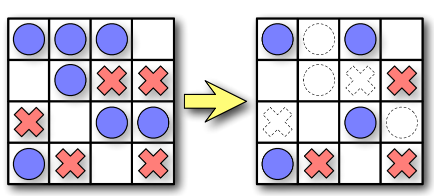
   Either prove that 2SAT is NP-hard or describe a polynomial-time algorithm to solve it. *[Hint: Recall that $(x \vee y) \equiv (\overline{x} \rightarrow y)$, and build a graph.]*

2. Let $G = (V, E)$ be a graph. A *dominating set* in G is a subset S of the vertices such that every vertex in G is either in S or adjacent to a vertex in S. The DominatingSet problem asks, given a graph G and an integer k as input, whether G contains a dominating set of size k. Either prove that this problem is NP-hard or describe a polynomial-time algorithm to solve it.

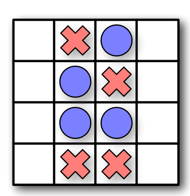


A dominating set of size 3 in the Peterson graph.

3. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

   Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

This exam lasts 120 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet with your answers.

1. Each of these ten questions has one of the following five answers:

   A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$

   Choose the correct answer for each question. Each correct answer is worth $+1$ point; each incorrect answer is worth $-1/2$ point; and each "I don't know" is worth $+1/4$ point. Negative scores will be recorded as 0.

   (a) What is $\dfrac{3}{n} + \dfrac{n}{3}$?

   (b) What is $\displaystyle\sum_{i=1}^{n} \frac{i}{n}$?

   (c) What is $\sqrt{\displaystyle\sum_{i=1}^{n} i}$?

   (d) How many bits are required to write the number $n!$ (the factorial of n) in binary?

   (e) What is the solution to the recurrence $E(n) = E(n-3) + 17n$?

   (f) What is the solution to the recurrence $F(n) = 2F(n/4) + 6n$?

   (g) What is the solution to the recurrence $G(n) = 9G(n/9) + 9n$?

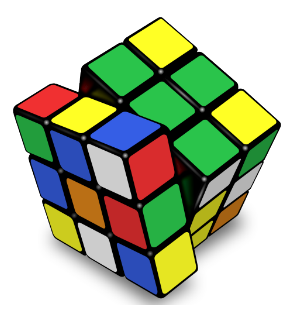
   (h) What is the worst-case running time of quicksort?

   (i) Let $X[1..n, 1..n]$ be a fixed array of numbers. Consider the following recursive function:

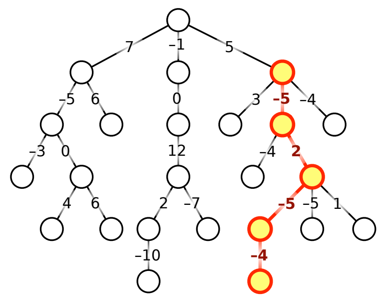
$$
WTF(i,j) = \begin{cases} 0 & \text{if } \min\{i,j\} \le 0 \\ -\infty & \text{if } \max\{i,j\} > n \\ X[i,j] + \max\left\{ \begin{array}{l} WTF(i-2, j+1) \\ WTF(i-2, j-1) \\ WTF(i-1, j-2) \\ WTF(i+1, j-2) \end{array} \right\} & \text{otherwise} \end{cases}
$$

   How long does it take to compute $WTF(n,n)$ using dynamic programming?

   (j) The Rubik's Cube is a mechanical puzzle invented in 1974 by Ernő Rubik, a Hungarian professor of architecture. The puzzle consists of a $3 \times 3 \times 3$ grid of 'cubelets', whose faces are covered with stickers in six different colors. In the puzzle's solved state, each face of the puzzle is one solid color. A mechanism inside the puzzle allows any face of the cube to be freely turned (as shown on the right). The puzzle can be scrambled by repeated turns. Given a scrambled Rubik's Cube, how long does it take to find the *shortest* sequence of turns that returns the cube to its solved state?



2. Let T be a rooted tree with integer weights on its edges, which could be positive, negative, or zero. The weight of a path in T is the sum of the weights of its edges. Describe and analyze an algorithm to compute the minimum weight of any path from a node in T down to one of its descendants. It is not necessary to compute the actual minimum-weight path; just its weight. For example, given the tree shown below, your algorithm should return the number -12.



The minimum-weight downward path in this tree has weight −12.

3. Describe and analyze efficient algorithms to solve the following problems:

   (a) Given a set of n integers, does it contain two elements a, b such that $a + b = 0$?

   (b) Given a set of n integers, does it contain three elements a, b, c such that $a + b = c$?

4. A *common supersequence* of two strings A and B is another string that includes both the characters of A in order and the characters of B in order. Describe and analyze and algorithm to compute the length of the *shortest* common supersequence of two strings $A[1..m]$ and $B[1..n]$. You do not need to compute an actual supersequence, just its length.

   For example, if the input strings are `ANTHROHOPOBIOLOGICAL` and `PRETERDIPLOMATICALLY`, your algorithm should output 31, because a shortest common supersequence of those two strings is `PREANTHEROHODPOBIOPLOMATGICALLY`.

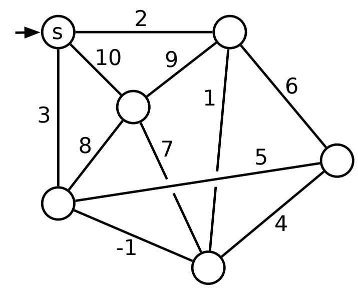
5. *[Taken directly from HBS0.]* Recall that the *Fibonacci numbers* F_n are recursively defined as follows: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for every integer $n \geq 2$. The first few Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$.

   Prove that any non-negative integer can be written as the sum of distinct *non-consecutive* Fibonacci numbers. That is, if any Fibonacci number F_n appears in the sum, then its neighbors F_{n-1} and F_{n+1} do not. For example:

$$\begin{aligned} 88 &= 55 + 21 + 8 + 3 + 1 &&= F_{10} + F_8 + F_6 + F_4 + F_2 \\ 42 &= 34 + 8 &&= F_9 + F_6 \\ 17 &= 13 + 3 + 1 &&= F_7 + F_4 + F_2 \end{aligned}$$

This exam lasts 120 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet with your answers.

1. Find the following spanning trees for the weighted graph shown below.



   (a) A depth-first spanning tree rooted at s.
   (b) A breadth-first spanning tree rooted at s.
   (c) ~~A shortest-path tree rooted at s.~~ ***Oops!***
   (d) A minimum spanning tree.

   You do *not* need to justify your answers; just clearly indicate the edges of each spanning tree in your answer booklet. Yes, one of the edges has negative weight.

2. *[Taken directly from HBS 6.]* An *Euler tour* of a graph G is a walk that starts and ends at the same vertex and traverses every edge of G exactly once. ***Prove*** that a connected undirected graph G has an Euler tour if and only if every vertex in G has even degree.

3. You saw in class that the standard algorithm to INCREMENT a binary counter runs in $O(1)$ amortized time. Now suppose we also want to support a second function called RESET, which resets all bits in the counter to zero.

   Here are the INCREMENT and RESET algorithms. In addition to the array $B[\ldots]$ of bits, we now also maintain the index of the most significant bit, in an integer variable *msb*.

```
INCREMENT(B[0..∞], msb):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
  if i > msb
      msb ← i
```

```
RESET(B[0..∞], msb):
  for i ← 0 to msb
      B[i] ← 0
  msb ← 0
```

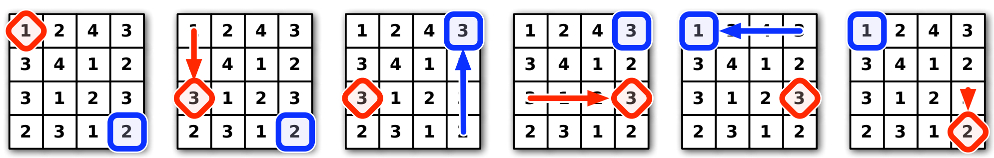
   In parts (a) and (b), let n denote the number currently stored in the counter.

   (a) What is the worst-case running time of INCREMENT, as a function of n?
   (b) What is the worst-case running time of RESET, as a function of n?
   (c) ***Prove*** that in an *arbitrary* intermixed sequence of INCREMENT and RESET operations, the amortized time for each operation is $O(1)$.

4. The following puzzle was invented by the infamous Mongolian puzzle-warrior Vidrach Itky Leda in the year 1473. The puzzle consists of an $n \times n$ grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

   In a single turn, you may move either token up, right, down, or left *by a distance determined by the **other** token*. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move a token off the grid or to the same square as the other token.

   Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle below, your algorithm would return the number 5.



A five-move solution for a 4×4 Vidrach Itky Leda puzzle.

5. Suppose you are given an array $X[1..n]$ of real numbers chosen independently and uniformly at random from the interval $[0,1]$. An array entry $X[i]$ is called a *local maximum* if it is larger than its neighbors $X[i-1]$ and $X[i+1]$ (if they exist).

   What is the *exact* expected number of local maxima in X? ***Prove*** that your answer is correct. *[Hint: Consider the special case $n = 3$.]*

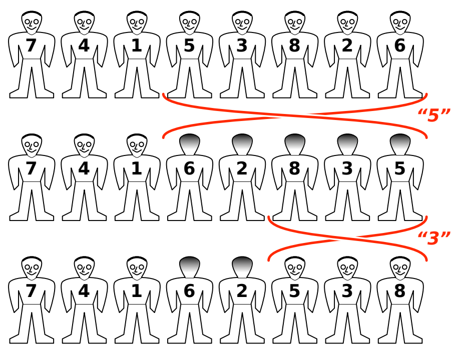
0.7	0.3	**1.0**	0.1	0.0	0.5	**0.6**	0.2	0.4	**0.9**	0.8

A randomly filled array with 4 local maxima.

This exam lasts 180 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet with your answers.

1. Choreographer Michael Flatley has hired a new dance company to perform his latest Irish step-dancing extravaganza. At their first practice session, the new dancers line up in a row on stage and practice a movement called the *Flatley Flip*: Whenever Mr. Flatley calls out any positive integer k, the k rightmost dancers rotate 180 degrees as a group, so that their order in the line is reversed.

   Each dancer wears a shirt with a positive integer printed on the front and back; different dancers have different numbers. Mr. Flatley wants to rearrange the dancers, using only a sequence of Flatley Flips, so that these numbers are sorted from left to right in increasing order.



Two Flatley flips.

   (a) Describe an algorithm to sort an arbitrary row of n numbered dancers, using $O(n)$ Flatley flips. (After sorting, the dancers may face forward, backward, or some of each.) *Exactly* how many flips does your algorithm perform in the worst case?[1]

   (b) Describe an algorithm that sorts an arbitrary row of n numbered dancers *and ensures that all dancers are facing forward*, using $O(n)$ Flatley flips. *Exactly* how many flips does your algorithm perform in the worst case?[2]

2. You're in charge of choreographing a musical for your local community theater, and it's time to figure out the final pose of the big show-stopping number at the end. ("Streetcar!") You've decided that each of the n cast members in the show will be positioned in a big line when the song finishes, all with their arms extended and showing off their best spirit fingers.

   The director has declared that during the final flourish, each cast member must either point both their arms up or point both their arms down; it's your job to figure out who points up and who points down. Moreover, in a fit of unchecked power, the director has also given you a list of arrangements that will upset his delicate artistic temperament. Each forbidden arrangement is a subset of cast members paired with arm positions; for example: "Marge may not point her arms up while Ned and Apu point their arms down."

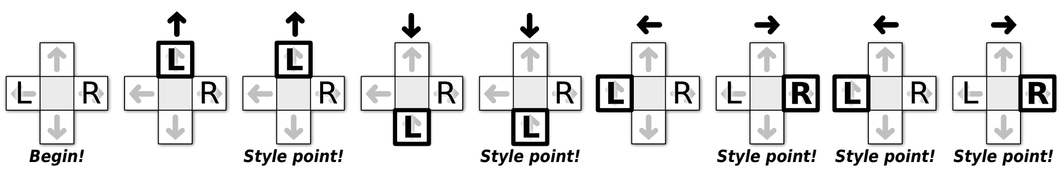
   Prove that finding an acceptable arrangement of arm positions is NP-hard. *[Hint: Describe a reduction from 3SAT.]*

[1]This is really a question about networking.
[2]This is really a question about mutating DNA.

3. ***Dance Dance Revolution*** is a dance video game, first introduced in Japan by Konami in 1998. Players stand on a platform marked with four arrows, pointing forward, back, left, and right, arranged in a cross pattern. During play, the game plays a song and scrolls a sequence of n arrows (←, ↑, ↓, or →) from the bottom to the top of the screen. At the precise moment each arrow reaches the top of the screen, the player must step on the corresponding arrow on the dance platform. (The arrows are timed so that you'll step with the beat of the song.)

   You are playing a variant of this game called "Vogue Vogue Revolution", where the goal is to play perfectly but move as little as possible. When an arrow reaches the top of the screen, if one of your feet is already on the correct arrow, you are awarded one style point for maintaining your current pose. If neither foot is on the right arrow, you must move one (and *only* one) of your feet from its current location to the correct arrow on the platform. If you ever step on the wrong arrow, or fail to step on the correct arrow, or move more than one foot at a time, all your style points are taken away and the games ends.

   How should you move your feet to maximize your total number of style points? For purposes of this problem, assume you always start with you left foot on ← and you right foot on →, and that you've memorized the entire sequence of arrows. For example, if the sequence is ↑↑↓↓←→←→, you can earn 5 style points by moving you feet as shown below:



   (a) ***Prove*** that for *any* sequence of n arrows, it is possible to earn at least $n/4 - 1$ style points.

   (b) Describe an efficient algorithm to find the maximum number of style points you can earn during a given VVR routine.[3] The input to your algorithm is an array $Arrow[1..n]$ containing the sequence of arrows. *[Hint: Build a graph!]*

4. It's almost time to show off your flippin' sweet dancing skills! Tomorrow is the big dance contest you've been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You've obtained an advance copy of the the list of n songs that the judges will play during the contest, in chronological order.

   You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly $Score[k]$ points, but then you will be physically unable to dance for the next $Wait[k]$ songs (that is, you cannot dance to songs $k+1$ through $k + Wait[k]$). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

   Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.[4]

[3]This is really a question about paging.
[4]This is really a question about processor scheduling.

5. You're organizing the First Annual UIUC Computer Science 72-Hour Dance Exchange, to be held all day Friday, Saturday, and Sunday. Several 30-minute sets of music will be played during the event, and a large number of DJs have applied to perform. You need to hire DJs according to the following constraints.

   - Exactly k sets of music must be played each day, and thus $3k$ sets altogether.
   - Each set must be played by a single DJ in a consistent music genre (ambient, bubblegum, dubstep, horrorcore, hyphy, trip-hop, Nitzhonot, Kwaito, J-pop, Nashville country, ...).
   - Each genre must be played at most once per day.
   - Each candidate DJ has given you a list of genres they are willing to play.
   - Each DJ can play at most three sets during the entire event.

   Suppose there are n candidate DJs and g different musical genres available. Describe and analyze an efficient algorithm that either assigns a DJ and a genre to each of the $3k$ sets, or correctly reports that no such assignment is possible.

6. You've been put in charge of putting together a team for the "Dancing with the Computer Scientists" international competition. Good teams in this competition must be capable of performing a wide variety of dance styles. You are auditioning a set of n dancing computer scientists, each of whom specializes in a particular style of dance.

   Describe an algorithm to determine in $O(n)$ time if *more than half* of the n dancers specialize exactly in the same dance style. The input to your algorithm is an array of n positive integers, where each integer identifies a style: 1 = ballroom, 2 = latin, 3 = swing, 4 = b-boy, 42 = contact improv, 101 = peanut butter jelly time, and so on. *[Hint: Remember the SELECT algorithm!]*

7. The party you are attending is going great, but now it's time to line up for ***The Algorithm March (アルゴリズムこうしん)***! This dance was originally developed by the Japanese comedy duo Itsumo Kokokara (いつもここから) for the children's television show PythagoraSwitch (ピタゴラスイッチ). The Algorithm March is performed by a line of people; each person in line starts a specific sequence of movements one measure later than the person directly in front of them. Thus, the march is the dance equivalent of a musical round or canon, like "Row Row Row Your Boat".

   Proper etiquette dictates that each marcher must know the person directly in front of them in line, lest a minor mistake during lead to horrible embarrassment between strangers. Suppose you are given a complete list of which people at your party know each other. ***Prove*** that it is NP-hard to determine the largest number of party-goers that can participate in the Algorithm March.[5]

[5]This is really a question about ninjas.

You may assume the following problems are NP-hard:

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

PlanarCircuitSat: Given a boolean circuit drawn in the plane so that no two wires cross, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinDominatingSet: Given an undirected graph G, what is the size of the smallest subset S of vertices such that every vertex in G is either in S or adjacent to a vertex in S?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

ChromaticNumber: Given an undirected graph G, what is the minimum number of colors needed to color its vertices, so that every edge touches vertices with two different colors?

MaxCut: Given a graph G, what is the size (number of edges) of the largest bipartite subgraph of G?

HamiltonianCycle: Given a graph G, is there a cycle in G that visits every vertex exactly once?

HamiltonianPath: Given a graph G, is there a path in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of n positive integers, can X be partitioned into $n/3$ three-element subsets, all with the same sum?

Minesweeper: Given a Minesweeper configuration and a particular square x, is it safe to click on x?

Tetris: Given a sequence of N Tetris pieces and a partially filled $n \times k$ board, is it possible to play every piece in the sequence without overflowing the board?

Sudoku: Given an $n \times n$ Sudoku puzzle, does it have a solution?

KenKen: Given an $n \times n$ Ken-Ken puzzle, does it have a solution?

CS 573: Graduate Algorithms, Fall 2010
Homework 0

Due Wednesday, September 1, 2010 **in class**

- This homework tests your familiarity with prerequisite material (http://www.cs.uiuc.edu/class/fa10/cs573/stuff-you-already-know.html) to help you identify gaps in your background knowledge. **You are responsible for filling those gaps.** Fr most topics, the early chapters of any algorithms textbook should be sufficient review, but you may also want consult your favorite discrete mathematics and data structures textbooks. If you need help, please ask in office hours and/or on the course newsgroup.

- Each student must submit individual solutions for these homework problems. For all future homeworks, groups of up to three students may submit (or present) a single group solution for each problem.

- Please carefully read the course policies linked from the course web site. If you have *any* questions, please ask during lecture or office hours, or post your question to the course newsgroup. In particular:
  - Submit five separately stapled solutions, one for each numbered problem, with your name and NetID clearly printed on each page. Please do *not* staple everything together.
  - You may use any source at your disposal—paper, electronic, or human—but you ***must*** write your solutions in your own words, and you ***must*** cite every source that you use. In particular, each solution should include a list of *everyone* you worked with to solve that problem.
  - Unless explicitly stated otherwise, ***every*** homework problem requires a proof.
  - Answering “I don’t know” to any homework or exam problem (except for extra credit problems) is worth 25% partial credit.
  - Algorithms or proofs containing phrases like “and so on” or “repeat this process for all n” instead of an explicit loop, recursion, or induction, will receive 0 points.

---

1. (•) ***Write the sentence "I understand the course policies."***
   Solutions that omit this sentence will not be graded.

   (a) **[5 pts]** Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Assume reasonable but nontrivial base cases if none are given. ***Do not submit proofs***—just a list of five functions—but you should do them anyway, just for practice.

   - $A(n) = 4A(n-1) + 1$
   - $B(n) = B(n-3) + n^2$
   - $C(n) = 2\,C(n/2) + 3\,C(n/3) + n^2$
   - $D(n) = 2\,D(n/3) + \sqrt{n}$
   - $E(n) = \begin{cases} n & \text{if } n \le 3, \\ \dfrac{E(n-1)E(n-2)}{E(n-3)} & \text{otherwise} \end{cases}$ *[Hint: This is easier than it looks!]*

   (b) **[5 pts]** Sort the following functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. ***Do not submit proofs***—just a sorted list of 16 functions—but you should do them anyway, just for practice.

   Write $f(n) \ll g(n)$ to indicate that $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. We use the notation $\lg n = \log_2 n$.

n	$\lg n$	$\sqrt{n}$	7^n
$\sqrt{\lg n}$	$\lg \sqrt{n}$	$7^{\sqrt{n}}$	$\sqrt{7^n}$
$7^{\lg n}$	$\lg(7^n)$	$7^{\lg \sqrt{n}}$	$7^{\sqrt{\lg n}}$
$\sqrt{7^{\lg n}}$	$\lg(7^{\sqrt{n}})$	$\lg \sqrt{7^n}$	$\sqrt{\lg(7^n)}$

2. Professore Giorgio della Giungla has a 23-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet, which is just like the modern English alphabet, but without the letters **J**, **U**, and **W**. Inorder and postorder traversals of the tree visit the nodes in the following order:

   - Inorder: **S V Z A T P R D B X O L F E H I Q M N G Y K C**
   - Postorder: **A Z P T X B D L E F O H R I V N M K C Y G Q S**

   (a) List the nodes in Prof. della Giungla's tree in the order visited by a *preorder* traversal.

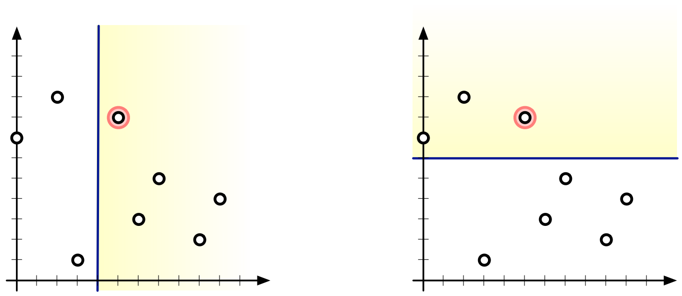
   (b) Draw Prof. della Giungla's tree.

3. The original version of this problem asked to support the mirror-image operations LowestToRight and LeftmostAbove, which are *much* harder to support with a single data structure that stores each point at most once. We will accept $O(n)$-space data structures for either version of the problem for full credit.

   Describe a data structure that stores a set S of n points in the plane, each represented by a pair (x, y) of coordinates, and supports the following queries.

   - **HighestToRight(ℓ):** Return the highest point in S whose x-coordinate is greater than or equal to ℓ. If every point in S has x-coordinate less than ℓ, return None.
   - **RightmostAbove(ℓ):** Return the rightmost point in S whose y-coordinate is greater than or equal to ℓ. If every point in S has y-coordinate less than ℓ, return None.

   For example, if $S = \{(3,1), (1,9), (9,2), (6,3), (5,8), (7,5), (10,4), (0,7)\}$, then both HighestToRight(4) and RightmostAbove(6) should return the point $(5,8)$, and HighestToRight(15) should return None.

   HighestToRight(4) = RightmostAbove(6) = (5, 8)

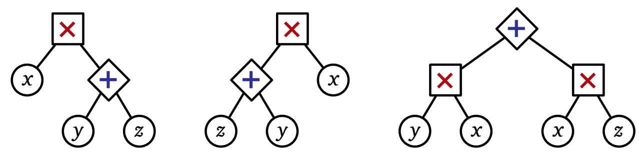
   Analyze both the size of your data structure and the running times of your query algorithms. For full credit, your data structure should use $O(n)$ space, and each query algorithm should run in $O(\log n)$ time. ***For 5 extra credit points, describe a data structure that stores each point at most once.*** You may assume that no two points in S have equal x-coordinates or equal y-coordinates.

   [Hint: Modify one of the standard data structures listed at http://www.cs.uiuc.edu/class/fa10/cs573/stuff-you-already-know.html, but just describe your changes; don't regurgitate the details of the standard data structure.]

4. An ***arithmetic expression tree*** is a binary tree where every leaf is labeled with a variable, every internal node is labeled with an arithmetic operation, and every internal node has exactly two children. For this problem, assume that the only allowed operations are $+$ and $\times$. Different leaves may or may not represent different variables.

   Every arithmetic expression tree represents a function, transforming input values for the leaf variables into an output value for the root, by following two simple rules: (1) The value of any $+$-node is the sum of the values of its children. (2) The value of any $\times$-node is the product of the values of its children.

   Two arithmetic expression trees are ***equivalent*** if they represent the same function; that is, the same input values for the leaf variables always leads to the same output value at both roots.



Three equivalent expression trees. Only the third tree is in normal form.

An arithmetic expression tree is in ***normal form*** if the parent of every $+$-node (if any) is another $+$-node.

Prove that for any arithmetic expression tree, there is an equivalent arithmetic expression tree in normal form. *[Hint: Be careful. This is trickier than it looks.]*

5. Recall that a standard (Anglo-American) deck of 52 playing cards contains 13 cards in each of four suits: spades (♠), hearts (♥), diamonds (♦), and clubs (♣). Within each suit, the 13 cards have distinct *ranks*: 2, 3, 4, 5, 6, 7, 8, 9, 10, jack (J), queen (Q), king (K), and ace (A). The ranks are ordered $2 < 3 < \cdots < 9 < 10 < J < Q < K < A$; thus, for example, the jack of spades has higher rank thank the eight of diamonds.

   Professor Jay is about to perform a public demonstration with two decks of cards, one with red backs ('the red deck') and one with blue backs ('the blue deck'). Both decks lie face-down on a table in front of Professor Jay, shuffled uniformly and independently. Thus, in each deck, every permutation of the 52 cards is equally likely.

   To begin the demonstration, Professor Jay turns over the top card from each deck. Then, while he has not yet turned over a three of clubs (3♣), the good Professor hurls the two cards he just turned over into the thick, pachydermatous outer melon layer of a nearby watermelon (that most prodigious of household fruits) and then turns over the next card from the top of each deck. Thus, if 3♣ is the last card in both decks, the demonstration ends with 102 cards embedded in the watermelon.

   (a) What is the *exact* expected number of cards that Professor Jay hurls into the watermelon?

   (b) For each of the statements below, give the *exact* probability that the statement is true of the ***first*** pair of cards Professor Jay turns over.

      i. Both cards are threes.
      ii. One card is a three, and the other card is a club.
      iii. If (at least) one card is a heart, then (at least) one card is a diamond.
      iv. The card from the red deck has higher rank than the card from the blue deck.

   (c) For each of the statements below, give the *exact* probability that the statement is true of the ***last*** pair of cards Professor Jay turns over.

      i. Both cards are threes.
      ii. One card is a three, and the other card is a club.
      iii. If (at least) one card is a heart, then (at least) one card is a diamond.
      iv. The card from the red deck has higher rank than the card from the blue deck.

   Express each of your answers as rational numbers in simplest form, like 123/4567. ***Do not submit proofs***—just a list of rational numbers—but you should do them anyway, just for practice.

CS 573: Graduate Algorithms, Fall 2010
Homework 1

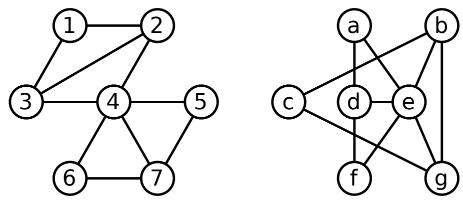
~~Due Friday, September 10, 2010 at 1pm~~

Due Monday, September 13, 2010 at 5pm
(in the homework drop boxes in the basement of Siebel)

For this and all future homeworks, groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name and NetID on each page of your submission.

---

1. Two graphs are said to be ***isomorphic*** if one can be transformed into the other just by relabeling the vertices. For example, the graphs shown below are isomorphic; the left graph can be transformed into the right graph by the relabeling $(1,2,3,4,5,6,7) \mapsto (c,g,b,e,a,f,d)$.



Two isomorphic graphs.

Consider the following related decision problems:

- GRAPHISOMORPHISM: Given two graphs G and H, determine whether G and H are isomorphic.
- EVENGRAPHISOMORPHISM: Given two graphs G and H, such that every vertex in G and H has even degree, determine whether G and H are isomorphic.
- SUBGRAPHISOMORPHISM: Given two graphs G and H, determine whether G is isomorphic to a subgraph of H.

(a) Describe a polynomial-time reduction from EVENGRAPHISOMORPHISM to GRAPHISOMORPHISM.

(b) Describe a polynomial-time reduction from GRAPHISOMORPHISM to EVENGRAPHISOMORPHISM.

(c) Describe a polynomial-time reduction from GRAPHISOMORPHISM to SUBGRAPHISOMORPHISM.

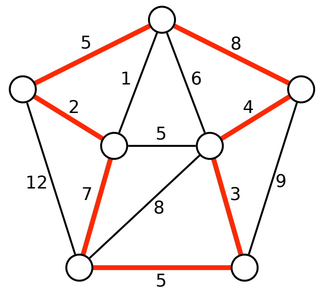
(d) Prove that SUBGRAPHISOMORPHISM is NP-complete.

(e) What can you conclude about the NP-hardness of GRAPHISOMORPHISM? Justify your answer.

[Hint: These are all easy!]

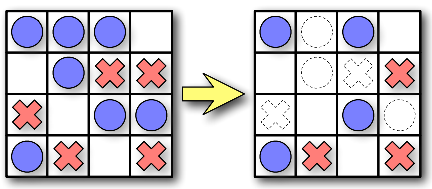
2. Suppose you are given a magic black box that can solve the 3COLORABLE problem ***in polynomial time***. That is, given an arbitrary graph G as input, the magic black box returns TRUE if G has a proper 3-coloring, and returns FALSE otherwise. Describe and analyze a ***polynomial-time*** algorithm that computes an actual proper 3-coloring of a given graph G, or correctly reports that no such coloring exists, using this magic black box as a subroutine. *[Hint: The input to the black box is a graph. Just a graph. Nothing else.]*

3. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-complete.

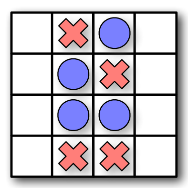


A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

4. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

5. A boolean formula in ***exclusive-or conjunctive normal form*** (XCNF) is a conjunction (AND) of several *clauses*, each of which is the *exclusive*-or of one or more literals. For example:

$$(u \oplus v \oplus \bar{w} \oplus x) \wedge (\bar{u} \oplus \bar{w} \oplus y) \wedge (\bar{v} \oplus y) \wedge (\bar{u} \oplus \bar{v} \oplus x \oplus y) \wedge (w \oplus x) \wedge y$$

The XCNF-SAT problem asks whether a given XCNF boolean formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-complete.

CS 573: Graduate Algorithms, Fall 2010
Homework 2

Due Monday, September 27, 2010 at 5pm
(in the homework drop boxes in the basement of Siebel)

- For this and all future homeworks, groups of up to three students may submit a single, common solution. Please neatly print (or typeset) the full name and NetID of every group member on the first page of your submission.
- We will use the following rubric to grade all dynamic programming algorithms:
  - 60% for a correct recurrence (including base cases and a plain-English specification); no credit for anything else if this is wrong.
  - 10% for describing a suitable memoization data structure.
  - 20% for describing a correct evaluation order. (A clear picture is sufficient.)
  - 10% point for analyzing the running time of the resulting algorithm.

  Official solutions will always include pseudocode for the final dynamic programming algorithm, but this is ***not*** required for full credit. However, if you do provide correct pseudocode for the dynamic programming algorithm, it is not necessary to separately describe the recurrence, the memoization data structure, or the evaluation order.

  It is ***not*** necessary to state a space bound. There is no penalty for using more space than the official solution, but +1 extra credit for using less space with the same (or better) running time.
- The official solution for every problem will provide a target time bound. Algorithms faster than the official solution are worth more points (as extra credit); algorithms slower than the official solution are worth fewer points. For slower algorithms, partial credit is scaled to the lower maximum score. For example, if a full dynamic programming algorithm would be worth 5 points, just the recurrence is worth 3 points. However, incorrect algorithms are worth zero points, no matter how fast they are.
- Greedy algorithms *must* be accompanied by proofs of correctness in order to receive *any* credit. Otherwise, **any correct algorithm, no matter how slow, is worth at least 2½ points**, assuming it is properly analyzed.

---

1. Suppose you are given an array $A[1..n]$ of positive integers. Describe and analyze an algorithm to find the *smallest* positive integer that is *not* an element of A in $O(n)$ time.

2. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..m, 1..n]$ whose entries are all 0 or 1. A *solid block* is a subarray of the form $M[i..i', j..j']$ in which every bit is equal to 1. Describe and analyze an efficient algorithm to find a solid block in M with maximum area.

3. Let T be a tree in which each edge e has a weight $w(e)$. A *matching* M in T is a subset of the edges such that each vertex of T is incident to at most one edge in M. The weight of a matching M is the sum of the weights of its edges. Describe and analyze an algorithm to compute a maximum weight matching, given the tree T as input.

4. For any string x and any non-negative integer k, let x^k denote the string obtained by concatenating k copies of x. For example, $\mathtt{STRING}^3 = \mathtt{STRINGSTRINGSTRING}$ and $\mathtt{STRING}^0$ is the empty string.

   A ***repetition*** of x is a prefix of x^k for some integer k. For example, STRINGSTRINGSTRINGST and STR are both repetitions of STRING, as is the empty string.

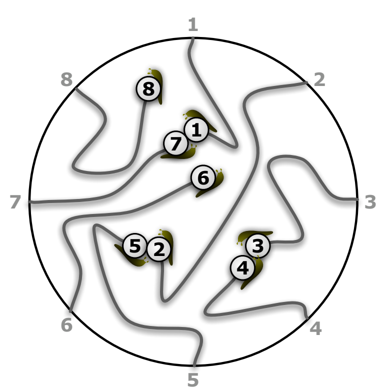
   An ***interleaving*** of two strings x and y is any string obtained by shuffling a repetition of x with a repetition of y. For example, STRWORINDGSTWORIRNGDWSTORR is an interleaving of STRING and WORD, as is the empty string.

   Describe and analyze an algorithm that accepts three strings x, y, and z as input, and decides whether z is an interleaving of x and y.

5. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.

   For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i, j] = M[j, i]$ is the reward to be paid if snails i and j meet.

   Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.



The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates.
The organizers must pay $M[3,4] + M[2,5] + M[1,7]$.

CS 573: Graduate Algorithms, Fall 2010
Homework 3

Due Monday, October 18, 2010 at 5pm
(in the homework drop boxes in the basement of Siebel)

---

1. Suppose we are given two arrays $C[1..n]$ and $R[1..n]$ of positive integers. An $n \times n$ matrix of 0s and 1s *agrees with R and C* if, for every index i, the ith row contains $R[i]$ 1s, and the ith column contains $C[i]$ 1s. Describe and analyze an algorithm that either constructs a matrix that agrees with R and C, or correctly reports that no such matrix exists.

2. Suppose we have n skiers with heights given in an array $P[1..n]$, and n skis with heights given in an array $S[1..n]$. Describe an efficient algorithm to assign a ski to each skier, so that the average difference between the height of a skier and her assigned ski is as small as possible. The algorithm should compute a permutation σ such that the expression
$$\frac{1}{n}\sum_{i=1}^{n}\Big|P[i]-S[\sigma(i)]\Big|$$
is as small as possible.

3. Alice wants to throw a party and she is trying to decide who to invite. She has n people to choose from, and she knows which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints:
   - For each guest, there should be at least five other guests that they already know.
   - For each guest, there should be at least five other guests that they *don't* already know.

   Describe and analyze an algorithm that computes the largest possible number of guests Alice can invite, given a list of n people and the list of pairs who know each other.

4. Consider the following heuristic for constructing a vertex cover of a connected graph G: return the set of non-leaf nodes in any depth-first spanning tree of G.
   (a) Prove that this heuristic returns a vertex cover of G.
   (b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G.
   (c) Describe an infinite family of graphs for which this heuristic returns a vertex cover of size $2 \cdot OPT$.

5. Suppose we want to route a set of N calls on a telecommunications network that consist of a cycle on n nodes, indexed in order from 0 to $n-1$. Each call has a source node and a destination node, and can be routed either clockwise or counterclockwise around the cycle. Our goal is to route the calls so as to minimize the overall load on the network. The load L_i on any edge $(i,(i+1) \bmod n)$ is the number of calls routed through that edge, and the overall load is $\max_i L_i$. Describe and analyze an efficient 2-approximation algorithm for this problem.

CS 573: Graduate Algorithms, Fall 2010
Homework 4

Due Monday, November 1, 2010 at 5pm
(in the homework drop boxes in the basement of Siebel)

---

1. Consider an n-node treap T. As in the lecture notes, we identify nodes in T by the ranks of their search keys. Thus, 'node 5' means the node with the 5th smallest search key. Let i, j, k be integers such that $1 \leq i \leq j \leq k \leq n$.

   (a) What is the *exact* probability that node j is a common ancestor of node i and node k?

   (b) What is the *exact* expected length of the unique path from node i to node k in T?

2. Let $M[1..n, 1..n]$ be an $n \times n$ matrix in which every row and every column is sorted. Such an array is called *totally monotone*. No two elements of M are equal.

   (a) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, compute the number of elements of M smaller than $M[i, j]$ and larger than $M[i', j']$.

   (b) Describe and analyze an algorithm to solve the following problem in $O(n)$ time: Given indices i, j, i', j' as input, return an element of M chosen uniformly at random from the elements smaller than $M[i, j]$ and larger than $M[i', j']$. Assume the requested range is always non-empty.

   (c) Describe and analyze a randomized algorithm to compute the median element of M in $O(n \log n)$ expected time.

3. Suppose we are given a complete undirected graph G, in which each edge is assigned a weight chosen independently and uniformly at random from the real interval $[0, 1]$. Consider the following greedy algorithm to construct a Hamiltonian cycle in G. We start at an arbitrary vertex. While there is at least one unvisited vertex, we traverse the minimum-weight edge from the current vertex to an unvisited neighbor. After $n - 1$ iterations, we have traversed a Hamiltonian path; to complete the Hamiltonian cycle, we traverse the edge from the last vertex back to the first vertex. What is the expected weight of the resulting Hamiltonian cycle? *[Hint: What is the expected weight of the first edge? Consider the case $n = 3$.]*

4. (a) Consider the following deterministic algorithm to construct a vertex cover C of a graph G.

```
VertexCover(G):
  C ← ∅
  while C is not a vertex cover
      pick an arbitrary edge uv that is not covered by C
      add either u or v to C
  return C
```

Prove that VertexCover can return a vertex cover that is $\Omega(n)$ times larger than the smallest vertex cover. You need to describe both an input graph with n vertices, for any integer n, and the sequence of edges and endpoints chosen by the algorithm.

(b) Now consider the following randomized variant of the previous algorithm.

```
RandomVertexCover(G):
  C ← ∅
  while C is not a vertex cover
      pick an arbitrary edge uv that is not covered by C
      with probability 1/2
          add u to C
      else
          add v to C
  return C
```

Prove that the expected size of the vertex cover returned by RandomVertexCover is at most $2 \cdot \text{OPT}$, where OPT is the size of the smallest vertex cover.

(c) Let G be a graph in which each vertex v has a weight $w(v)$. Now consider the following randomized algorithm that constructs a vertex cover.

```
RandomWeightedVertexCover(G):
  C ← ∅
  while C is not a vertex cover
      pick an arbitrary edge uv that is not covered by C
      with probability w(v)/(w(u) + w(v))
          add u to C
      else
          add v to C
  return C
```

Prove that the expected weight of the vertex cover returned by RandomWeightedVertexCover is at most $2 \cdot \text{OPT}$, where OPT is the weight of the minimum-weight vertex cover. A correct answer to this part automatically earns full credit for part (b).

5. (a) Suppose n balls are thrown uniformly and independently at random into m bins. For any integer k, what is the *exact* expected number of bins that contain exactly k balls?

   (b) Consider the following balls and bins experiment, where we repeatedly throw a fixed number of balls randomly into a shrinking set of bins. The experiment starts with n balls and n bins. In each round i, we throw n balls into the remaining bins, and then discard any non-empty bins; thus, only bins that are empty at the end of round i survive to round $i+1$.

```
BALLSDESTROYBINS(n):
  start with n empty bins
  while any bins remain
      throw n balls randomly into the remaining bins
      discard all bins that contain at least one ball
```

   Suppose that in every round, *precisely* the expected number of bins are empty. Prove that under these conditions, the experiment ends after $O(\log^* n)$ rounds.[1]

   ⋆(c) **[Extra credit]** Now assume that the balls are really thrown randomly into the bins in each round. Prove that with high probability, BALLSDESTROYBINS(n) ends after $O(\log^* n)$ rounds.

   (d) Now consider a variant of the previous experiment in which we discard balls instead of bins. Again, the experiment n balls and n bins. In each round i, we throw the remaining balls into n bins, and then discard any ball that lies in a bin by itself; thus, only balls that collide in round i survive to round $i+1$.

```
BINSDESTROYSINGLEBALLS(n):
  start with n balls
  while any balls remain
      throw the remaining balls randomly into n bins
      discard every ball that lies in a bin by itself
      retrieve the remaining balls from the bins
```

   Suppose that in every round, *precisely* the expected number of bins contain exactly one ball. Prove that under these conditions, the experiment ends after $O(\log\log n)$ rounds.

   ⋆(e) **[Extra credit]** Now assume that the balls are really thrown randomly into the bins in each round. Prove that with high probability, BINSDESTROYSINGLEBALLS(n) ends after $O(\log\log n)$ rounds.

[1] Recall that the iterated logarithm is defined as follows: $\log^* n = 0$ if $n \le 1$, and $\log^* n = 1 + \log^*(\lg n)$ otherwise.

CS 573: Graduate Algorithms, Fall 2010
Homework 5

Due Friday, November 19, 2010 at 5pm
(in the homework drop boxes in the basement of Siebel)

---

1. Suppose we are given a set of boxes, each specified by their height, width, and depth in centimeters. All three side lengths of every box lie strictly between 10cm and 20cm. As you should expect, one box can be placed inside another if the smaller box can be rotated so that its height, width, and depth are respectively smaller than the height, width, and depth of the larger box. Boxes can be nested recursively. Call a box is *visible* if it is not inside another box.

   Describe and analyze an algorithm to nest the boxes so that the number of visible boxes is as small as possible.

2. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round A* to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

   Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

3. The Autocratic Party is gearing up their fund-raising campaign for the 2012 election. Party leaders have already chosen their slate of candidates for president and vice-president, as well as various governors, senators, representatives, city council members, school board members, and dog-catchers. For each candidate, the party leaders have determined how much money they must spend on that candidate's campaign to guarantee their election.

   The party is soliciting donations from each of its members. Each voter has declared the total amount of money they are willing to give each candidate between now and the election. (Each voter pledges different amounts to different candidates. For example, everyone is happy to donate to the presidential candidate,[1] but most voters in New York will not donate anything to the candidate for Trash Commissioner of Los Angeles.) Federal election law limits each person's total political contributions to \$100 per day.

   Describe and analyze an algorithm to compute a donation schedule, describing how much money each voter should send to each candidate on each day, that guarantees that every candidate gets enough money to win their election. (Party members will of course follow their given schedule perfectly.[2]) The schedule must obey both Federal laws and individual voters' budget constraints. If no such schedule exists, your algorithm should report that fact.

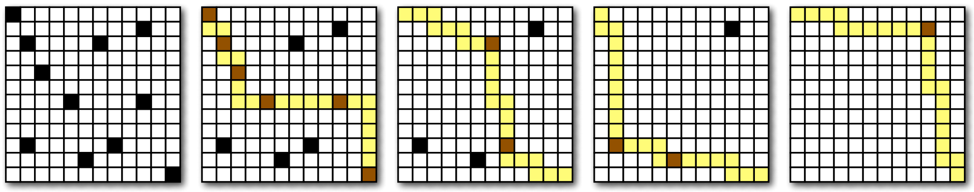
[1] or some nice men in suits will be visiting their home.
[2] It's a nice house you've got here. Shame if anything happened to it.

4. Consider an $n \times n$ grid, some of whose cells are marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. We want to compute the minimum number of monotone paths that cover all the marked cells.

   (a) One of your friends suggests the following greedy strategy:

   - Find (somehow) one "good" path π that covers the maximum number of marked cells.
   - Unmark the cells covered by π.
   - If any cells are still marked, recursively cover them.

   Prove that this greedy strategy does *not* always compute an optimal solution.



Greedily covering the marked cells in a grid with four monotone paths.

   (b) Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell. The input to your algorithm is an array $M[1..n, 1..n]$ of booleans, where $M[i, j] = \text{True}$ if and only if cell (i, j) is marked.

5. Let G be a directed graph with two distinguished vertices s and t, and let r be a positive integer. Two players named Paul and Sally play the following game. Paul chooses a path P from s to t, and Sally chooses a subset S of ***at most*** r edges in G. The players reveal their chosen subgraphs simultaneously. If $P \cap S = \varnothing$, Paul wins; if $P \cap S \neq \varnothing$, then Sally wins. Both players want to maximize their chances of winning the game.

   (a) Prove that if Paul uses a deterministic strategy, and Sally knows his strategy, then Sally can guarantee that she wins.[3]

   (b) Let M be the number of edges in a minimum (s, t)-cut. Describe a deterministic strategy for Sally that guarantees that she wins when $\boldsymbol{r \geq M}$, no matter what strategy Paul uses.

   (c) Prove that if Sally uses a deterministic strategy, and Paul knows her strategy then Paul can guarantee that he wins when $r < M$.

   (d) Describe a randomized strategy for Sally that guarantees that she wins with probability at least $\min\{r/M, 1\}$, no matter what strategy Paul uses.

   (e) Describe a randomized strategy for Paul that guarantees that he loses with probability at most $\min\{r/M, 1\}$, no matter what strategy Sally uses.

   Paul and Sally's strategies are, of course, algorithms. (For example, Paul's strategy is an algorithm that takes the graph G and the integer r as input and produces a path P as output.) You do *not* need to analyze the running times of these algorithms, but you must prove all claims about their winning probabilities. Most of these questions are easy.

[3] "Good old rock. Nothing beats rock. ...D'oh!"

CS 573: Graduate Algorithms, Fall 2010
Homework 5

Practice only — Do not submit solutions

---

1. (a) Describe how to transform any linear program written in general form into an equivalent linear program written in ***slack*** form.

$$\boxed{\begin{array}{lll} \text{maximize} & \displaystyle\sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \displaystyle\sum_{j=1}^{d} a_{ij} x_j \le b_i & \text{for each } i = 1 \,..\, p \\ & \displaystyle\sum_{j=1}^{d} a_{ij} x_j = b_i & \text{for each } i = p+1 \,..\, p+q \\ & \displaystyle\sum_{j=1}^{d} a_{ij} x_j \ge b_i & \text{for each } i = p+q+1 \,..\, n \end{array}} \implies \boxed{\begin{array}{rl} \max & c \cdot x \\ \text{s.t.} & Ax = b \\ & x \ge 0 \end{array}}$$

   (b) Describe precisely how to dualize a linear program written in slack form.

   (c) Describe precisely how to dualize a linear program written in general form.

   In all cases, keep the number of variables in the resulting linear program as small as possible.

2. Suppose you have a subroutine that can solve linear programs in polynomial time, but only if they are both feasible and bounded. Describe an algorithm that solves *arbitrary* linear programs in polynomial time. Your algorithm should return an optimal solution if one exists; if no optimum exists, your algorithm should report that the input instance is UNBOUNDED or INFEASIBLE, whichever is appropriate. *[Hint: Add one variable and one constraint.]*

3. An *integer program* is a linear program with the additional constraint that the variables must take only integer values.

   (a) Prove that deciding whether an integer program has a feasible solution is NP-complete.

   (b) Prove that finding the optimal feasible solution to an integer program is NP-hard.

   [Hint: Almost any NP-hard decision problem can be formulated as an integer program. Pick your favorite.]

4. Give a linear-programming formulation of the *minimum-cost feasible circulation problem*. You are given a flow network whose edges have both capacities and costs, and your goal is to find a feasible circulation (flow with value 0) whose cost is as small as possible.

5. Given points $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ in the plane, the *linear regression problem* asks for real numbers a and b such that the line $y = ax + b$ fits the points as closely as possible, according to some criterion. The most common fit criterion is minimizing the L_2 *error*, defined as follows:[1]

$$\varepsilon_2(a, b) = \sum_{i=1}^{n} (y_i - ax_i - b)^2.$$

But there are several other fit criteria, some of which can be optimized via linear programming.

(a) The L_1 *error* (or *total absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_1(a, b) = \sum_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_1 error.

(b) The L_∞ *error* (or *maximum absolute deviation*) of the line $y = ax + b$ is defined as follows:

$$\varepsilon_\infty(a, b) = \max_{i=1}^{n} \left| y_i - ax_i - b \right|.$$

Describe a linear program whose solution (a, b) describes the line with minimum L_∞ error.

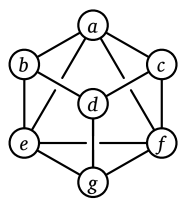
[1]This measure is also known as *sum of squared residuals*, and the algorithm to compute the best fit is normally called *(ordinary/linear) least squares fitting*.

This exam lasts 90 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet with your answers.

1. (a) Suppose $A[1..n]$ is an array of n distinct integers, sorted so that $A[1] < A[2] < \cdots < A[n]$. Each integer $A[i]$ could be positive, negative, or zero. Describe and analyze an efficient algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

   (b) Now suppose $A[1..n]$ is a sorted array of n distinct ***positive*** integers. Describe and analyze an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

2. A *double-Hamiltonian circuit* a closed walk in a graph that visits every vertex exactly *twice*. ***Prove*** that it is NP-hard to determine whether a given graph contains a double-Hamiltonian circuit.



This graph contains the double-Hamiltonian circuit $a \to b \to d \to g \to e \to b \to d \to c \to f \to a \to c \to f \to g \to e \to a$.

3. A palindrome is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `HANNAH`, or `AMANAPLANACATACANALPANAMA`. Describe and analyze an algorithm to find the length of the longest subsequence of a given string that is also a palindrome.

   For example, the longest palindrome subsequence of `MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM` is `MHYMRORMYHM`, so given that string as input, your algorithm should return the integer 11.

4. Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, the number of vertices in the largest complete subgraph of G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

5. Suppose we are given a $4 \times n$ grid, where each grid cell has an integer value. Suppose we want to mark a subset of the grid cells, so that the total value of the marked cells is as large as possible. However, we are forbidden to mark any pair of grid cells that are immediate horizontal or vertical neighbors. (Marking diagonal neighbors is fine.) Describe and analyze an algorithm that computes the largest possible sum of marked cells, subject to this non-adjacency condition.

   For example, given the grid on the left below, your algorithm should return the integer 36, which is the sum of the circled numbers on the right.

4	−5	1	6
2	6	−1	8
5	4	3	3
1	−1	7	4
−3	4	5	−2

$\Longrightarrow$

4	−5	1	6
2	6	−1	8
5	4	3	3
1	−1	7	4
−3	5	4	−2

You may assume the following problems are NP-hard:

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

PlanarCircuitSat: Given a boolean circuit drawn in the plane so that no two wires cross, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

Max2Sat: Given a boolean formula in conjunctive normal form, with exactly two literals per clause, what is the largest number of clauses that can be satisfied by an assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinDominatingSet: Given an undirected graph G, what is the size of the smallest subset S of vertices such that every vertex in G is either in S or adjacent to a vertex in S?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

ChromaticNumber: Given an undirected graph G, what is the minimum number of colors needed to color its vertices, so that every edge touches vertices with two different colors?

MaxCut: Given a graph G, what is the size (number of edges) of the largest bipartite subgraph of G?

HamiltonianCycle: Given a graph G, is there a cycle in G that visits every vertex exactly once?

HamiltonianPath: Given a graph G, is there a path in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of n positive integers, can X be partitioned into $n/3$ three-element subsets, all with the same sum?

Minesweeper: Given a Minesweeper configuration and a particular square x, is it safe to click on x?

Tetris: Given a sequence of N Tetris pieces and a partially filled $n \times k$ board, is it possible to play every piece in the sequence without overflowing the board?

Sudoku: Given an $n \times n$ Sudoku puzzle, does it have a solution?

KenKen: Given an $n \times n$ Ken-Ken puzzle, does it have a solution?

This exam lasts 90 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet with your answers.

1. Assume we have access to a function RANDOM(k) that returns, given any positive integer k, an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, k\}$, in $O(1)$ time. For example, to perform a fair coin flip, we could call RANDOM(2).

   Now suppose we want to write an efficient function RANDOMPERMUTATION(n) that returns a permutation of the set $\{1, 2, \ldots, n\}$ chosen uniformly at random; that is, each permutation must be chosen with probability $1/n!$.

   (a) ***Prove*** that the following algorithm is ***not*** correct. *[Hint: Consider the case $n = 3$.]*

   ```
   RandomPermutation(n):
     for i ← 1 to n
         π[i] ← i
     for i ← 1 to n
         swap π[i] ↔ π[Random(n)]
     return π
   ```

   (b) Describe and analyze a correct RANDOMPERMUTATION algorithm that runs in $O(n)$ expected time. (In fact, $O(n)$ worst-case time is possible.)

2. Suppose we have n pieces of candy with weights $W[1..n]$ (in ounces) that we want to load into boxes. Our goal is to load the candy into as many boxes as possible, so that each box contains at least L ounces of candy. Describe an efficient 2-approximation algorithm for this problem. ***Prove*** that the approximation ratio of your algorithm is 2.

   (For 7 points partial credit, assume that every piece of candy weighs less than L ounces.)

3. The MAXIMUM-k-CUT problem is defined as follows. We are given a graph G with weighted edges and an integer k. Our goal is to partition the vertices of G into k subsets $S_1, S_2, \ldots, S_k$, so that the sum of the weights of the edges that cross the partition (that is, with endpoints in different subsets) is as large as possible.

   (a) Describe an efficient randomized approximation algorithm for MAXIMUM-k-CUT, and ***prove*** that its expected approximation ratio is ~~at most~~ $(k-1)/k$.

   (b) Now suppose we want to minimize the sum of the weights of edges that do *not* cross the partition. What expected approximation ratio does your algorithm from part (a) achieve for this new problem? ***Prove*** your answer is correct.

4. The citizens of Binaria use coins whose values are powers of two. That is, for any non-negative integer k, there are Binarian coins with value is 2^k bits. Consider the natural greedy algorithm to make x bits in change: If $x > 0$, use one coin with the largest denomination $d \le x$ and then recursively make $x - d$ bits in change. (Assume you have an unlimited supply of each denomination.)

   (a) ***Prove*** that this algorithm uses at most one coin of each denomination.

   (b) ***Prove*** that this algorithm finds the minimum number of coins whose total value is x.

5. Any permutation π can be represented as a set of disjoint cycles, by considering the directed graph whose vertices are the integers between 1 and n and whose edges are $i \rightarrow \pi(i)$ for each i. For example, the permutation $\langle 5, 4, 2, 6, 7, 8, 1, 3, 9 \rangle$ has three cycles: $(175)(24683)(9)$.

   In the following questions, let π be a permutation of $\{1, 2, \ldots, n\}$ chosen uniformly at random, and let k be an arbitrary integer such that $1 \le k \le n$.

   (a) ***Prove*** that the probability that the number 1 lies in a cycle of length k in π is precisely $1/n$. *[Hint: Consider the cases $k = 1$ and $k = 2$.]*

   (b) What is the *exact* expected length of the cycle in π that contains the number 1?

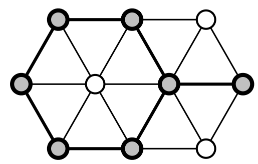
   (c) What is the *exact* expected number of cycles of length k in π?

   (d) What is the *exact* expected number of cycles in π?

   You may assume part (a) in your solutions to parts (b), (c), and (d).

This exam lasts 180 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet with your answers.

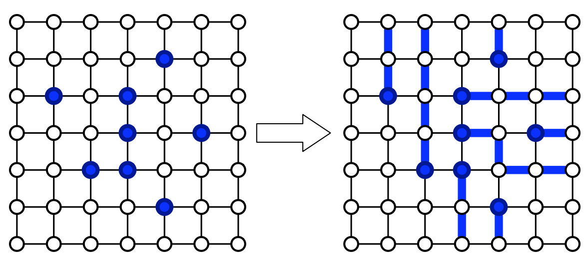
1. A subset S of vertices in an undirected graph G is called *triangle-free* if, for every triple of vertices $u, v, w \in S$, at least one of the three edges uv, uw, vw is *absent* from G. ***Prove*** that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.



A triangle-free subset of 7 vertices.
This is ***not*** the largest triangle-free subset in this graph.

2. An $n \times n$ grid is an undirected graph with n^2 vertices organized into n rows and n columns. We denote the vertex in the ith row and the jth column by (i, j). Every vertex in the grid have exactly four neighbors, except for the *boundary* vertices, which are the vertices (i, j) such that $i = 1$, $i = n$, $j = 1$, or $j = n$.

   Let $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$ be distinct vertices, called *terminals*, in the $n \times n$ grid. The ***escape problem*** is to determine whether there are m vertex-disjoint paths in the grid that connect the terminals to any m distinct boundary vertices. Describe and analyze an efficient algorithm to solve the escape problem.



A positive instance of the escape problem, and its solution.

3. Consider the following problem, called UniqueSetCover. The input is an n-element set S, together with a collection of m subsets $S_1, S_2, \ldots, S_m \subseteq S$, such that each element of S lies in exactly k subsets S_i. Our goal is to select some of the subsets so as to maximize the number of elements of S that lie in *exactly one* selected subset.

   (a) Fix a real number p between 0 and 1, and consider the following algorithm:

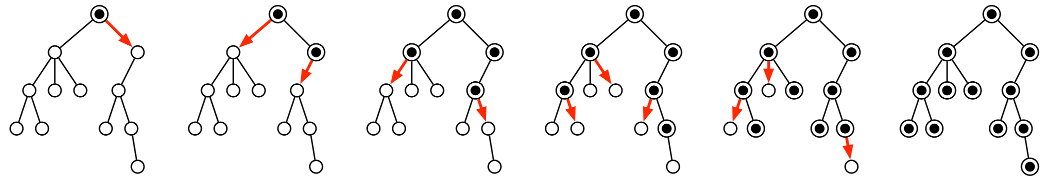
   For each index i, select subset S_i independently with probability p.

   What is the *exact* expected number of elements that are uniquely covered by the chosen subsets? (Express your answer as a function of the parameters p and k.)

   (b) What value of p maximizes this expectation?

   (c) Describe a polynomial-time randomized algorithm for UniqueSetCover whose expected approximation ratio is $O(1)$.

4. Suppose we need to distribute a message to all the nodes in a rooted tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. Describe and analyze an efficient algorithm to compute the minimum number of rounds required for the message to be delivered to every node.



A message being distributed through a tree in five rounds.

5. Every year, Professor Dumbledore assigns the instructors at Hogwarts to various faculty committees. There are n faculty members and c committees. Each committee member has submitted a list of their *prices* for serving on each committee; each price could be positive, negative, zero, or even infinite. For example, Professor Snape might declare that he would serve on the Student Recruiting Committee for 1000 Galleons, that he would *pay* 10000 Galleons to serve on the Defense Against the Dark Arts Course Revision Committee, and that he would not serve on the Muggle Relations committee for any price.

   Conversely, Dumbledore knows how many instructors are needed for each committee, as well as a list of instructors who would be suitable members for each committee. (For example: "Dark Arts Revision: 5 members, anyone but Snape.") If Dumbledore assigns an instructor to a committee, he must pay that instructor's price from the Hogwarts treasury.

   Dumbledore needs to assign instructors to committees so that (1) each committee is full, (3) no instructor is assigned to more than three committees, (2) only suitable and willing instructors are assigned to each committee, and (4) the total cost of the assignment is as small as possible. Describe and analyze an efficient algorithm that either solves Dumbledore's problem, or correctly reports that there is no valid assignment whose total cost is finite.

6. Suppose we are given a rooted tree T, where every edge e has a non-negative *length* $\ell(e)$. Describe and analyze an efficient algorithm to assign a *stretched* length $s\ell(e) \geq \ell(e)$ to every edge e, satisfying the following conditions:

   - Every root-to-leaf path in T has the same total stretched length.
   - The total stretch $\sum_e (s\ell(e) - \ell(e))$ is as small as possible.

7. Let $G = (V, E)$ be a directed graph with edge capacities $c\colon E \to \mathbb{R}^+$, a source vertex s, and a target vertex t. Suppose someone hands you an *arbitrary* function $f : E \to \mathbb{R}$. Describe and analyze fast and *simple* algorithms to answer the following questions:

   (a) Is f a feasible (s,t)-flow in G?

   (b) Is f a *maximum* (s,t)-flow in G?

   (c) Is f the ***unique*** maximum (s,t)-flow in G?

Chernoff bounds:

If X is the sum of independent indicator variables and $\mu = \mathrm{E}[X]$, then

$$\Pr[X > (1+\delta)\mu] \le \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \qquad \text{for any } \delta > 0$$

$$\Pr[X > (1-\delta)\mu] \le \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} \qquad \text{for any } 0 < \delta < 1$$

You may assume the following running times:

- Maximum flow or minimum cut: $O(E|f^*|)$ or $O(VE\log V)$
- Minimum-cost maximum flow: $O(E^2\log^2 V)$

(These are *not* the best time bounds known, but they're close enough for the final exam.)

You may assume the following problems are NP-hard:

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

PlanarCircuitSat: Given a boolean circuit drawn in the plane so that no two wires cross, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

Max2Sat: Given a boolean formula in conjunctive normal form, with exactly two literals per clause, what is the largest number of clauses that can be satisfied by an assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

MaxCut: Given a graph G, what is the size (number of edges) of the largest bipartite subgraph of G?

HamiltonianCycle: Given a graph G, is there a cycle in G that visits every vertex exactly once?

HamiltonianPath: Given a graph G, is there a path in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of n positive integers, can X be partitioned into $n/3$ three-element subsets, all with the same sum?

CS 473: Undergraduate Algorithms, Fall 2012
Homework 0

Due Tuesday, September 4, 2012 at noon

Quiz 0 (on the course Moodle page)
is also due Tuesday, September 4, 2012 at noon.

- Homework 0 and Quiz 0 test your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, graphs, and most importantly, induction—to help you identify gaps in your background knowledge. **You are responsible for filling those gaps.** The course web page has pointers to several excellent online resources for prerequisite material. If you need help, please ask in headbanging, on Piazza, in office hours, or by email.

- Each student must submit individual solutions for these homework problems. For all future homeworks, groups of up to three students may submit (or present) a single group solution for each problem.

- Please carefully read the course policies on the course web site. If you have *any* questions, please ask in lecture, in headbanging, on Piazza, in office hours, or by email. In particular:

  – Submit separately stapled solutions, one for each numbered problem, with your name and NetID clearly printed on each page, in the corresponding drop boxes outside 1404 Siebel.

  – You may use any source at your disposal—paper, electronic, human, or other—but you ***must*** write your solutions in your own words, and you ***must*** cite every source that you use (except for official course materials). Please see the academic integrity policy for more details.

  – No late homework will be accepted for any reason. However, we may *forgive* quizzes or homeworks in extenuating circumstances; ask Jeff for details.

  – Answering "I don't know" to any (non-extra-credit) problem or subproblem, on any homework or exam, is worth 25% partial credit.

  – Algorithms or proofs containing phrases like "and so on" or "repeat this process for all n", instead of an explicit loop, recursion, or induction, will receive a score of 0.

  – Unless explicitly stated otherwise, ***every*** homework problem requires a proof.

---

1. **[CS 173]** The ***Lucas numbers*** L_n are defined recursively as follows:

$$L_n = \begin{cases} 2 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ L_{n-2} + L_{n-1} & \text{otherwise} \end{cases}$$

You may recognize this as the Fibonacci recurrence, but with a different base case ($L_0 = 2$ instead of $F_0 = 0$). Similarly, the ***anti-Lucas numbers*** Γ_n are defined recursively as follows:

$$\Gamma_n = \begin{cases} 1 & \text{if } n = 0 \\ 2 & \text{if } n = 1 \\ \Gamma_{n-2} - \Gamma_{n-1} & \text{otherwise} \end{cases}$$

Here are the first several Lucas and anti-Lucas numbers:

n	0	1	2	3	4	5	6	7	8	9	10	11	12
L_n	2	1	3	4	7	11	18	29	47	76	123	199	322
Γ_n	1	2	−1	3	−4	7	−11	18	−29	47	−76	123	−199

(a) Prove that $\Gamma_n = (-1)^{n-1} L_{n-1}$ for every positive integer n.

(b) Prove that any non-negative integer can be written as the sum of distinct *non-consecutive* Lucas numbers; that is, if L_i appears in the sum, then L_{i-1} and L_{i+1} cannot. For example:

$$\begin{aligned} 4 &= 4 &&= L_3 \\ 8 &= 7+1 &&= L_4 + L_1 \\ 15 &= 11+4 &&= L_5 + L_3 \\ 16 &= 11+4+1 &&= L_5 + L_3 + L_1 \\ 23 &= 18+4+1 &&= L_6 + L_3 + L_1 \\ 42 &= 29+11+2 &&= L_7 + L_5 + L_0 \end{aligned}$$

2. **[CS 173 + CS 373]** Consider the language over the alphabet $\{♠, ♥, ♦, ♣\}$ generated by the following context-free grammar:

$$S \to ♥ \mid ♠S \mid S♣ \mid S♦S$$

Prove that every string in this language has the following properties:

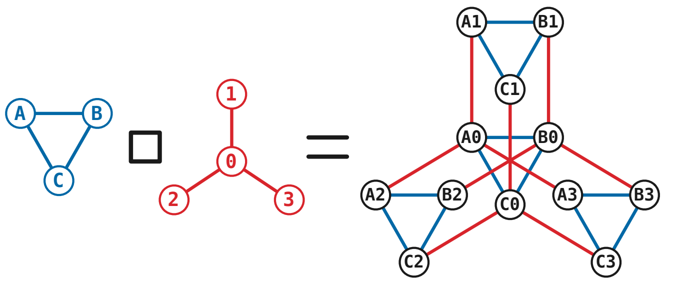
(a) The number of ♥s is exactly one more than the number of ♦s.

(b) There is a ♦ between any two ♥s.

3. **[CS 173 + mathematical maturity]** Given two undirected graphs $G = (V, E)$ and $G' = (V', E')$, we define a new graph $\boldsymbol{G \,\square\, G'}$, called the ***box product*** of G and G', as follows:

   - The vertices of $G \,\square\, G'$ are all pairs (v, v') where $v \in V$ and $v' \in V'$.
   - Two vertices (v, v') and (w, w') are connected by an edge in $G \,\square\, G'$ if and only if either $\big(v = w$ and $(v', w') \in E'\big)$ or $\big((v, w) \in E$ and $v' = w'\big)$.

   Intuitively, every pair of edges $e \in E$ and $e' \in E'$ define a "box" of four edges in $G \,\square\, G'$. For example, if G is a path of length n, then $G \,\square\, G$ is an $n \times n$ grid. Another example is shown below.



The box product of two graphs.

   (a) Let I denote the unique connected graph with two vertices. Give a concise *English* description of the following graphs. You do ***not*** have to prove that your answers are correct.
      i. What is $I \,\square\, I$?
      ii. What is $I \,\square\, I \,\square\, I$?
      iii. What is $I \,\square\, I \,\square\, I \,\square\, I$?

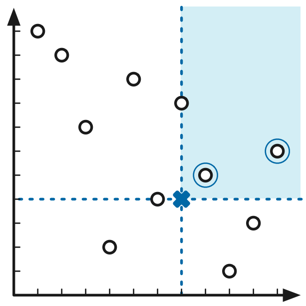
   (b) Recall that a ***Hamiltonian path*** in a graph G is a path in G that visits every vertex of G exactly once. Prove that for any graphs G and G' that both contain Hamiltonian paths, the box product $G \,\square\, G'$ also contains a Hamiltonian path. *[Hint: **Don't** use induction.]*

4. **[CS 225]** Describe and analyze a data structure that stores a set S of n points in the plane, each represented by a pair of integer coordinates, and supports queries of the following form:

   SomethingAboveRight(x, y): Return an arbitrary point (a, b) in S such that $a > x$ and $b > y$. If there is no such point in S, return None.

   For example, if S is the 11-point set $\{(1,11), (2,10), (3,7), (4,2), (5,9), (6,4), (7,8), (8,5), (9,1), (10,3), (11,6)\}$, as illustrated on the next page, then

   - SomethingAboveRight$(0,0)$ may return any point in S;
   - SomethingAboveRight$(7,7)$ must return None;
   - SomethingAboveRight$(7,4)$ must return either $(8,5)$ or $(11,6)$.



SOMETHINGABOVERIGHT$(7,4)$ returns either $(8,5)$ or $(11,6)$.

A complete solution must (a) describe a data structure, (b) analyze the space it uses, (c) describe a query algorithm, (d) prove that it is correct, and (e) analyze its worst-case running time. You do *not* need to describe how to build your data structure from a given set of points. Smaller and simpler data structures with faster and simpler query algorithms are worth more points. You may assume all points in S have distinct x-coordinates and distinct y-coordinates.

*5. ***[Extra credit]*** A ***Gaussian integer*** is a complex number of the form $x + yi$, where x and y are integers. Prove that any Gaussian integer can be expressed as the sum of distinct powers of the complex number $\alpha = -1 + i$. For example:

$$\begin{aligned}
4 &= 16 + (-8-8i) + 8i + (-4) &&= \alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 \\
-8 &= (-8-8i) + 8i &&= \alpha^7 + \alpha^6 \\
15i &= (-16+16i) + 16 + (-2i) + (-1+i) + 1 &&= \alpha^9 + \alpha^8 + \alpha^2 + \alpha^1 + \alpha^0 \\
1+6i &= (8i) + (-2i) + 1 &&= \alpha^6 + \alpha^2 + \alpha^0 \\
2-3i &= (4-4i) + (-4) + (2+2i) + (-2i) + (-1+i) + 1 &&= \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha^1 + \alpha^0 \\
-4+2i &= (-16+16i) + 16 + (-8-8i) + (4-4i) + (-2i) &&= \alpha^9 + \alpha^8 + \alpha^7 + \alpha^5 + \alpha^2
\end{aligned}$$

The following list of values may be helpful:

$$\begin{array}{llll}
\alpha^0 = 1 & \alpha^4 = -4 & \alpha^8 = 16 & \alpha^{12} = -64 \\
\alpha^1 = -1+i & \alpha^5 = 4-4i & \alpha^9 = -16+16i & \alpha^{13} = 64-64i \\
\alpha^2 = -2i & \alpha^6 = 8i & \alpha^{10} = -32i & \alpha^{14} = 128i \\
\alpha^3 = 2+2i & \alpha^7 = -8-8i & \alpha^{11} = 32+32i & \alpha^{15} = -128-128i
\end{array}$$

[Hint: How do you write $-2-i$?]

Starting with this homework, groups of up to three students may submit a single solution for each homework problem. Every student in the group receives the same grade.

---

1. Describe and analyze an algorithm to reconstruct a binary search tree T, given the sequence of keys visited by a postorder traversal of T (as in Quiz 0 problem 3).

   Assume that all the input keys are distinct. Don't worry about detecting invalid inputs; the input sequence is guaranteed to be the postorder traversal of some binary search tree.

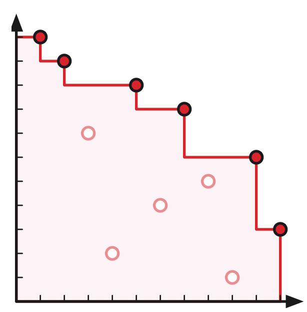
2. An array $A[0\,..\,n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3	is bitonic, but
3	6	9	8	7	5	1	2	4	is *not* bitonic.

   Describe and analyze an algorithm to find the *smallest* element in an n-element bitonic array in $O(\log n)$ time. You may assume that the numbers in the input array are distinct.

3. Let S be a set of n points in the plane. A point $p \in S$ ***maximal*** (or *Pareto-optimal*) if no point in S is both above and to the right of p. The maximal points in S intuitively define a *staircase* with all the other points in S below and to the left.

   Describe and analyze a divide-and-conquer algorithm to find all the maximal points in a given n-point set in $O(n \log n)$ time. You may assume all the input points have distinct x-coordinates and distinct y-coordinates.



Maximal points define a staircase.

*4. ***[Extra Credit]*** Describe and analyze an algorithm to find all the maximal points in a given n-point set in $\boldsymbol{O(n \log m)}$ time, where m is the number of maximal points. In particular, your algorithm should run in $O(n)$ time if the input set contains only one maximal point, and in $O(n \log n)$ time in the worst case. *[Hint: I know of at least two different ways to do this.]*

1. Describe and analyze an algorithm to determine whether a given set of positive integers can be partitioned into three disjoint subsets whose sums are equal. That is, given a set X whose elements sum to S, your algorithm should determine whether there are three disjoint subsets $A, B, C \subseteq X$ such that $A \cup B \cup C = X$ and
$$\sum_{a \in A} a = \sum_{b \in B} b = \sum_{c \in C} c = \frac{S}{3}.$$
For full credit, your algorithm should run in time polynomial in S and n.

   For example, the set $\{2,3,4,6,7,8,9,12\}$, can be partitioned into the subsets $\{2,3,12\}$, $\{4,6,7\}$, and $\{8,9\}$, each of whose elements sum to 17. On the other hand, there is no balanced partition of the set $\{4,8,15,16,23,42\}$ into three subsets.

2. Suppose Scrooge McDuck wants to split a gold chain into $n+1$ smaller chains to distribute to his numerous nephews and nieces as their inheritance. Scrooge has carefully marked n locations where he wants the chain to be cut.

   McDuck's jeweler, Fenton Crackshell, agrees to cut any chain of length ℓ inches into two smaller pieces, at any specified location, for a fee of ℓ dollars. To cut a chain into more pieces, Scrooge must pay Crackshell separately for each individual cut. As a result, the cost of breaking the chain into multiple pieces depends on the order that Crackshell makes his cuts. Obviously, Scrooge wants to pay Crackshell as little as possible.

   For example, suppose the chain is 12 inches long, and the cut locations are 3 inches, 7 inches, and 9 inches from one end of the chain. If Crackshell cuts first at the 3-inch mark, then at the 7-inch mark, and then at the 9-inch mark, Scrooge must pay $12+9+5=26$ dollars. If Crackshell cuts the chain first at the 9-inch mark, then at the 7-inch mark, and then at the 3-inch mark, Scrooge must pay $12+9+7=28$ dollars. Finally, if Crackshell makes his first cut at the 7-inch mark, Scrooge only owes him $12+7+5=24$ dollars.

   Describe and analyze an efficient algorithm that computes the minimum cost of partitioning the chain. The input to your algorithm is a sorted array $C[1\,..\,n+1]$, where $C[1]$ through $C[n]$ are the n cut positions, and $C[n+1]$ is the total length of the chain.

Rubric: For *all* dynamic programming problems in this class:

- 60% for a correct recurrence, including base cases and a plain-English description of the function. No credit for anything else if this is wrong.
- 10% for describing a suitable memoization data structure.
- 20% for describing a correct evaluation order. A clear picture is sufficient.
- 10% for analyzing the resulting algorithm's running time. It is not necessary to state a space bound.

Official solutions will always include pseudocode for the final iterative dynamic programming algorithm, but this is ***not*** required for full credit. On the other hand, if you do provide correct pseudocode for the iterative algorithm, you do not need to ***separately*** describe the recurrence, the memoization structure, or the evaluation order.

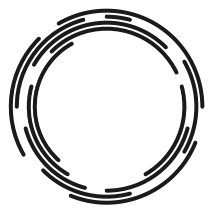
In this and all future homeworks, please clearly print the day and meeting time of *one* discussion section at the top of each page, together with the homework number, problem number, your names, and your NetIDs. We will return your graded homeworks at the discussion section you indicate. For example:

CS 473 Homework 3 Problem 2

Kyle Jao (fjao2), Nirman Kumar (nkumar5), Ernie Pan (erniepan)
Wednesday 3pm section

---

1. You are given a set A of n arcs on the unit circle, each specified by the angular coordinates of its endpoints. Describe an efficient algorithm for finding the largest possible subset X of A such that no two arcs in X intersect. Assume that all $2n$ endpoints are distinct.



2. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..m, 1..n]$ whose entries are all 0 or 1. A *checkered block* is a subarray of the form $M[i..i', j..j']$ in which no pair of adjacent entries is equal. Describe an efficient algorithm for finding the number of elements in the largest checkered block(s) in M.

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Two checkered blocks in a 6×6 bitmap

3. A company is planning a party for its employees. The employees in the company are organized in a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party have assigned a real number to each employee indicating the *awkwardness* of inviting both that employee and their immediate supervisor; a negative value indicates that the employee and their supervisor actually like each other. The organizers want to selectively invite employees to the party so that the the total awkwardness is as small as possible. For example, if the guest list does not include both an employee and their supervisor, the total awkwardness is zero.

   However, the president of the company insists on inviting exactly k employees to the party, including herself. Moreover, everyone who is invited is required to attend. Yeah, that'll be fun.

   Describe an algorithm that computes the total awkwardness of the least awkward subset of k employees to invite to the party. The input to your algorithm is an n-node rooted tree T representing the company hierarchy, an integer $awk(x)$ for each node x in T, and an integer $0 \le k \le n$. Your algorithm should return a single integer.

   For full credit, you may assume that the input is a binary tree. A complete solution for arbitrary rooted trees is worth 10 points extra credit.

4. **[Extra credit]** Two players A and B play a turn-based game on a rooted tree T. Each node v is labeled with a real number $\ell(v)$, which could be positive, negative, or zero.

   The game starts with three tokens at the root of T. In each turn, the current player moves one of the tokens from its current node down to one of its children, and the current player's score is increased by $\ell(u) \cdot \ell(v)$, where u and v are the locations of the two tokens that did *not* move. At most one token can be on any node (except the root) at any time. The game ends when the current player is unable to move, for example, if all three tokens are at leaves. The player with the higher score at the end of the game is the winner.

   Assuming that both players play optimally, describe an efficient algorithm to determine who wins on a given labeled tree. Do not assume that T is a *binary* tree.

1. Consider an n-node treap T. As in the lecture notes, we identify nodes in T by the ranks of their search keys; for example, 'node 5' means the node with the 5th smallest search key. Let i, j, k be integers such that $1 \le i \le j \le k \le n$.

   (a) What is the *exact* probability that node j is a common ancestor of node i and node k?

   (b) What is the *exact* expected length of the unique path in T from node i to node k?

   Don't forget to prove that your answers are correct!

2. A *meldable priority queue* stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

   - MAKEQUEUE: Return a new priority queue containing the empty set.
   - FINDMIN(Q): Return the smallest element of Q (if any).
   - DELETEMIN(Q): Remove the smallest element in Q (if any).
   - INSERT(Q, x): Insert element x into Q, if it is not already there.
   - DECREASEKEY(Q, x, y): Replace an element $x \in Q$ with a smaller key y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
   - DELETE(Q, x): Delete the element $x \in Q$. The input is a pointer directly to the node in Q containing x.
   - MELD(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

   A simple way to implement such a data structure is to use a heap-ordered binary tree, where each node stores a key, along with pointers to its parent and two children. MELD can be implemented using the following randomized algorithm:

```
MELD(Q1, Q2):
  if Q1 is empty return Q2
  if Q2 is empty return Q1
  if key(Q1) > key(Q2)
      swap Q1 ↔ Q2
  with probability 1/2
      left(Q1) ← MELD(left(Q1), Q2)
  else
      right(Q1) ← MELD(right(Q1), Q2)
  return Q1
```

   (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (not just those constructed by the operations listed above), the expected running time of MELD(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if each left/right choice is made with equal probability?]*

   (b) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ expected time.)

1. Suppose we can insert or delete an element into a hash table in $O(1)$ time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

   - After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
   - After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

   Show that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$. *[Hint: Do not use potential functions.]*

2. Recall that a standard FIFO queue supports the following operations:

   - PUSH(x): Add item x to the back of the queue.
   - PULL(): Remove and return the first item at the front of the queue.

   It is easy to implement a queue using a doubly-linked list, so that it uses $O(n)$ space (where n is the number of items in the queue) and the worst-case time per operation is $O(1)$.

   (a) Now suppose we want to support the following operation instead of PULL:

   - MULTIPULL(k): Remove the first k items from the front of the queue, and return the kth item removed.

   Suppose we use the obvious algorithm to implement MULTIPULL:

   ```
   MULTIPULL(k):
     for i ← 1 to k
         x ← PULL()
     return x
   ```

   Prove that in any intermixed sequence of PUSH and MULTIPULL operations, the amortized cost of each operation is $O(1)$

   (b) Now suppose we *also* want to support the following operation instead of PUSH:

   - MULTIPUSH(x, k): Insert k copies of x into the back of the queue.

   Suppose we use the obvious algorithm to implement MULTIPUUSH:

   ```
   MULTIPUSH(k, x):
     for i ← 1 to k
         PUSH(x)
   ```

   Prove that for any integers ℓ and n, there is a sequence of ℓ MULTIPUSH and MULTIPULL operations that require $\Omega(n\ell)$ time, where n is the maximum number of items in the queue at any time. Such a sequence implies that the amortized cost of each operation is $\Omega(n)$.

   (c) Describe a data structure that supports arbitrary intermixed sequences of MULTIPUSH and MULTIPULL operations in $O(1)$ amortized cost per operation. Like a standard queue, your data structure should use only $O(1)$ space per item.

3. (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any enqueue or dequeue operation is $O(1)$. The ***only*** access you have to the stacks is through the standard subroutines PUSH and POP. You may not implement your own nontrivial data structure or examine the internal contents of the stacks.

   (b) A *quack* is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that three operations are possible:

   - QUACKPUSH(x): add a new item x to the left end of the list
   - QUACKPOP(): remove and return the item on the left end of the list;
   - QUACKPULL(): remove the item on the right end of the list.

   Implement a quack using *three* stacks and $O(1)$ additional memory, so that the amortized time for any QUACKPUSH, QUACKPOP, or QUACKPULL operation is $O(1)$. In particular, each element in the quack must be stored in *exactly one* of the three stacks. Again, you are ***only*** allowed to access the component stacks through the interface functions PUSH and POP.

In both problems in this homework, your goal is to design a data structure that efficiently maintains a collection of strings (sequences of characters) subject to some or all of the following operations:

- NEWSTRING(a) creates a new string of length 1 containing only the character a and returns a pointer to that string.
- CONCAT(S, T) removes the strings S and T (given by pointers) from the data structure, adds the concatenated string ST to the data structure, and returns a pointer to the new string.
- SPLIT(S, k) removes the string S (given by a pointer) from the data structure, adds the substrings $S[1..k]$ and $S[k+1..length(S)]$ to the data structure, and returns pointers to the two new strings. You can safely assume that $1 \le k \le length(S) - 1$.
- REVERSE(S) removes the string S (given by a pointer) from the data structure, adds the reversal of S to the data structure, and returns a pointer to the new string.
- LOOKUP(S, k) returns the kth character in string S (given by a pointer), or NULL if the length of S is less than k.

For example, we can build the strings `SPLAYTREE` and `UNIONFIND` with 18 calls to NEWSTRING and 16 calls to CONCAT. Further operations modify the collection of strings as follows:

operation	result	stored strings
REVERSE(SPLAYTREE)	EERTYALPS	{**EERTYALPS**, UNIONFIND}
SPLIT(EERTYALPS, 5)	EERTY, ALPS	{**EERTY**, **ALPS**, UNIONFIND}
SPLIT(UNIONFIND, 3)	UNI, ONFIND	{EERTY, ALPS, **UNI**, **ONFIND**}
REVERSE(ONFIND)	DNIFNO	{EERTY, ALPS, UNI, **DNIFNO**}
CONCAT(UNI, EERTY)	UNIEERTY	{**UNIEERTY**, ALPS, DNIFNO}
SPLIT(UNIEERTY, 5)	UNIEE, RTY	{**UNIEE**, **RTY**, ALPS, DNIFNO}
NEWSTRING(LOOKUP(UNIEE, 5))	E	{**E**, UNIEE, RTY, ALPS, DNIFNO}

Except for NEWSTRING and LOOKUP, these operations are destructive; at the end of the sequence above, the string `ONFIND` is no longer stored in the data structure.

In both of the following problems, you need to describe a data structure that uses $O(n)$ space, where n is the sum of the stored string lengths, describe your algorithms for the various operations, prove that your algorithms are correct, and prove that they have the necessary running times. (Most of these steps should be very easy.)

---

1. Describe a data structure that supports the following operations:
   - SPLIT, CONCAT, and LOOKUP in $O(\log n)$ time (either worst-case, expected, or amortized).
   - NEWSTRING in $O(1)$ worst-case and amortized time.
   - ***5 points extra credit:*** REVERSE in $O(1)$ worst-case and amortized time.

2. Describe a data structure that supports the following operations:
   - CONCAT in $O(\log n)$ amortized time.
   - NEWSTRING and LOOKUP in $O(1)$ worst-case and amortized time.
   - ***5 points extra credit:*** REVERSE in $O(1)$ worst-case and amortized time.

   This data structure does not need to support SPLIT at all.

---

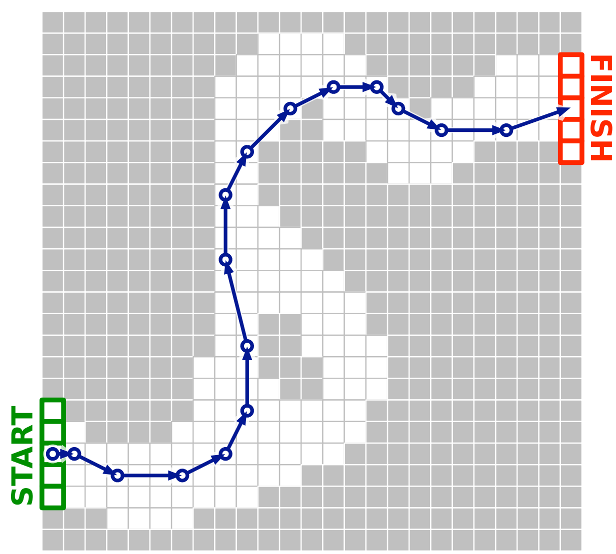
1. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[1] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

   Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0,0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position inside the finishing area.

   Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the 'starting area' is the first column, and the 'finishing area' is the last column.

   Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting area to the finishing area of a given racetrack.

velocity	position
$(0,0)$	$(1,5)$
$(1,0)$	$(2,5)$
$(2,-1)$	$(4,4)$
$(3,0)$	$(7,4)$
$(2,1)$	$(9,5)$
$(1,2)$	$(10,7)$
$(0,3)$	$(10,10)$
$(-1,4)$	$(9,14)$
$(0,3)$	$(9,17)$
$(1,2)$	$(10,19)$
$(2,2)$	$(12,21)$
$(2,1)$	$(14,22)$
$(2,0)$	$(16,22)$
$(1,-1)$	$(17,21)$
$(2,-1)$	$(19,20)$
$(3,0)$	$(22,20)$
$(3,1)$	$(25,21)$



A 16-step Racetrack run, on a 25×25 track. This is *not* the shortest run on this track.

[1]The actual game is a bit more complicated than the version described here. See http://harmmade.com/vectorracer/ for an excellent online version.

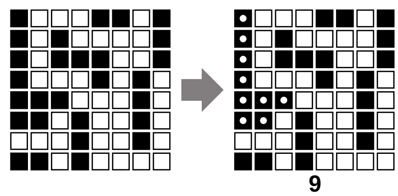
2. An ***Euler tour*** of a graph G is a walk that starts and ends at the same vertex and traverses every edge of G exactly once.

   (a) Prove that a connected undirected graph G has an Euler tour if and only if every vertex in G has even degree.

   (b) Describe and analyze an algorithm that constructs an Euler tour of a given graph G, or correctly reports that no such tour exists.

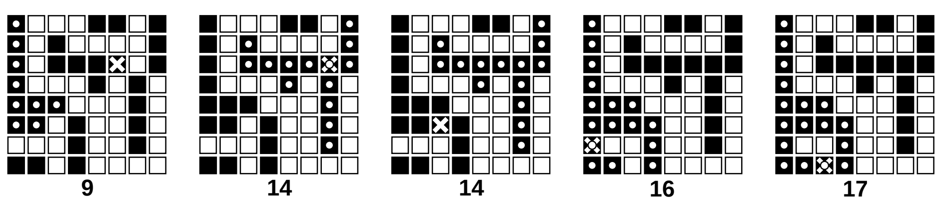
3. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in a given $n \times n$ bitmap $B[1..n, 1..n]$.

   For example, given the bitmap below as input, your algorithm should return the number 9, because the largest conected black component (marked with white dots on the right) contains nine pixels.



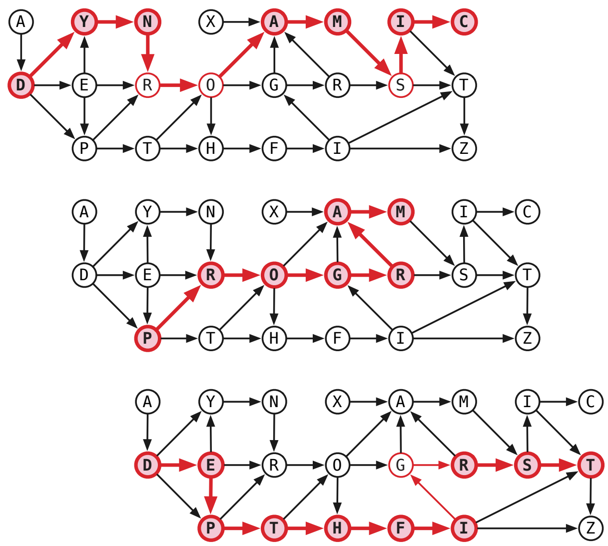
   (b) Design and analyze an algorithm Blacken(i, j) that colors the pixel $B[i, j]$ black and returns the size of the largest black component in the bitmap. For full credit, the *amortized* running time of your algorithm (starting with an all-white bitmap) must be as small as possible.

   For example, at each step in the sequence below, we blacken the pixel marked with an X. The largest black component is marked with white dots; the number underneath shows the correct output of the Blacken algorithm.



   (c) What is the *worst-case* running time of your Blacken algorithm?

1. Let G be a directed acyclic graph where each node has a label from some finite alphabet; different nodes may have the same label. Any directed path in G has a ***signature***, which is the string defined by concatenating the labels of its vertices. A ***subsequence*** of G is a subsequence of the signature of some directed path in G. For example, the strings `DYNAMIC`, `PROGRAM`, and `DETPHFIRST` are all subsequences of the dag shown below; in fact, `PROGRAM` is the signature of a path.



Describe and analyze an algorithm to compute the length of the longest common subsequence of two given directed acyclic graphs, that is, the longest string that is a subsequence of both dags.

2. Suppose you are given a graph G with weighted edges and a minimum spanning tree T of G.

   (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.

   (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

   In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. *[Hint: Consider the cases $e \in T$ and $e \notin T$ separately.]*

3. When there is more than one shortest paths from one node s to another node t, it is often most convenient to choose the shortest path with the fewest edges; call this the ***best path*** from s to t. For instance, if nodes represent cities and edge lengths represent costs of flying between cities, there could be many ways to fly from city s to city t for the same cost; the most desirable these schedules is the one with the fewest flights.

   Suppose we are given a directed graph G with positive edge weights and a source vertex s in G. Describe and analyze an algorithm to compute *best* paths in G from s to every other vertex. *[Hint: What is the actual output of your algorithm? If possible, use one of the standard shortest-path algorithms as a black box.]*

Orlin's algorithm computes maximum flows in $O(VE)$ time.
You may use this algorithm as a black box.

1. Suppose you are given a flow network G with ***integer*** edge capacities and an ***integer*** maximum flow f^* in G. Describe algorithms for the following operations:

   (a) INCREMENT(e): Increase the capacity of edge e by 1 and update the maximum flow.

   (b) DECREMENT(e): Decrease the capacity of edge e by 1 and update the maximum flow.

   Both algorithms should modify f^* so that it is still a maximum flow, more quickly than recomputing a maximum flow from scratch.

2. Suppose we are given a set of boxes, each specified by its height, width, and depth in centimeters. All three dimensions of each box lie strictly between 25cm and 50cm; box dimensions are *not* necessarily integers. As you might expect, one box can be placed inside another box if, possibly after rotating one or both boxes, each dimension of the first box is smaller than the corresponding dimension of the second box. Boxes can be nested recursively.

   Call a box is *visible* if it is not inside another box. Describe and analyze an algorithm to nest the boxes so that the number of visible boxes is as small as possible.

3. The University of Southern North Dakota at Hoople has hired you to write an algorithm to schedule their final exams. Each semester, USNDH offers n different classes. There are r different rooms on campus and t different time slots in which exams can be offered. You are given two arrays $E[1..n]$ and $S[1..r]$, where $E[i]$ is the number of students enrolled in the ith class, and $S[j]$ is the number of seats in the jth room. At most one final exam can be held in each room during each time slot. Class i can hold its final exam in room j only if $E[i] < S[j]$.

   Describe and analyze an efficient algorithm to assign a room and a time slot to each class (or report correctly that no such assignment is possible).

This homework is optional. Any problem that you do not submit will be automatically forgiven.

A useful list of NP-hard problems appears on the next page.

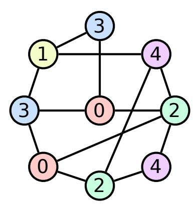
---

1. In the task scheduling problem, we are given n tasks, identified by the integers 1 through n, and a set of precedence constraints of the form "Task i must be executed before task j." A ***feasible schedule*** is an ordering of the n tasks that satisfies all the given precedence constraints.

   (a) Given a set of tasks and precedence constraints, describe and analyze a polynomial-time algorithm to determine whether a feasible schedule exists.

   (b) Suppose we are given a set of precedence constraints for which there is no feasible schedule. In this case, we would like a schedule that violates the minimum number of precedence constraints. Prove that finding such a schedule is NP-hard.

2. Recall that a 5-coloring of a graph G is a function that assigns each vertex of G a 'color' from the set $\{0, 1, 2, 3, 4\}$, such that for any edge uv, vertices u and v are assigned different 'colors'. A 5-coloring is *careful* if the colors assigned to adjacent vertices are not only distinct, but differ by more than 1 (mod 5). Prove that deciding whether a given graph has a careful 5-coloring is NP-hard. *[Hint: Reduce from the standard* 5Color *problem.]*



A careful 5-coloring.

3. (a) A ***tonian path*** in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a given graph has a tonian path is NP-hard.

   (b) A ***tonian cycle*** in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a given graph has a tonian cycle is NP-hard. *[Hint: Use part (a).]*

You may assume the following problems are NP-hard:

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

PlanarCircuitSat: Given a boolean circuit drawn in the plane so that no two wires cross, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

Max2Sat: Given a boolean formula in conjunctive normal form, with exactly two literals per clause, what is the largest number of clauses that can be satisfied by an assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

MaxCut: Given a graph G, what is the size (number of edges) of the largest bipartite subgraph of G?

HamiltonianCycle: Given a graph G, is there a cycle in G that visits every vertex exactly once?

HamiltonianPath: Given a graph G, is there a path in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of n positive integers, can X be partitioned into $n/3$ three-element subsets, all with the same sum?

CS 473: Undergraduate Algorithms, Fall 2012

Headbanging 0: Induction!

August 28 and 29

---

1. Prove that any non-negative integer can be represented as the sum of distinct powers of 2. ("Write it in binary" is not a proof; it's just a restatement of what you have to prove.)

2. Prove that any integer can be represented as the sum of distinct powers of -2.

3. Write ***four different*** proofs that any n-node tree has exactly $n - 1$ edges.

Take-home points:

- Induction is recursion. Recursion is induction.
- All induction is strong/structural induction. There is absolutely no point in using a *weak* induction hypothesis. None. Ever.
- To prove that all snarks are boojums, start with an *arbitrary* snark and remove some tentacles. Do not start with a smaller snark and try to add tentacles. Snarks don't like that.
- Every induction proof requires an exhaustive case analysis. Write down the cases. Make sure they're exhaustive.
- Do the most general cases first. Whatever is left over are the base cases.
- The empty set is the best base case.

Khelm is Warsaw. Warsaw is Khelm. Khelm is Warsaw. Zay gezunt!
Warsaw is Khelm. Khelm is Warsaw. Warsaw is Khelm. For gezunt!

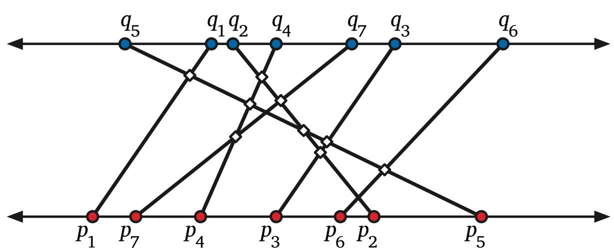
— Golem (feat. Amanda Palmer), "Warsaw is Khelm", *Fresh Off Boat* (2006)

1. An *inversion* in an array $A[1\,..\,n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward).

   Describe and analyze a divide-and-conquer algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. Assume all the elements of the input array are distinct.

2. Suppose you are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Create a set of n line segments by connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time. *[Hint: Use your solution to problem 1.]*

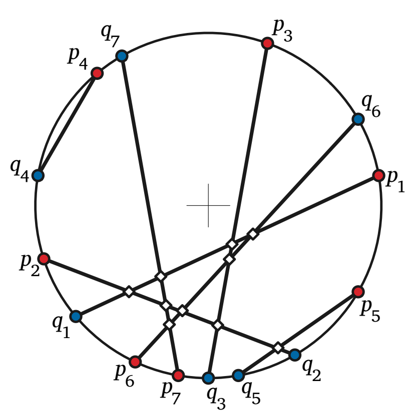
   Assume a reasonable representation for the input points, and assume the x-coordinates of the input points are distinct. For example, for the input shown below, your algorithm should return the number 9.



Nine intersecting pairs of segments with endpoints on parallel lines.

3. Now suppose you are given two sets $\{p_1, p_2, \ldots, p_n\}$ and $\{q_1, q_2, \ldots, q_n\}$ of n points *on the unit circle*. Connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect in $O(n \log^2 n)$ time. *[Hint: Use your solution to problem 2.]*

   Assume a reasonable representation for the input points, and assume all input points are distinct. For example, for the input shown below, your algorithm should return the number 10.



Ten intersecting pairs of segments with endpoints on a circle.

4. **To think about later:** Solve the previous problem in $O(n \log n)$ time.

A subsequence is anything obtained from a sequence by deleting a subset of elements; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings `I`, `PRO`, `DAMMM`, `NROAIG`, and `DYNAMICPROGRAMMING` are all subsequences of the string `DYNAMICPROGRAMMING`.

1. Suppose we are given two arrays $A[1..m]$ and $B[1..m]$. A *common subsequence* of A and B is any subsequence of A that is also a subsequence of B. For example, `AMI` is a common subsequence of `DYNAMIC` and `PROGRAMMING`. Describe and analyze an efficient algorithm to compute the length of the *longest* common subsequence of A and B.

2. Describe and analyze an efficient algorithm to compute the length of the longest common subsequence of *three* given arrays $A[1..l]$, $B[1..m]$, and $C[1..n]$.

3. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, such that the characters of X and Y remain in the same order. For example, given the strings `'dynamic'` and `'programming'`, the string `'prodgyrnamammiincg'` is indeed a shuffle of the two:

$$\texttt{pro}^{\texttt{d}}\texttt{g}^{\texttt{y}}\texttt{r}^{\texttt{nam}}\texttt{ammi}^{\texttt{i}}\texttt{n}^{\texttt{c}}\texttt{g}$$

   Given three strings $A[1..m]$, $B[1..n]$ and $C[1..m+n]$, describe an algorithm to determine whether C is a shuffle of A and B.

In all cases, first describe a recursive algorithm and only then transform it into an iterative algorithm.

1. A set of vectors A is said to be linearly independent if no $v \in A$ can be expressed as a linear combination of the vectors in $A - \{v\}$. Given a set of vectors S, describe an efficient algorithm for finding a linearly independent subset of S with the maximum possible size. Assume you are given a function that can check if n vectors are linearly independent in $O(n^2)$ time.

2. You live in a country with n different types of coins, with values $1, 2, 2^2, \ldots, 2^{n-1}$. Describe an efficient algorithm for determining how to make change for a given value W using the least possible number of coins.

3. Let X be a set of n intervals on the real line. A *proper coloring* of X assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Describe an efficient algorithm to compute the minimum number of colors needed to properly color X. Assume that your input consists of two array $L[1..n]$ and $R[1..n]$, where $L[i]$ and $R[i]$ are the left and right endpoints of the ith interval.

1. Any connected graph with n nodes and n edges has exactly one cycle.

2. Any n-node binary tree can be transformed into any other n-node binary tree by a sequence of at most $2n-2$ rotations.

3. If $d_1, \ldots, d_n$ are positive integers such that $\sum_{i=1}^{n} d_i = 2n-2$, then there is a tree having $d_1, \ldots, d_n$ as its vertex degrees. For examples, $\{1,1,1,1,1,5\}$ has sum $2 \cdot 6 - 2$, and so the hypothesis is satisfied. The tree that is the star with five leaves has vertex degrees $\{1,1,1,1,1,5\}$. Also, $\{1,1,1,1,2,3,3\}$ has sum $2 \cdot 7 - 2$, and the perfect binary three with depth 2 has vertex degrees $\{1,1,1,1,2,3,3\}$.

1. Prove that the expected space requirement of a skip list constructed on n numbers is $O(n)$.

2. Let S be a set of n points in the plane. A point p in S is called ***maximal*** (or *Pareto-optimal*) if no other point in S is both above and to the right of p. If each point in S is chosen independently and uniformly at random from the unit square $[0,1] \times [0,1]$ what is the *exact* expected number of Pareto-optimal points in S.

3. A *data stream* is an extremely long sequence of items that you can read only once. A data stream algorithm looks roughly like this:

```
DoSomethingInteresting(stream S):
repeat
    x ← next item in S
    ⟨⟨ do something fast with x ⟩⟩
until S ends
return ⟨⟨ something ⟩⟩
```

   Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, *without knowing the length of the stream in advance*. Your algorithm should spend $O(1)$ time per stream element and use $O(1)$ space (not counting the stream itself).

1. An ***extendable array*** is a data structure that stores a sequence of items and supports the following operations.

   - AddToFront(x) adds x to the *beginning* of the sequence.
   - AddToEnd(x) adds x to the *end* of the sequence.
   - Lookup(k) returns the kth item in the sequence, or Null if the current length of the sequence is less than k.

   Describe a ***simple*** data structure that implements an extendable array. Your AddToFront and AddToBack algorithms should take $O(1)$ *amortized* time, and your Lookup algorithm should take $O(1)$ *worst-case* time. The data structure should use $O(n)$ space, where n is the ***current*** length of the sequence.

2. An ***ordered stack*** is a data structure that stores a sequence of items and supports the following operations.

   - OrderedPush(x) removes all items smaller than x from the beginning of the sequence and then adds x to the beginning of the sequence.
   - Pop deletes and returns the first item in the sequence (or Null if the sequence is empty).

   Suppose we implement an ordered stack with a simple linked list, using the obvious OrderedPush and Pop algorithms. Prove that if we start with an empty data structure, the amortized cost of each OrderedPush or Pop operation is $O(1)$.

3. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array $A[1..n]$ that stores the height of n buildings on a city block, indexed from west to east. Building i has a good view of Lake Michigan if and only if every building to the east of i is shorter than i.

   Here is an algorithm that computes which buildings have a good view of Lake Michigan. What is the running time of this algorithm?

```
GoodView(A[1..n]):
  initialize a stack S
  for i ← 1 to n
      while (S not empty and A[i] > A[Top(S)])
          Pop(S)
      Push(S, i)
  return S
```

1. Suppose we want to maintain a dynamic set of numbers, subject to the following operations:

   - INSERT(x): Add x to the set. (Assume x is not already in the set.)
   - PRINT&DELETEBETWEEN(a, b): Print every element x in the range $a \le x \le b$ in increasing order, and then delete those elements from the set.

   For example, if the current set is $\{1, 5, 3, 4, 8\}$, then

   - PRINT&DELETEBETWEEN($4, 6$) prints the numbers 4 and 5 and changes the set to $\{1, 3, 8\}$.
   - PRINT&DELETEBETWEEN($6, 7$) prints nothing and does not change the set.
   - PRINT&DELETEBETWEEN($0, 10$) prints the sequence $1, 3, 4, 5, 8$ and deletes everything.

   Describe a data structure that supports both operations in $O(\log n)$ amortized time, where n is the *current* number of elements in the set.

   [Hint: As warmup, argue that the obvious implementation of PRINT&DELETEBETWEEN*—while the successor of a is less than or equal to b, print it and delete it—runs in $O(\log N)$ amortized time, where N is the* ***maximum*** *number of elements that are ever in the set.]*

2. Describe a data structure that stores a set of numbers (which is initially empty) and supports the following operations in $O(1)$ amortized time:

   - INSERT(x): Insert x into the set. (You can safely assume that x is not already in the set.)
   - FINDMIN: Return the smallest element of the set (or NULL if the set is empty).
   - DELETEBOTTOMHALF: Remove the smallest $\lceil n/2 \rceil$ elements the set. (That's smallest by *value*, not smallest by *insertion time*.)

3. Consider the following solution for the union-find problem, called *union-by-weight*. Each set leader $\overline{x}$ stores the number of elements of its set in the field *weight*($\overline{x}$). Whenever we UNION two sets, the leader of the *smaller* set becomes a new child of the leader of the *larger* set (breaking ties arbitrarily).

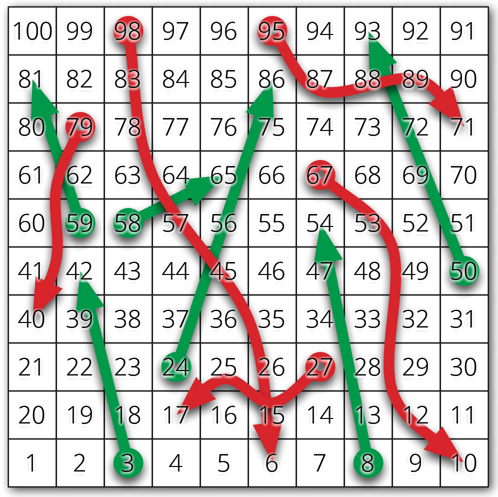
```
MAKESET(x):
    parent(x) ← x
    weight(x) ← 1
```

```
FIND(x):
    while x ≠ parent(x)
        x ← parent(x)
    return x
```

```
UNION(x, y)
    x̄ ← FIND(x)
    ȳ ← FIND(y)
    if weight(x̄) > weight(ȳ)
        parent(ȳ) ← x̄
        weight(x̄) ← weight(x̄) + weight(ȳ)
    else
        parent(x̄) ← ȳ
        weight(x̄) ← weight(x̄) + weight(ȳ)
```

   Prove that if we always use union-by-weight, the *worst-case* running time of FIND(x) is $O(\log n)$, where n is the cardinality of the set containing x.

1. ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.



A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k. If the token ends the move at the *top* end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the *bottom* end of a ladder, it climbs up to the top of that ladder.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

2. Suppose you are given a set of n jobs, indexed from 1 to n, together with a list of *precedence constraints*. Each precedence constraint is a pair (i, j), indicating that job i must be finished before job j begins. Describe and analyze an algorithm that either finds an schedule for executing all n jobs on a single processor, such that all precedence constraints are satisfied, or correctly reports that no such schedule is possible.

3. For the previous problem, describe and analyze an algorithm to determine whether there is a *unique* schedule for a given number of jobs that satisfies a given set of precedence constraints.

1. Let $G = (V, E)$ be a directed graph. If the indedgree of each vertex is at least 1 prove that G contains a cycle. Give an algorithm to find a cycle in such a graph. How fast is your algorithm?

2. At a party with n people $P_1, \ldots, P_n$, certain pairs of individuals cannot stand each other. Given a list of such pairs, determine if we can divide the n people into two groups such that all the people in both group are amicable, that is, they can stand each other.

3. Given a graph $G = (V, E)$ prove that one can orient each edge so that after the orientation, the indegree and outdegree of each vertex differ by at most 1. How fast can you compute this orientation? **[Hint: Can you change the graph so that it has an Euler tour?]**

1. Describe and analyze an algorithm that finds a *maximum* spanning tree of a graph, that is, the spanning tree with largest total weight.

2. During your CS 473 final exam, you are given a specific undirected graph $G = (V, E)$ with non-negative edge weights, and you are asked to compute both the minimum spanning tree of G and the tree of shortest paths in G rooted at some fixed source vertex s.

   Two and a half hours into the exam, Jeff announces that there was a mistake in the exam; every edge weight should be increased by 1. Well, that's just great. Now what?!

   (a) Do you need to recompute the minimum spanning tree? Either prove that increasing all edge weights by 1 cannot change the minimum spanning tree, or give an example where the minimum spanning tree changes.

   (b) Do you need to recompute the shortest path tree rooted at s? Either prove that increasing all edge weights by 1 cannot change the shortest path tree, or give an example where the shortest path tree changes.

3. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of every flight on the planet.

   Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights. *[Hint: Modify the input data and apply Dijkstra's algorithm.]*

1. Suppose you are given the following information:

   - A directed graph $G = (V, E)$.
   - Two vertices s and t in V.
   - A positive edge capacity function $c\colon E \to \mathbb{R}^+$.
   - Another function $f : E \to \mathbb{R}$

   Describe and analyze an algorithm to determine whether f is a maximum (s, t)-flow in G.

2. Describe an efficient algorithm to determine whether a given flow network contains a *unique* maximum flow.

3. Suppose that a flow network has vertex capacities in addition to edge capacities. That is, the total amount of flow into or out of any vertex v is at most the capacity of v:

$$\sum_u f(u \to v) \;=\; \sum_w f(v \to w) \;\le\; c(v)$$

   Describe and analyze an algorithm to compute maximum flows with this additional constraint.

1. The UIUC Faculty Senate has decided to convene a committee to determine whether Chief Illiniwek should become the official ~~mascot~~ *symbol* of the University of Illinois Global Campus.[1] Exactly one faculty member must be chosen from each academic department to serve on this committee. Some faculty members have appointments in multiple departments, but each committee member will represent only one department. For example, if Prof. Blagojevich is affiliated with both the Department of Corruption and the Department of Stupidity, and he is chosen as the Stupidity representative, then someone else must represent Corruption. Finally, University policy requires that any committee on virtual ~~mascots~~ *symbols* must contain the same number of assistant professors, associate professors, and full professors. Fortunately, the number of departments is a multiple of 3.

   Describe an efficient algorithm to select the membership of the Global Illiniwek Committee. Your input is a list of all UIUC faculty members, their ranks (assistant, associate, or full), and their departmental affiliation(s). There are n faculty members and $3k$ departments.

2. *Ad-hoc networks* are made up of low-powered wireless devices. In principle[2], these networks can be used on battlefields, in regions that have recently suffered from natural disasters, and in other hard-to-reach areas. The idea is that a large collection of cheap, simple devices could be distributed through the area of interest (for example, by dropping them from an airplane); the devices would then automatically configure themselves into a functioning wireless network.

   These devices can communicate only within a limited range. We assume all the devices are identical; there is a distance D such that two devices can communicate if and only if the distance between them is at most D.

   We would like our ad-hoc network to be reliable, but because the devices are cheap and low-powered, they frequently fail. If a device detects that it is likely to fail, it should transmit its information to some other *backup* device within its communication range. We require each device x to have k potential backup devices, all within distance D of x; we call these k devices the ***backup set*** of x. Also, we do not want any device to be in the backup set of too many other devices; otherwise, a single failure might affect a large fraction of the network.

   So suppose we are given the communication radius D, parameters b and k, and an array $d[1..n, 1..n]$ of distances, where $d[i, j]$ is the distance between device i and device j. Describe an algorithm that either computes a backup set of size k for each of the n devices, such that no device appears in more than b backup sets, or reports (correctly) that no good collection of backup sets exists.

3. Given an undirected graph $G = (V, E)$, with three vertices u, v, and w, describe and analyze an algorithm to determine whether there is a path from u to w that passes through v.

[1] Thankfully, the Global Campus has faded into well-deserved obscurity, thanks in part to the 2009 admissions scandal. Imagine MOOCs, but with the same business model and faculty oversight as the University of Phoenix.

[2] but not really in practice

A useful list of NP-hard problems appears on the next page.

The KNAPSACK problem is the following. We are given a set of n objects, each with a positive integer ***size*** and a positive integer *value*; we are also given a positive integer B. The problem is to choose a subset of the n objects with maximum total value, whose total size is at most B. Let V denote the sum of the values of all objects.

1. Describe an algorithm to solve KNAPSACK in time polynomial in n and V.

2. Prove that the KNAPSACK problem is NP-hard.

   Given the algorithm from problem 1, why doesn't this immediately imply that P=NP?

---

3. ***Facility location*** is a family of problems that require choosing a subset of facilities (for example, gas stations, cell towers, garbage dumps, Starbuckses, ...) to serve a given set of locations cheaply. In its most abstract formulation, the input to the facility location problem is a pair of arrays $Open[1..n]$ and $Connect[1..n, 1..m]$, where

   - $Open[i]$ is the cost of opening a facility i, and
   - $Connect[i, j]$ is the cost of connecting facility i to location j.

   Given these two arrays, the problem is to compute a subset $I \subseteq \{1, 2, \ldots, n\}$ of the n facilities to open and a function $\phi : \{1, 2, \ldots, m\} \to I$ that assigns an open facility to each of the m locations, so that the total cost

   $$\sum_{i \in I} Open[i] + \sum_{j=1}^{m} Connect[\phi(j), j]$$

   is minimized. Prove that this problem is NP-hard.

You may assume the following problems are NP-hard:

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

PlanarCircuitSat: Given a boolean circuit drawn in the plane so that no two wires cross, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

Max2Sat: Given a boolean formula in conjunctive normal form, with exactly two literals per clause, what is the largest number of clauses that can be satisfied by an assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

MaxCut: Given a graph G, what is the size (number of edges) of the largest bipartite subgraph of G?

HamiltonianCycle: Given a graph G, is there a cycle in G that visits every vertex exactly once?

HamiltonianPath: Given a graph G, is there a path in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of n positive integers, can X be partitioned into $n/3$ three-element subsets, all with the same sum?

This exam lasts 120 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. Each of these ten questions has one of the following five answers:

   A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$

   (a) What is $\dfrac{n^3 + 3n^2 - 5n + 1}{4n^2 - 2n + \sqrt{3}}$?

   (b) What is $\displaystyle\sum_{i=1}^{n} \sum_{j=1}^{i} 5$?

   (c) What is $\displaystyle\sum_{i=1}^{n} \left(\frac{i}{n} + \frac{n}{i} \right)$?

   (d) How many bits are required to write the nth Fibonacci number F_n in binary?

   (e) What is the solution to the recurrence $E(n) = E(n-3) + 2n - 1$?

   (f) What is the solution to the recurrence $F(n) = 2F(n/3) + 2F(n/6) + n$?

   (g) What is the solution to the recurrence $G(n) = 12\, G(n/4) + n^2$?

   (h) What is the worst-case depth of an n-node binary tree?

   (i) Consider the following recursive function, which is defined in terms of a fixed array $X[1\,..\,n]$:

   $$WTF(i, j) = \begin{cases} 0 & \text{if } i \le 0 \text{ or } j \le 0 \\ X[j] + WTF(i-1, j) + WTF(i, \lfloor j/2 \rfloor) & \text{otherwise} \end{cases}$$

   How long does it take to compute $WTF(n, n)$ using dynamic programming?

   (j) How many seconds does it take for a 10-megapixel image taken by the Curiosity Rover to be encoded, transmitted from Mars to Earth, decoded, and tweeted?

2. A palindrome is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `HANNAH`, or `SATOR AREPO TENET OPERA ROTAS`. Describe and analyze an algorithm to find the length of the longest subsequence of a given string that is also a palindrome.

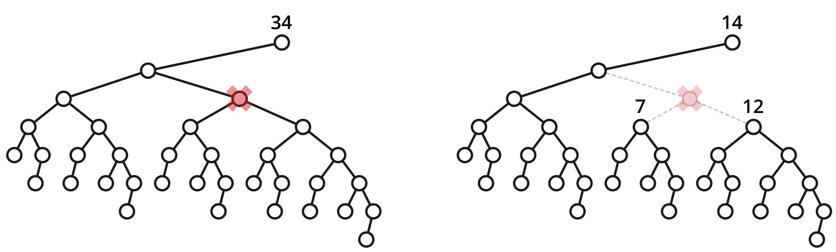
   For example, the longest palindrome subsequence of `MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM` is `MHYMRORMYHM`, so given that string as input, your algorithm should return the integer 11.

3. ***Prove*** that any integer—positive, negative, or zero—can be represented as the sum of distinct powers of -2. For example:

$$\begin{aligned}
4 &= 4 &&= (-2)^2 \\
8 &= 16-8 &&= (-2)^4+(-2)^3 \\
15 &= 16-2+1 &&= (-2)^4+(-2)^1+(-2)^0 \\
-16 &= -32+16 &&= (-2)^5+(-2)^4 \\
23 &= 64-32-8-2+1 &&= (-2)^6+(-2)^5+(-2)^3+(-2)^1+(-2)^0 \\
-42 &= -32-8-2 &&= (-2)^5+(-2)^3+(-2)^1
\end{aligned}$$

4. Let T be a binary tree with n vertices. Deleting any vertex v splits T into at most three subtrees, containing the left child of v (if any), the right child of v (if any), and the parent of v (if any). We call v a ***central*** vertex if each of these smaller trees has at most $n/2$ vertices.

   Describe and analyze an algorithm to find a central vertex in a given binary tree.



Deleting a central vertex in a 34-node binary tree,
leaving subtrees with 14 nodes, 7 nodes, and 12 nodes.

5. Let $n = 2^\ell - 1$ for some positive integer ℓ. Suppose someone claims to hold an array $A[1\,..\,n]$ of *distinct* ℓ-bit strings; thus, exactly one ℓ-bit string does *not* appear in A. Suppose further that the ***only*** way we can access A is by calling the function FetchBit(i,j), which returns the jth bit of the string $A[i]$ in $O(1)$ time.

   Describe an algorithm to find the missing string in A using only $O(n)$ calls to FetchBit.

This exam lasts 120 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

Given any positive integer k as input, the function call RANDOM(k) returns an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, k\}$ in $O(1)$ time.

---

1. (a) The *left spine* of a binary tree is the path from the root to the leftmost node. For example, if the root has no left child, the left spine contains only the root. What is the expected number of nodes in the left spine of an n-node treap? *[Hint: What is the probability that the node with kth smallest search key is in the left spine?]*

   (b) What is the expected number of leaves in an n-node treap? *[Hint: What is the probability that the node with kth smallest search key is a leaf?]*

   You do ***not*** need to prove that your answers are correct.

2. Recall that a *queue* maintains a sequence of items subject to the following operations.

   - PUSH(x): Add item x to the end of the sequence.
   - PULL: Remove and return the item at the beginning of the sequence.
   - SIZE: Return the current number of items in the sequence.

   It is easy to support all three operations in $O(1)$ worst-case time, using a doubly-linked list and a counter. Now consider the following new operation, which removes every tenth element from the queue, starting at the beginning, in $\Theta(n)$ worst-case time.

```
DECIMATE:
  n ← SIZE
  for i ← 0 to n − 1
      if i mod 10 = 0
          PULL   ⟨⟨result discarded⟩⟩
      else
          PUSH(PULL)
```

   Prove that in any intermixed sequence of PUSH, PULL, and DECIMATE operations, the amortized cost of each operation is $O(1)$.

3. Let $G = (V, E)$ be a connected undirected graph. For any vertices u and v, let $d_G(u, v)$ denote the length of the shortest path in G from u to v. For any *sets* of vertices A and B, let $d_G(A, B)$ denote the length of the shortest path in G from any vertex in A to any vertex in B:

$$d_G(A, B) = \min_{u \in A} \min_{v \in B} d_G(u, v).$$

   Describe and analyze a fast algorithm to compute $d_G(A, B)$, given the graph G and subsets A and B as input. You do ***not*** need to prove that your algorithm is correct.

4. Consider the following modification of the standard algorithm for incrementing a binary counter.

```
INCREMENT(B[0..∞]):
  i ← 0
  while B[i] = 1
      B[i] ← 0
      i ← i + 1
  B[i] ← 1
  SOMETHINGELSE(i)
```

The only difference from the standard algorithm is the function call at the end, to a black-box subroutine called SomethingElse.

Suppose we call Increment n times, starting with a counter with value 0. The amortized time of each Increment clearly depends on the running time of SomethingElse. Let $T(i)$ denote the worst-case running time of SomethingElse(i). For example, we proved in class that Increment algorithm runs in $O(1)$ amortized time when $T(i) = 0$.

(a) What is the amortized time per Increment if $T(i) = 42$?

(b) What is the amortized time per Increment if $T(i) = 2^i$?

(c) What is the amortized time per Increment if $T(i) = 4^i$?

(d) What is the amortized time per Increment if $T(i) = \sqrt{2}^i$?

(e) What is the amortized time per Increment if $T(i) = 2^i/(i+1)$?

You do ***not*** need to prove your answers are correct.

5. A *data stream* is a long sequence of items that you can only read only once, in order. Every data-stream algorithm looks roughly like this:

```
DOSOMETHINGINTERESTING(stream S):
  repeat
      x ← next item in S
      ⟨⟨do something fast with x⟩⟩
  until S ends
  return ⟨⟨something⟩⟩
```

Describe and analyze an algorithm that chooses one element uniformly at random from a data stream.

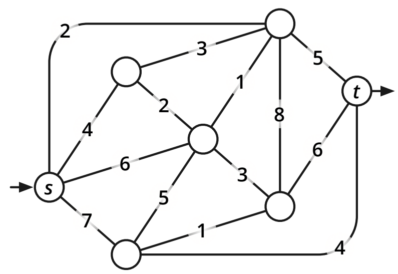
Your algorithm should spend $O(1)$ time per stream element and use only $O(1)$ space (excluding the stream itself). ***You do not know the length of the stream in advance***; your algorithm learns that the stream has ended only when a request for the next item fails.

You do ***not*** need to prove your algorithm is correct.

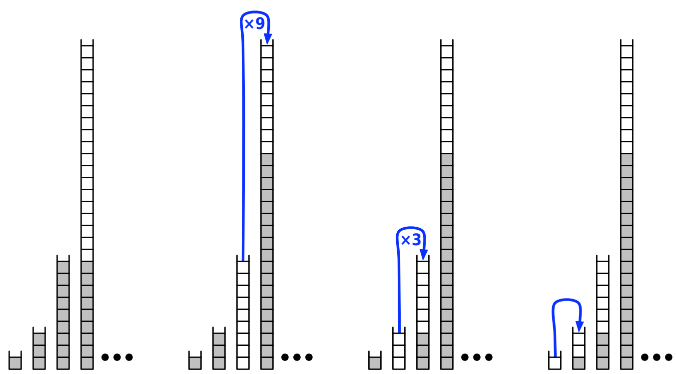
This exam lasts 180 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheets with your answers.

1. Clearly indicate the following structures in the weighted graph pictured below. Some of these subproblems have more than one correct answer.

   (a) A depth-first spanning tree rooted at s
   (b) A breadth-first spanning tree rooted at s
   (c) A shortest-path tree rooted at s
   (d) A minimum spanning tree
   (e) A minimum (s,t)-cut



2. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i, we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move all its members to S_{i+2}.) Moving a single element from one stack to the next takes $O(1)$ time.



Making room for one new element in a multistack.

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?
   (b) ***Prove*** that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack.

3. Suppose we are given an array $A[1..n]$ of numbers with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. A ***local minimum*** is an element $A[i]$ such that $A[i-1] \geq A[i]$ and $A[i] \leq A[i+1]$. For example, there are six local minima in the following array:

9	*7*	7	2	*1*	3	7	5	*4*	7	*3*	*3*	4	8	*6*	9

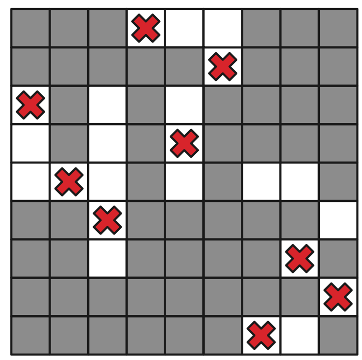
Describe and analyze an algorithm that finds a local minimum in the array A in $O(\log n)$ time.

4. Suppose we are given an n-digit integer X. Repeatedly remove one digit from either end of X (your choice) until no digits are left. The *square-depth* of X is the maximum number of perfect squares that you can see during this process. For example, the number 32492 has square-depth 3, by the following sequence of removals:

$$32492 \to \underline{\mathbf{3249}}\cancel{2} \to \underline{\mathbf{324}}\cancel{9} \to \cancel{3}24 \to \cancel{2}\underline{4} \to \cancel{4}.$$

Describe and analyze an algorithm to compute the square-depth of a given integer X, represented as an array $X[1..n]$ of n decimal digits. Assume you have access to a subroutine IsSQUARE that determines whether a given k-digit number (represented by an array of digits) is a perfect square ***in $O(k^2)$ time***.

5. Suppose we are given an $n \times n$ square grid, some of whose squares are colored black and the rest white. Describe and analyze an algorithm to determine whether tokens can be placed on the grid so that



- every token is on a white square;
- every row of the grid contains exactly one token; and
- every column of the grid contains exactly one token.

Your input is a two dimensional array $IsWhite[1..n, 1..n]$ of booleans, indicating which squares are white. (You solved an instance of this problem in the last quiz.)

6. Recall the following problem from Homework 2:

- 3WAYPARTITION: Given a set X of positive integers, determine whether there are three disjoint subsets $A, B, C \subseteq X$ such that $A \cup B \cup C = X$ and

$$\sum_{a \in A} a = \sum_{b \in B} b = \sum_{c \in C} c.$$

(a) ***Prove*** that 3WAYPARTITION is NP-hard. *[Hint: Don't try to reduce from 3SAT or 3COLOR; in this rare instance, the 3 is just a coincidence.]*

(b) In Homework 2, you described an algorithm to solve 3WAYPARTITION in $O(nS^2)$ time, where S is the sum of all elements of X. Why doesn't this algorithm imply that P=NP?

7. Describe and analyze efficient algorithms to solve the following problems:

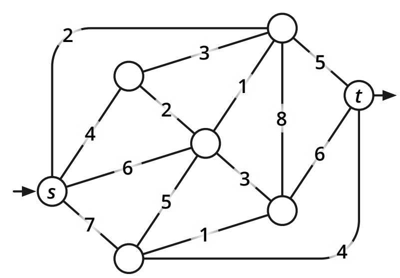
(a) Given an array of n integers, does it contain two elements a, b such that $a + b = 0$?

(b) Given an array of n integers, does it contain three elements a, b, c such that $a + b + c = 0$?

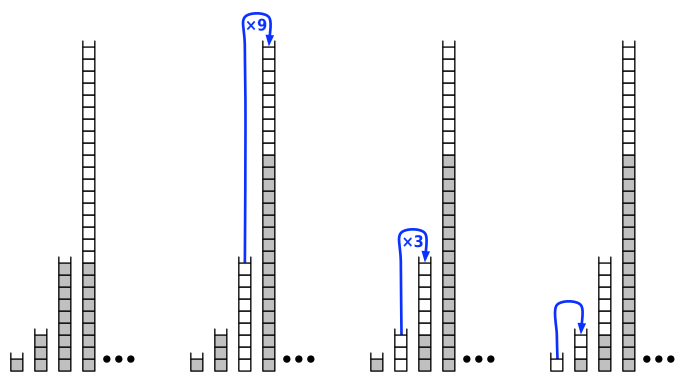
This exam lasts 180 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheets with your answers.

1. Clearly indicate the following structures in the weighted graph pictured below. Some of these subproblems have more than one correct answer.

   (a) A depth-first spanning tree rooted at s
   (b) A breadth-first spanning tree rooted at s
   (c) A shortest-path tree rooted at s
   (d) A minimum spanning tree
   (e) A minimum (s,t)-cut



2. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i, we first pop all the elements off S_i and push them onto stack S_{i+1} to make room. (Thus, if S_{i+1} is already full, we first recursively move all its members to S_{i+2}.) Moving a single element from one stack to the next takes $O(1)$ time.



Making room for one new element in a multistack.

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?
   (b) ***Prove*** that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack.

3. Describe and analyze an algorithm to determine, given an undirected graph $G = (V, E)$ and three vertices $u, v, w \in V$ as input, whether G contains a simple path from u to w that passes through v. You do ***not*** need to prove your algorithm is correct.

4. Suppose we are given an n-digit integer X. Repeatedly remove one digit from either end of X (your choice) until no digits are left. The *square-depth* of X is the maximum number of perfect squares that you can see during this process. For example, the number 32492 has square-depth 3, by the following sequence of removals:

$$32492 \rightarrow \underline{\mathbf{3249}}\cancel{2} \rightarrow \underline{\mathbf{324}}\cancel{9} \rightarrow \cancel{3}24 \rightarrow \cancel{2}\underline{\mathbf{4}} \rightarrow \cancel{4}.$$

Describe and analyze an algorithm to compute the square-depth of a given integer X, represented as an array $X[1\,..\,n]$ of n decimal digits. Assume you have access to a subroutine IsSquare that determines whether a given k-digit number (represented by an array of digits) is a perfect square ***in $O(k^2)$ time***.

5. Suppose we are given two *sorted* arrays $A[1\,..\,n]$ and $B[1\,..\,n]$ containing $2n$ distinct numbers. Describe and analyze an algorithm that finds the nth smallest element in the union $A \cup B$ in $O(\log n)$ time.

6. Recall the following problem from Homework 2:

   - 3WayPartition: Given a set X of positive integers, determine whether there are three disjoint subsets $A, B, C \subseteq X$ such that $A \cup B \cup C = X$ and
   $$\sum_{a \in A} a = \sum_{b \in B} b = \sum_{c \in C} c.$$

   (a) ***Prove*** that 3WayPartition is NP-hard. *[Hint: Don't try to reduce from 3SAT or 3COLOR; in this rare instance, the 3 is just a coincidence.]*

   (b) In Homework 2, you described an algorithm to solve 3WayPartition in $O(nS^2)$ time, where S is the sum of all elements of X. Why doesn't this algorithm imply that P=NP?

7. Describe and analyze efficient algorithms to solve the following problems:

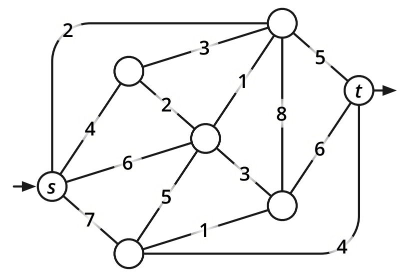
   (a) Given an array of n integers, does it contain two elements a, b such that $a + b = 0$?

   (b) Given an array of n integers, does it contain three elements a, b, c such that $a + b + c = 0$?

This exam lasts 180 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheets with your answers.

1. Clearly indicate the following structures in the weighted graph pictured below. Some of these subproblems have more than one correct answer.

   (a) A depth-first spanning tree rooted at s
   (b) A breadth-first spanning tree rooted at s
   (c) A shortest-path tree rooted at s
   (d) A minimum spanning tree
   (e) A minimum (s, t)-cut



2. Describe a data structure that stores a set of numbers (which is initially empty) and supports the following operations in $O(1)$ amortized time:

   - INSERT(x): Insert x into the set. (You can safely assume that x is not already in the set.)
   - FINDMIN: Return the smallest element of the set (or NULL if the set is empty).
   - DELETEBOTTOMHALF: Remove the smallest $\lceil n/2 \rceil$ elements the set. (That's smallest by *value*, not smallest by *insertion time*.)

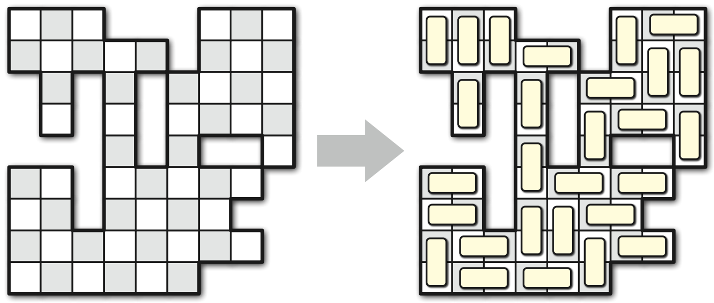
3. Suppose we are given two *sorted* arrays $A[1..n]$ and $B[1..n]$ containing $2n$ distinct numbers. Describe and analyze an algorithm that finds the nth smallest element in the union $A \cup B$ in $O(\log n)$ time.

4. Suppose you have a black-box subroutine QUALITY that can compute the 'quality' of any given string $A[1..k]$ in $O(k)$ time. For example, the quality of a string might be 1 if the string is a Québecois curse word, and 0 otherwise.

   Given an arbitrary input string $T[1..n]$, we would like to break it into contiguous substrings, such that the total quality of all the substrings is as large as possible. For example, the string `SAINTCIBOIREDESACRAMENTDECRISSE` can be decomposed into the substrings `SAINT + CIBOIRE + DE + SACRAMENT + DE + CRISSE`, of which three (or possibly four) are *sacres*.

   Describe an algorithm that breaks a string into substrings of maximum total quality, using the QUALITY subroutine.

5. Suppose you are given an $n \times n$ checkerboard with some of the squares removed. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether one can place dominos on the board so that each domino covers exactly two squares (meaning squares that have *not* been removed) and each square is covered by exactly one domino. Your input is a two-dimensional array $\mathit{Removed}[1..n, 1..n]$ of booleans, where $\mathit{Removed}[i, j] = \text{TRUE}$ if and only if the square in row i and column j has been removed. For example, for the board shown below, your algorithm should return TRUE.



6. Recall the following problem from Homework 2:

   - 3WAYPARTITION: Given a set X of positive integers, determine whether there are three disjoint subsets $A, B, C \subseteq X$ such that $A \cup B \cup C = X$ and

$$\sum_{a \in A} a = \sum_{b \in B} b = \sum_{c \in C} c.$$

   (a) ***Prove*** that 3WAYPARTITION is NP-hard. *[Hint: Don't try to reduce from* 3SAT *or* 3COLOR*; in this rare instance, the 3 is just a coincidence.]*

   (b) In Homework 2, you described an algorithm to solve 3WAYPARTITION in $O(nS^2)$ time, where S is the sum of all elements of X. Why doesn't this algorithm imply that P=NP?

7. Describe and analyze efficient algorithms to solve the following problems:

   (a) Given an array of n integers, does it contain two elements a, b such that $a + b = 0$?

   (b) Given an array of n integers, does it contain three elements a, b, c such that $a + b + c = 0$?

CS 473: Undergraduate Algorithms, Fall 2013
Homework 0

Due Tuesday, September 3, 2013 at 12:30pm

Quiz 0 (on the course Moodle page)
is also due Tuesday, September 3, 2013 at noon.

- **Please carefully read the course policies on the course web site.** These policies may be different than other classes you have taken. (For example: No late anything ever; "I don't know" is worth 25%, but "Repeat this for all n" is an automatic zero; ***every*** homework question requires a proof; collaboration is allowed, but you must cite your collaborators.) If you have *any* questions, please ask in lecture, in headbanging, on Piazza, in office hours, or by email.

- Homework 0 and Quiz 0 test your familiarity with prerequisite material—big-Oh notation, elementary algorithms and data structures, recurrences, graphs, and most importantly, induction—to help you identify gaps in your background knowledge. **You are responsible for filling those gaps.** The course web page has pointers to several excellent online resources for prerequisite material. If you need help, please ask in headbanging, on Piazza, in office hours, or by email.

- ***Each student must submit individual solutions for these homework problems.*** You may use any source at your disposal—paper, electronic, or human—but you ***must*** cite ***every*** source that you use. For all future homeworks, groups of up to three students may submit joint solutions.

- **Submit your solutions on standard printer/copier paper, not notebook paper.** If you write your solutions by hand, please use the last three pages of this homework as a template. At the top of each page, please clearly print your name and NetID, and indicate your registered discussion section. Use both sides of the page. If you plan to typeset your homework, you can find a LaTeX template on the course web site; well-typeset homework will get a small amount of extra credit.

- Submit your solution to each numbered problem (stapled if necessary) in the corresponding drop box outside 1404 Siebel, ***or*** in the corresponding box in Siebel 1404 immediately before/after class. (This is the last homework we'll collect in 1404.) ***Do not staple your entire homework together.***

---

1. Consider the following recursively-defined sets of strings of left brackets $[\![$ and right brackets $]\!]$:

   - A string x is ***balanced*** if it satisfies one of the following conditions:
     - x is the empty string, or
     - $x = [\![\, y \,]\!]\, z$, where y and z are balanced strings.

     For example, the following diagram shows that the string $[\![\,[\![\,]\!]\,[\![\,]\!]\,]\!]\,[\![\,]\!]$ is balanced. Each boxed substring is balanced, and ε is the empty string.

     $$[\![\; \boxed{[\![\, \boxed{\varepsilon} \,]\!]\; \boxed{[\![\, \boxed{\varepsilon} \,]\!]\; \boxed{\varepsilon}}} \;]\!]\; \boxed{[\![\, \boxed{\varepsilon} \,]\!]\; \boxed{\varepsilon}}$$

   - A string x is ***erasable*** if it satisfies one of two conditions:
     - x is the empty string, or
     - $x = y\, [\![\,]\!]\, z$, where yz is an erasable string.

     For example, we can prove that the string $[\![\,[\![\,]\!]\,[\![\,]\!]\,]\!]\,[\![\,]\!]$ is erasable as follows:

     $$[\![\,[\![\,]\!]\,\cancel{[\![\,]\!]}\,]\!]\,[\![\,]\!] \;\to\; [\![\,\cancel{[\![\,]\!]}\,]\!]\,[\![\,]\!] \;\to\; [\![\,]\!]\,\cancel{[\![\,]\!]} \;\to\; \cancel{[\![\,]\!]} \;\to\; \varepsilon$$

   Your task is to prove that these two definitions are equivalent.

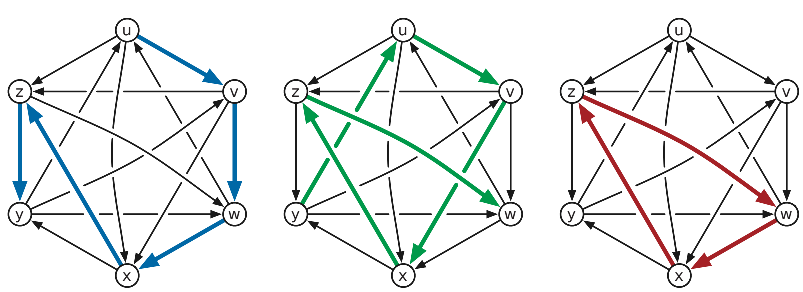
   (a) Prove that every balanced string is erasable.

   (b) Prove that every erasable string is balanced.

2. A ***tournament*** is a directed graph with exactly one directed edge between each pair of vertices. That is, for any vertices v and w, a tournament contains either an edge $v \to w$ or an edge $w \to v$, but not both. A ***Hamiltonian path*** in a directed graph G is a directed path that visits every vertex of G exactly once.

   (a) Prove that every tournament contains a Hamiltonian path.

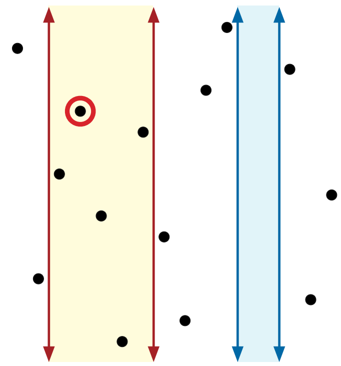
   (b) Prove that every tournament contains either *exactly one* Hamiltonian path or a directed cycle of length three.



A tournament with two Hamiltonian paths $u \to v \to w \to x \to z \to y$ and $y \to u \to v \to x \to z \to w$ and a directed triangle $w \to x \to z \to w$.

3. Suppose you are given a set $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of n points in the plane with distinct x- and y-coordinates. Describe a data structure that can answer the following query as quickly as possible:

   > Given two numbers l and r, find the highest point in P inside the vertical slab $l < x < r$. More formally, find the point $(x_i, y_i) \in P$ such that $l < x_i < r$ and y_i is as large as possible. Return NONE if the slab does not contain any points in P.

   A query with the left slab returns the indicated point.
   A query with the right slab returns NONE.

   To receive full credit, your solution must include (a) a concise description of your data structure, (b) a concise description of your query algorithm, (c) a proof that your query algorithm is correct, (d) a bound on the size of your data structure, and (e) a bound on the running time of your query algorithm. You do ***not*** need to describe or analyze an algorithm to construct your data structure.

   Smaller data structures and faster query times are worth more points.

Starting with this homework, groups of up to three students may submit a single solution for each numbered problem. Every student in the group receives the same grade. Groups can be different for different problems.

---

1. Consider the following cruel and unusual sorting algorithm.

```
CRUEL(A[1..n]):
  if n > 1
      CRUEL(A[1..n/2])
      CRUEL(A[n/2+1..n])
      UNUSUAL(A[1..n])
```

```
UNUSUAL(A[1..n]):
  if n = 2
      if A[1] > A[2]                  ⟨⟨the only comparison!⟩⟩
          swap A[1] ⟷ A[2]
  else
      for i ← 1 to n/4                ⟨⟨swap 2nd and 3rd quarters⟩⟩
          swap A[i+n/4] ⟷ A[i+n/2]
      UNUSUAL(A[1..n/2])              ⟨⟨recurse on left half⟩⟩
      UNUSUAL(A[n/2+1..n])            ⟨⟨recurse on right half⟩⟩
      UNUSUAL(A[n/4+1..3n/4])         ⟨⟨recurse on middle half⟩⟩
```

Notice that the comparisons performed by the algorithm do not depend at all on the values in the input array; such a sorting algorithm is called ***oblivious***. Assume for this problem that the input size n is always a power of 2.

(a) Prove that CRUEL correctly sorts any input array. *[Hint: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is considering this special case enough? What does* UNUSUAL *actually do?]*

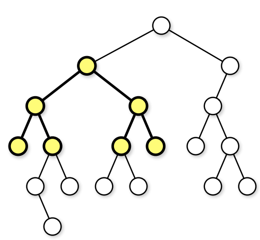
(b) Prove that CRUEL would *not* always sort correctly if we removed the for-loop from UNUSUAL.

(c) Prove that CRUEL would *not* always sort correctly if we swapped the last two lines of UNUSUAL.

(d) What is the running time of UNUSUAL? Justify your answer.

(e) What is the running time of CRUEL? Justify your answer.

2. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

3. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$. Describe an algorithm to find the median of the union of A and B in $O(\log n)$ time. Assume the arrays contain no duplicate elements.

   (b) Now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$. Describe an algorithm to find the median element of $A \cup B \cup C$ in $O(\log n)$ time.

*4. ***Extra credit; due September 17.*** (The "I don't know" rule does not apply to extra credit problems.)

Bob Ratenbur, a new student in CS 225, is trying to write code to perform preorder, inorder, and postorder traversal of binary trees. Bob understands the basic idea behind the traversal algorithms, but whenever he tries to implement them, he keeps mixing up the recursive calls. Five minutes before the deadline, Bob submitted code with the following structure:

```
PREORDER(v):
  if v = NULL
      return
  else
      print label(v)
      [redacted]ORDER(left(v))
      [redacted]ORDER(right(v))
```

```
INORDER(v):
  if v = NULL
      return
  else
      [redacted]ORDER(left(v))
      print label(v)
      [redacted]ORDER(right(v))
```

```
POSTORDER(v):
  if v = NULL
      return
  else
      [redacted]ORDER(left(v))
      [redacted]ORDER(right(v))
      print label(v)
```

Each [redacted] represents either PRE, IN, or POST. Moreover, each of the following function calls appears exactly once in Bob's submitted code:

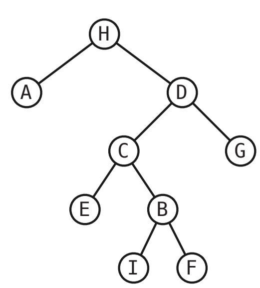
PREORDER(*left*(v))	INORDER(*left*(v))	POSTORDER(*left*(v))
PREORDER(*right*(v))	INORDER(*right*(v))	POSTORDER(*right*(v))

Thus, there are exactly 36 possibilities for Bob's code. Unfortunately, Bob accidentally deleted his source code after submitting the executable, so neither you nor he knows which functions were called where.

Your task is to reconstruct a binary tree T from the output of Bob's traversal algorithms, which has been helpfully parsed into three arrays $Pre[1..n]$, $In[1..n]$, and $Post[1..n]$. Your algorithm should return the unknown tree T. You may assume that the vertex labels of the unknown tree are distinct, and that every internal node has exactly two children. For example, given the input

$$Pre[1..n] = [\mathtt{H\ A\ E\ C\ B\ I\ F\ G\ D}]$$
$$In[1..n] = [\mathtt{A\ H\ D\ C\ E\ I\ F\ B\ G}]$$
$$Post[1..n] = [\mathtt{A\ E\ I\ B\ F\ C\ D\ G\ H}]$$

your algorithm should return the following tree:



In general, the traversal sequences may not give you enough information to reconstruct Bob's code; however, to produce the example sequences above, Bob's code must look like this:

```
PreOrder(v):
  if v = Null
      return
  else
      print label(v)
      PreOrder(left(v))
      PostOrder(right(v))
```

```
InOrder(v):
  if v = Null
      return
  else
      PostOrder(left(v))
      print label(v)
      PreOrder(right(v))
```

```
PostOrder(v):
  if v = Null
      return
  else
      InOrder(left(v))
      InOrder(right(v))
      print label(v)
```

1. Suppose we are given an array $A[1..n]$ of integers, some positive and negative, which we are asked to partition into contiguous subarrays, which we call ***chunks***. The *value* of any chunk is the *square* of the sum of elements in that chunk; the value of a partition of A is the sum of the values of its chunks.

   For example, suppose $A = [3, -1, 4, -1, 5, -9]$. The partition $[3, -1, 4], [-1, 5], [-9]$ has three chunks with total value $(3-1+4)^2 + (-1+5)^2 + (-9)^2 = 6^2 + 4^2 + 9^2 = 133$, while the partition $[3, -1], [4, -1, 5, -9]$ has two chunks with total value $(3-1)^2 + (4-1+5-9)^2 = 5$.

   (a) Describe and analyze an algorithm that computes the minimum-value partition of a given array of n numbers.

   (b) Now suppose we also given an integer $k > 0$. Describe and analyze an algorithm that computes the minimum-value partition of a given array of n numbers ***into at most k chunks***.

2. Consider the following solitaire form of Scrabble. We begin with a fixed, finite sequence of tiles; each tile contains a letter and a numerical value. At the start of the game, we draw the seven tiles from the sequence and put them into our hand. In each turn, we form an English word from some or all of the tiles in our hand, place those tiles on the table, and receive the total value of those tiles as points. If no English word can be formed from the tiles in our hand, the game immediately ends. Then we repeatedly draw the next tile from the start of the sequence until either (a) we have seven tiles in our hand, or (b) the sequence is empty. (Sorry, no double/triple word/letter scores, bingos, blanks, or passing.) Our goal is to obtain as many points as possible.

   For example, suppose we are given the tile sequence

   $\boxed{I_2}\boxed{N_2}\boxed{X_8}\boxed{A_1}\boxed{N_2}\boxed{A_1}\boxed{D_3}\boxed{U_5}\boxed{D_3}\boxed{I_2}\boxed{D_3}\boxed{K_8}\boxed{U_5}\boxed{B_4}\boxed{L_2}\boxed{A_1}\boxed{K_8}\boxed{H_5}\boxed{A_1}\boxed{N_2}$.

   Then we can earn 68 points as follows:

   - We initially draw $\boxed{I_2}\boxed{N_2}\boxed{X_8}\boxed{A_1}\boxed{N_2}\boxed{A_1}\boxed{D_3}$.
   - Play the word $\boxed{N_2}\boxed{A_1}\boxed{I_2}\boxed{A_1}\boxed{D_3}$ for 9 points, leaving $\boxed{N_2}\boxed{X_8}$ in our hand.
   - Draw the next five tiles $\boxed{U_5}\boxed{D_3}\boxed{I_2}\boxed{D_3}\boxed{K_8}$.
   - Play the word $\boxed{U_5}\boxed{N_2}\boxed{D_3}\boxed{I_2}\boxed{D_3}$ for 15 points, leaving $\boxed{K_8}\boxed{X_8}$ in our hand.
   - Draw the next five tiles $\boxed{U_5}\boxed{B_4}\boxed{L_2}\boxed{A_1}\boxed{K_8}$.
   - Play the word $\boxed{B_4}\boxed{U_5}\boxed{L_2}\boxed{K_8}$ for 19 points, leaving $\boxed{K_8}\boxed{X_8}\boxed{A_1}$ in our hand.
   - Draw the next three tiles $\boxed{H_5}\boxed{A_1}\boxed{N_2}$, emptying the list.
   - Play the word $\boxed{A_1}\boxed{N_2}\boxed{K_8}\boxed{H_5}$ for 16 points, leaving $\boxed{X_8}\boxed{A_1}$ in our hand.
   - Play the word $\boxed{A_1}\boxed{X_8}$ for 9 points, emptying our hand and ending the game.

   Design and analyze an algorithm to compute the maximum number of points that can be earned from a given sequence of tiles. ***The input consists of two arrays*** Letter$[1..n]$***, containing a sequence of letters between A and Z, and*** Value$[A..Z]$***, where*** Value$[i]$ ***is the value of letter i.*** The output is a single number. Assume that you can find all English words that can be made from any seven tiles, along with the point values of those words, in $O(1)$ time.

3. ***Extra credit.*** Submit your answer to Homework 1 problem 4.

1. A standard method to improve the cache performance of search trees is to pack more search keys and subtrees into each node. A ***B-tree*** is a rooted tree in which each internal node stores up to B keys and pointers to up to $B+1$ children, each the root of a smaller B-tree. Specifically each node v stores three fields:

   - a positive integer $v.d \le B$,
   - a *sorted* array $v.key[1\,..\,v.d]$, and
   - an array $v.child[0\,..\,v.d]$ of child pointers.

   In particular, the number of child pointers is always exactly one more than the number of keys.

   Each pointer $v.child[i]$ is either NULL or a pointer to the root of a B-tree whose keys are all larger than $v.key[i]$ and smaller than $v.key[i+1]$. In particular, all keys in the leftmost subtree $v.child[0]$ are smaller than $v.key[1]$, and all keys in the rightmost subtree $v.child[v.d]$ are larger than $v.key[v.d]$.

   Intuitively, you should have the following picture in mind:

$$\begin{array}{ccccccccc} [\ \bullet & < key[1] < & \bullet & < key[2] < & \bullet & \cdots & \bullet & < key[d] < & \bullet\ ] \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\ T_0 & & T_1 & & T_2 & \cdots & T_{d-1} & & T_d \end{array}$$

   Here T_i is the subtree pointed to by $child[i]$.

   The ***cost*** of searching for a key x in a B-tree is the number of nodes in the path from the root to the node containing x as one of its keys. A 1-tree is just a standard binary search tree.

   Fix an arbitrary positive integer $B > 0$. (I suggest $B = 8$.) Suppose we are given a sorted array $A[1,\ldots,n]$ of search keys and a corresponding array $F[1,\ldots,n]$ of frequency counts, where $F[i]$ is the number of times that we will search for $A[i]$.

   Describe and analyze an efficient algorithm to find a B-tree that minimizes the total cost of searching for n keys with a given array of frequencies.

   - For 5 points, describe a polynomial-time algorithm for the special case $B = 2$.
   - For 10 points, describe an algorithm for arbitrary B that runs in $O(n^{B+c})$ time for some fixed integer c.
   - For 15 points, describe an algorithm for arbitrary B that runs in $O(n^c)$ time for some fixed integer c that does *not* depend on B.

   Like all other homework problems, 10 points is full credit; any points above 10 will be awarded as extra credit.

   A few comments about B-trees. Normally, B-trees are required to satisfy two additional constraints, which guarantee a worst-case search cost of $O(\log_B n)$: Every leaf must have exactly the same depth, and every node except possibly the root must contain at least $B/2$ keys. However, in this problem, we are not interested in optimizing the *worst-case* search cost, but rather the *total* cost of a sequence of searches, so we will not impose these additional constraints.

   In most large database systems, the parameter B is chosen so that each node exactly fits in a cache line. Since the entire cache line is loaded into cache anyway, and the cost of loading a cache line exceeds the cost of searching within the cache, the running time is dominated by the number of cache faults. This effect is even more noticeable if the data is too big to lie in RAM at all; then the cost is dominated by the number of page faults, and B should be roughly the size of a page.

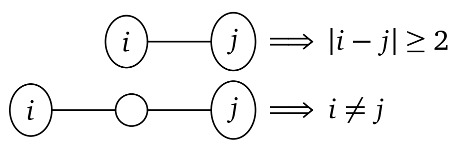
Finally, don't worry about the cache/disk performance in your homework solutions; just analyze the CPU time as usual. Designing algorithms with few cache misses or page faults is a interesting pastime; simultaneously optimizing CPU time *and* cache misses *and* page faults is even more interesting. But this kind of design and analysis requires tools we won't see in this class.

2. ***Extra credit***, because we screwed up the first version.

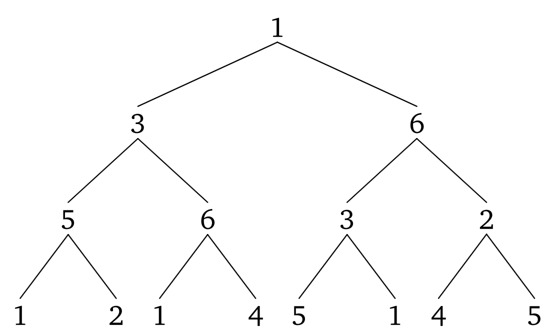
A ***k*-2 coloring** of a tree assigns each vertex a value from the set $\{1, 2, \ldots, k\}$, called its *color*, such that the following constraints are satisfied:

- A node and its parent cannot have the same color or adjacent colors.
- A node and its grandparent cannot have the same color.
- Two nodes with the same parent cannot have the same color.

The last two rules can be written more simply as "Two nodes that are two edges apart cannot have the same color." Diagrammatically, if we write the names of the colors inside the vertices,



For example, here is a valid 6-2 coloring of the complete binary tree with depth 3:



(a) Describe and analyze an algorithm that computes a 6-2 coloring of a given binary tree. The existence of such an algorithm proves that every binary tree has a 6-2 coloring.

(b) Prove that not every binary tree has a 5-2 coloring.

(c) A *ternary* tree is a rooted tree where every node has at most *three* children. What is the smallest integer k such that every ternary tree has a k-2 coloring? Prove your answer is correct.

1. A *meldable priority queue* stores a set of values, called *priorities*, from some totally-ordered universe (such as the integers) and supports the following operations:

   - MakeQueue: Return a new priority queue containing the empty set.
   - FindMin(Q): Return the smallest element of Q (if any).
   - DeleteMin(Q): Remove the smallest element in Q (if any).
   - Insert(Q, x): Insert priority x into Q, if it is not already there.
   - Decrease(Q, x, y): Replace some element $x \in Q$ with a smaller priority y. (If $y > x$, the operation fails.) The input is a pointer directly to the node in Q containing x.
   - Delete(Q, x): Delete the priority $x \in Q$. The input is a pointer directly to the node in Q containing x.
   - Meld(Q_1, Q_2): Return a new priority queue containing all the elements of Q_1 and Q_2; this operation destroys Q_1 and Q_2.

   A simple way to implement such a data structure is to use a ***heap-ordered binary tree*** — each node stores a priority, which is smaller than the priorities of its children, along with pointers to its parent and at most two children. Meld can be implemented using the following randomized algorithm:

   ```
   MELD(Q1, Q2):
     if Q1 is empty return Q2
     if Q2 is empty return Q1

     if priority(Q1) > priority(Q2)
         swap Q1 ↔ Q2

     with probability 1/2
         left(Q1) ← MELD(left(Q1), Q2)
     else
         right(Q1) ← MELD(right(Q1), Q2)

     return Q1
   ```

   (a) Prove that for *any* heap-ordered binary trees Q_1 and Q_2 (***not*** just those constructed by the operations listed above), the expected running time of Meld(Q_1, Q_2) is $O(\log n)$, where $n = |Q_1| + |Q_2|$. *[Hint: How long is a random root-to-leaf path in an n-node binary tree if each left/right choice is made uniformly and independently at random?]*

   (b) Show that each of the other meldable priority queue operations can be implemented with at most one call to Meld and $O(1)$ additional time. (This implies that every operation takes $O(\log n)$ expected time.)

2. Recall that a *priority search tree* is a binary tree in which every node has both a *search key* and a *priority*, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A *treap* is a priority search tree whose search keys are given by the user and whose priorities are independent random numbers.

   A ***heater*** is a priority search tree whose *priorities* are given by the user and whose *search keys* are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is a sort of anti-treap.[1]

[1]There are those who think that life has nothing left to chance, a host of holy horrors to direct our aimless dance.

The following problems consider an n-node heater T. We identify nodes in T by their *priority rank*; for example, "node 5" means the node in T with the 5th smallest priority. The min-heap property implies that node 1 is the root of T. You may assume all search keys and priorities are distinct. Finally, let i and j be arbitrary integers with $1 \le i < j \le n$.

(a) Prove that if we permute the set $\{1, 2, \ldots, n\}$ uniformly at random, integers i and j are adjacent with probability $2/n$.

(b) Prove that node i is an ancestor of node j with probability $2/(i+1)$. *[Hint: Use part (a)!]*

(c) What is the probability that node i is a *descendant* of node j? *[Hint: Don't use part (a)!]*

(d) What is the *exact* expected depth of node j?

(e) Describe and analyze an algorithm to insert a new item into an n-node heater.

(f) Describe and analyze an algorithm to delete the smallest priority (the root) from an n-node heater.

*3. ***Extra credit; due October 15.*** In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval $[0, 1]$. In practice, however, computers have access only to random *bits*. This problem asks you to analyze an implementation of treaps that takes this limitation into account.

Suppose the priority of a node v is abstractly represented as an infinite sequence $\pi_v[1..\infty]$ of random bits, which is interpreted as the rational number

$$priority(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

However, only a finite number ℓ_v of these bits are actually known at any given time. When a node v is first created, *none* of the priority bits are known: $\ell_v = 0$. We generate (or "reveal") new random bits only when they are necessary to compare priorities. The following algorithm compares the priorities of any two nodes in $O(1)$ expected time:

```
LargerPriority(v, w):
  for i ← 1 to ∞
      if i > ℓ_v
          ℓ_v ← i;  π_v[i] ← RandomBit
      if i > ℓ_w
          ℓ_w ← i;  π_w[i] ← RandomBit
      if π_v[i] > π_w[i]
          return v
      else if π_v[i] < π_w[i]
          return w
```

Suppose we insert n items one at a time into an initially empty treap. Let $L = \sum_v \ell_v$ denote the total number of random bits generated by calls to LargerPriority during these insertions.

(a) Prove that $E[L] = \Theta(n)$.

(b) Prove that $E[\ell_v] = \Theta(1)$ for any node v. *[Hint: This is equivalent to part (a). Why?]*

(c) Prove that $E[\ell_{\text{root}}] = \Theta(\log n)$. *[Hint: Why doesn't this contradict part (b)?]*

1. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations.

   - PUSH(x): Add item x to the end of the sequence.
   - PULL(): Remove and return the item at the beginning of the sequence.
   - SIZE(): Return the current number of items in the sequence.

   It is easy to implement a queue using a doubly-linked list, so that it uses $O(n)$ space (where n is the number of items in the queue) and the worst-case time for each of these operations is $O(1)$.

   Consider the following new operation, which removes every tenth element from the queue, starting at the beginning, in $\Theta(n)$ worst-case time.

```
DECIMATE():
  n ← SIZE()
  for i ← 0 to n − 1
      if i mod 10 = 0
          PULL()    ⟨⟨result discarded⟩⟩
      else
          PUSH(PULL())
```

   Prove that in any intermixed sequence of PUSH, PULL, and DECIMATE operations, the amortized cost of each operation is $O(1)$.

2. ***This problem is extra credit***, because the original problem statement had several confusing small errors. I believe these erors are corrected in the current revision.

   Deleting an item from an open-addressed hash table is not as straightforward as deleting from a chained hash table. The obvious method for deleting an item x simply empties the entry in the hash table that contains x. Unfortunately, the obvious method doesn't always work. (Part (a) of this question asks you to prove this.)

   Knuth proposed the following *lazy* deletion strategy. Every cell in the table stores both an *item* and a *label*; the possible labels are **EMPTY**, **FULL**, and **JUNK**. The DELETE operation marks cells as **JUNK** instead of actually erasing their contents. Then FIND pretends that **JUNK** cells are occupied, and INSERT pretends that **JUNK** cells are actually empty. In more detail:

```
FIND(H, x):
  for i ← 0 to m − 1
      j ← h_i(x)
      if H.label[j] = FULL and H.item[j] = x
          return j
      else if H.label[j] = EMPTY
          return NONE
```

```
INSERT(H, x):
  for i ← 0 to m − 1
      j ← h_i(x)
      if H.label[j] = FULL and H.item[j] = x
          return         ⟨⟨already there⟩⟩
      if H.label[j] ≠ FULL
          H.item[j] ← x
          H.label[j] ← FULL
          return
```

```
DELETE(H, x):
  j ← FIND(H, x)
  if j ≠ NONE
      H.label[j] ← JUNK
```

Lazy deletion is always *correct*, but it is only *efficient* if we don't perform too many deletions. The search time depends on the fraction of non-**EMPTY** cells, not on the number of actual items stored in the table; thus, even if the number of items stays small, the table may fill up with **JUNK** cells, causing unsuccessful searches to scan the entire table. Less significantly, the data structure may use significantly more space than necessary for the number of items it actually stores. To avoid both of these issues, we use the following rebuilding rules:

- After each INSERT operation, if less than $1/4$ of the cells are **EMPTY**, rebuild the hash table.
- After each DELETE operation, if less than $1/4$ of the cells are **FULL**, rebuild the hash table.

To rebuild the hash table, we allocate a new hash table whose size is twice the number of **FULL** cells (unless that number is smaller than some fixed constant), INSERT each item in a **FULL** cell in the old hash table into the new hash table, and then discard the old hash table, as follows:

```
REBUILD(H):
  count ← 0
  for j ← 0 to H.size − 1
      if H.label[j] = FULL
          count ← count + 1
  H′ ← new hash table of size max{2 · count, 32}
  for j ← 0 to H.size − 1
      if H.label[j] = FULL
          INSERT(H′, H.item[j])
  discard H
  return H′
```

Finally, here are your actual homework questions!

(a) Describe a ***small*** example where the "obvious" deletion algorithm is incorrect; that is, show that the hash table can reach a state where a search can return the wrong result. Assume collisions are resolved by linear probing.

(b) Suppose we use Knuth's lazy deletion strategy instead. Prove that after several INSERT and DELETE operations into a table of arbitrary size m, it is possible for a single item x to be stored in *almost half* of the table cells. (However, at most one of those cells can be labeled **FULL**.)

(c) For purposes of analysis,[1] suppose FIND and INSERT run in $O(1)$ time when at least $1/4$ of the table cells are **EMPTY**. Prove that in any intermixed sequence of INSERT and DELETE operations, using Knuth's lazy deletion strategy, the amortized time per operation is $O(1)$.

*3. ***Extra credit.*** Submit your answer to Homework 4 problem 3.

[1]In fact, FIND and INSERT run in $O(1)$ *expected* time when at least $1/4$ of the table cells are **EMPTY**, and therefore each INSERT and DELETE takes $O(1)$ *expected* amortized time. But probability doesn't play any role whatsoever in the amortized analysis, so we can safely ignore the word "expected".

1. Suppose we want to maintain an array $X[1..n]$ of bits, which are all initially subject to the following operations.

   - LOOKUP(i): Given an index i, return $X[i]$.
   - BLACKEN(i): Given an index $i < n$, set $X[i] \leftarrow 1$.
   - NEXTWHITE(i): Given an index i, return the smallest index $j \geq i$ such that $X[j] = 0$. (Because we never change $X[n]$, such an index always exists.)

   If we use the array $X[1..n]$, it is trivial to implement LOOKUP and BLACKEN in $O(1)$ time and NEXTWHITE in $O(n)$ time. But you can do better! Describe data structures that support LOOKUP in $O(1)$ worst-case time and the other two operations in the following time bounds. (We want a different data structure for each set of time bounds, not one data structure that satisfies all bounds simultaneously!)

   (a) The worst-case time for both BLACKEN and NEXTWHITE is $O(\log n)$.

   (b) The amortized time for both BLACKEN and NEXTWHITE is $O(\log n)$. In addition, the *worst-case* time for BLACKEN is $O(1)$.

   (c) The amortized time for BLACKEN is $O(\log n)$, and the worst-case time for NEXTWHITE is $O(1)$.

   (d) The worst-case time for BLACKEN is $O(1)$, and the amortized time for NEXTWHITE is $O(\alpha(n))$. *[Hint: There is no* WHITEN.*]*

2. Recall that a standard (FIFO) queue maintains a sequence of items subject to the following operations:

   - PUSH(x): Add item x to the back of the queue (the end of the sequence).
   - PULL(): Remove and return the item at the front of the queue (the beginning of the sequence).

   It is easy to implement a queue using a doubly-linked list and a counter, using $O(n)$ space altogether, so that each PUSH or PULL requires $O(1)$ time.

   (a) Now suppose we want to support the following operation instead of PULL:

   - MULTIPULL(k): Remove the first k items from the front of the queue, and return the kth item removed.

   Suppose further that we implement MULTIPULL using the obvious algorithm:

   ```
   MULTIPULL(k):
     for i ← 1 to k
         x ← PULL()
     return x
   ```

   Prove that in any intermixed sequence of PUSH and MULTIPULL operations, starting with an empty queue, the amortized cost of each operation is $O(1)$. You may assume that k is never larger than the number of items in the queue.

   (b) Now suppose we ***also*** want to support the following operation instead of PUSH:

   - MULTIPUSH(x, k): Insert k copies of x into the back of the queue.

   Suppose further that we implement MULTIPUSH using the obvious algorithm:

<u>MULTIPUSH(k, x):</u>
for $i \leftarrow 1$ to k
 PUSH(x)

Prove that for any integers ℓ and n, there is a sequence of ℓ MULTIPUSH and MULTIPULL operations that require $\Omega(n\ell)$ time, where n is the maximum number of items in the queue at any time. Such a sequence implies that the amortized cost of each operation is $\Omega(n)$.

(c) Finally, describe a data structure that supports arbitrary intermixed sequences of MULTIPUSH and MULTIPULL operations in $O(1)$ amortized cost per operation. Like a standard queue, your data structure must use only $O(1)$ space per item. *[Hint: **Don't** use the obvious algorithms!]*

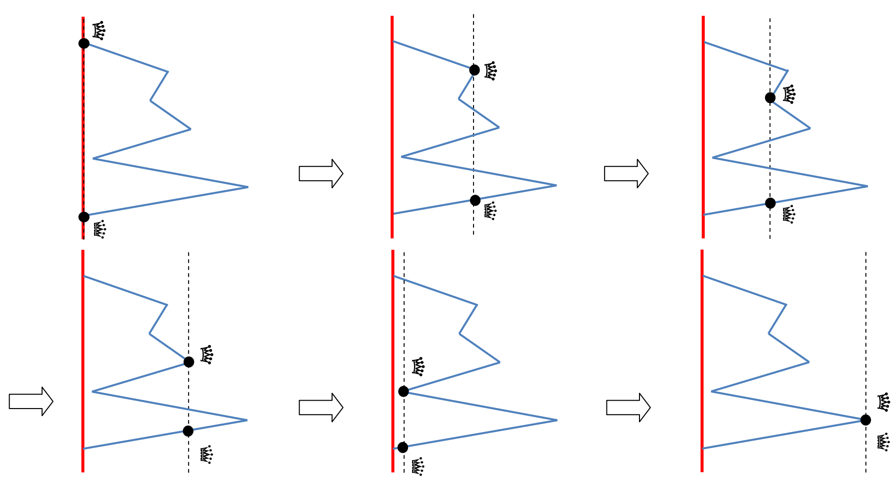
3. In every cheesy romance movie there's always that scene where the romantic couple, physically separated and looking for one another, suddenly matches eyes and then slowly approach one another with unwavering eye contact as the music rolls and in and the rain lifts and the sun shines through the clouds and kittens and puppies....

 Suppose a romantic couple—in grand computer science tradition, named Alice and Bob—enters a park from the northwest and southwest corners of the park, locked in dramatic eye contact. However, they can't just walk to one another in a straight line, because the paths of the park zig-zag between the northwest and southwest entrances. Instead, Alice and Bob must traverse the zig-zagging path so that their eyes are always locked perfectly in vertical eye-contact; thus, their x-coordinates must always be identical.

 We can describe the zigzag path as an array $P[0..n]$ of points, which are the corners of the path in order from the southwest endpoint to the northwest endpoint, satisfying the following conditions:

 - $P[i].y > P[i-1].y$ for every index i. That is, the path always moves upward.
 - $P[0].x = P[n].x = 0$, and $P[i].x > 0$ for every index $1 \le i \le n-1$. Thus, the ends of the path are further to the left then any other point on the path.

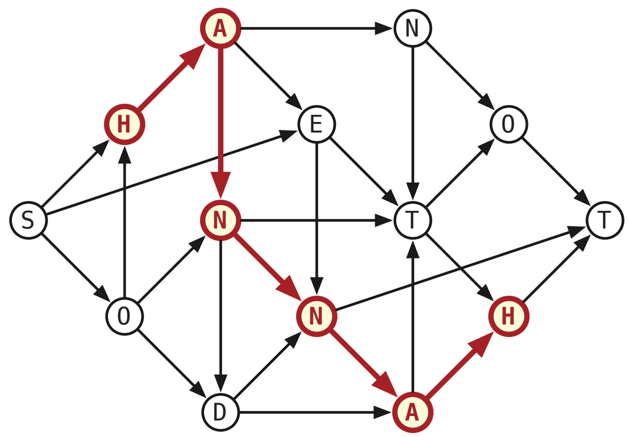
 Prove that Alice and Bob can *always* meet.[1] *[Hint: Describe a graph that models all possible locations of the couple along the path. What are the vertices of this graph? What are the edges? What can we say about the degrees of the vertices?]*



[1] It follows that every cheesy romance movie (that reaches this scene) must have a happy, sappy ending.

1. Suppose we are given a directed acyclic graph G with labeled vertices. Every path in G has a label, which is a string obtained by concatenating the labels of its vertices in order. Recall that a *palindrome* is a string that is equal to its reversal.

   Describe and analyze an algorithm to find the length of the longest palindrome that is the label of a path in G. For example, given the graph below, your algorithm should return the integer 6, which is the length of the palindrome **HANNAH**.



2. Let G be a connected directed graph that contains both directions of every edge; that is, if $u \to v$ is an edge in G, its reversal $v \to u$ is also an edge in G. Consider the following non-standard traversal algorithm.

```
SpaghettiTraversal(G):
    for all vertices v in G
        unmark v
    for all edges u→v in G
        color u→v white
    s ← any vertex in G
    Spaghetti(s)
```

```
Spaghetti(v):
    mark v                              ⟨⟨"visit v"⟩⟩
    if there is a white arc v→w
        if w is unmarked
            color w→v green
        color v→w red                   ⟨⟨"traverse v→w"⟩⟩
        Spaghetti(w)
    else if there is a green arc v→w
        color v→w red                   ⟨⟨"traverse v→w"⟩⟩
        Spaghetti(w)
    ⟨⟨else every arc v→w is red, so halt⟩⟩
```

   We informally say that this algorithm "visits" vertex v every time it marks v, and it "traverses" edge $v \to w$ when it colors that edge red. Unlike our standard graph-traversal algorithms, Spaghetti may (in fact, *will*) mark/visit each vertex more than once.

   The following series of exercises leads to a proof that Spaghetti traverses each directed edge of G exactly once. Most of the solutions are very short.

   (a) Prove that no directed edge in G is traversed more than once.

   (b) When the algorithm visits a vertex v for the kth time, exactly how many edges into v are red, and exactly how many edges out of v are red? *[Hint: Consider the starting vertex s separately from the other vertices.]*

(c) Prove each vertex v is visited at most $\deg(v)$ times, except the starting vertex s, which is visited at most $\deg(s)+1$ times. This claim immediately implies that SPAGHETTITRAVERSAL(G) terminates.

(d) Prove that when SPAGHETTITRAVERSAL(G) ends, the last visited vertex is the starting vertex s.

(e) For every vertex v that SPAGHETTITRAVERSAL(G) visits, prove that all edges incident to v (either in or out) are red when SPAGHETTITRAVERSAL(G) halts. *[Hint: Consider the vertices in the order that they are marked for the first time, starting with s, and prove the claim by induction.]*

(f) Prove that SPAGHETTITRAVERSAL(G) visits every vertex of G.

(g) Finally, prove that SPAGHETTITRAVERSAL(G) traverses every edge of G exactly once.

1. Let G be a directed graph with (possibly negative!) edge weights, and let s be an arbitrary vertex of G. Suppose every vertex $v \neq s$ stores a pointer $pred(v)$ to another vertex in G.

   Describe and analyze an algorithm to determine whether these predecessor pointers define a single-source shortest path tree rooted at s. Do ***not*** assume that the graph G has no negative cycles.

   [Hint: There is a similar problem in head-banging, where you're given distances instead of predecessor pointers.]

2. Let G be a directed graph with positive edge weights, and let s and t be an arbitrary vertices of G. Describe an algorithm to determine the *number* of different shortest paths in G from s to t. Assume that you can perform arbitrary arithmetic operations in $O(1)$ time. *[Hint: Which edges of G belong to shortest paths from s to t?]*

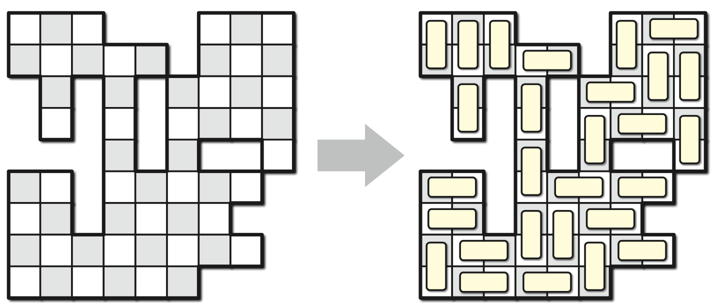
3. Describe and analyze and algorithm to find the second smallest spanning tree of a given undirected graph G with weighted edges, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree.

1. You're organizing the First Annual UIUC Computer Science 72-Hour Dance Exchange, to be held all day Friday, Saturday, and Sunday. Several 30-minute sets of music will be played during the event, and a large number of DJs have applied to perform. You need to hire DJs according to the following constraints.

   - Exactly k sets of music must be played each day, and thus $3k$ sets altogether.
   - Each set must be played by a single DJ in a consistent music genre (ambient, bubblegum, dubstep, horrorcore, hyphy, trip-hop, Nitzhonot, Kwaito, J-pop, Nashville country, ...).
   - Each genre must be played at most once per day.
   - Each candidate DJ has given you a list of genres they are willing to play.
   - Each DJ can play at most three sets during the entire event.

   Suppose there are n candidate DJs and g different musical genres available. Describe and analyze an efficient algorithm that either assigns a DJ and a genre to each of the $3k$ sets, or correctly reports that no such assignment is possible.

2. Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether one can tile the board with dominos—each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.



   Your input is a two-dimensional array $Deleted[1..n, 1..n]$ of bits, where $Deleted[i, j] = \textsc{True}$ if and only if the square in row i and column j has been deleted. Your output is a single bit; you do ***not*** have to compute the actual placement of dominos. For example, for the board shown above, your algorithm should return TRUE.

3. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round A* to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. For example:

$$\begin{bmatrix} 1.2 & 3.4 & 2.4 \\ 3.9 & 4.0 & 2.1 \\ 7.9 & 1.6 & 0.5 \end{bmatrix} \longmapsto \begin{bmatrix} 1 & 4 & 2 \\ 4 & 4 & 2 \\ 8 & 1 & 1 \end{bmatrix}$$

   Describe and analyze an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

1. For any integer k, the problem k-COLOR asks whether the vertices of a given graph G can be colored using at most k colors so that neighboring vertices does not have the same color.

   (a) Prove that k-COLOR is NP-hard, for every integer $k \geq 3$.

   (b) Now fix an integer $k \geq 3$. Suppose you are given a magic black box that can determine **in polynomial time** whether an arbitrary graph is k-colorable; the box returns TRUE if the given graph is k-colorable and FALSE otherwise. The input to the magic black box is a graph. Just a graph. Vertices and edges. Nothing else.

   Describe and analyze a **polynomial-time** algorithm that either computes a proper k-coloring of a given graph G or correctly reports that no such coloring exists, using this magic black box as a subroutine.

2. A boolean formula is in *conjunctive normal form* (or *CNF*) if it consists of a *conjuction* (AND) or several *terms*, each of which is the disjunction (OR) of one or more literals. For example, the formula

$$(\overline{x} \vee y \vee \overline{z}) \wedge (y \vee z) \wedge (x \vee \overline{y} \vee \overline{z})$$

   is in conjunctive normal form. The problem ***CNF-SAT*** asks whether a boolean formula in conjunctive normal form is satisfiable. 3SAT is the special case of CNF-SAT where every clause in the input formula must have exactly three literals; it follows immediately that CNF-SAT is NP-hard.

   Symmetrically, a boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (OR) or several *terms*, each of which is the conjunction (AND) of one or more literals. For example, the formula

$$(\overline{x} \wedge y \wedge \overline{z}) \vee (y \wedge z) \vee (x \wedge \overline{y} \wedge \overline{z})$$

   is in disjunctive normal form. The problem DNF-SAT asks whether a boolean formula in disjunctive normal form is satisfiable.

   (a) Describe a polynomial-time algorithm to solve DNF-SAT.

   (b) Describe a reduction from CNF-SAT to DNF-SAT.

   (c) Why do parts (a) and (b) not imply that P=NP?

3. The 42-PARTITION problem asks whether a given set S of n positive integers can be partitioned into subsets A and B (meaning $A \cup B = S$ and $A \cap B = \varnothing$) such that

$$\sum_{a \in A} a = 42 \sum_{b \in B} b$$

   For example, we can 42-partition the set $\{1, 2, 34, 40, 52\}$ into $A = \{34, 40, 52\}$ and $B = \{1, 2\}$, since $\sum A = 126 = 42 \cdot 3$ and $\sum B = 3$. But the set $\{4, 8, 15, 16, 23, 42\}$ cannot be 42-partitioned.

   (a) Prove that 42-PARTITION is NP-hard.

   (b) Let M denote the largest integer in the input set S. Describe an algorithm to solve 42-PARTITION in time polynomial in n and M. For example, your algorithm should return TRUE when $S = \{1, 2, 34, 40, 52\}$ and FALSE when $S = \{4, 8, 15, 16, 23, 42\}$.

   (c) Why do parts (a) and (b) not imply that P=NP?

CS 473: Undergraduate Algorithms, Fall 2013

Headbanging 0: Induction!

August 28 and 29

---

1. Prove that any non-negative integer can be represented as the sum of distinct powers of 2. ("Write it in binary" is not a proof; it's just a restatement of what you have to prove.)

2. Prove that every integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example:

$$42 = 3^4 - 3^3 - 3^2 - 3^1 \qquad 25 = 3^3 - 3^1 + 3^0 \qquad 17 = 3^3 - 3^2 - 3^0$$

3. Recall that a ***full binary tree*** is either an isolated *leaf*, or an *internal node* with a left subtree and a right subtree, each of which is a full binary tree. Equivalently, a binary tree is ***full*** if every internal node has exactly two children. Give at least *three different* proofs of the following fact: *In every full binary tree, the number of leaves is exactly one more than the number of internal nodes.*

Take-home points:

- Induction is recursion. Recursion is induction.
- All induction is strong/structural induction. There is absolutely no point in using a *weak* induction hypothesis. None. Ever.
- To prove that all snarks are boojums, start with an *arbitrary* snark and remove some tentacles. Do not start with a smaller snark and try to add tentacles. Snarks don't like that.
- Every induction proof requires an exhaustive case analysis. Write down the cases. Make sure they're exhaustive.
- Do the most general cases first. Whatever is left over are the base cases.
- The empty set is the best base case.

Khelm is Warsaw. Warsaw is Khelm. Khelm is Warsaw. Zay gezunt!
Warsaw is Khelm. Khelm is Warsaw. Warsaw is Khelm. For gezunt!

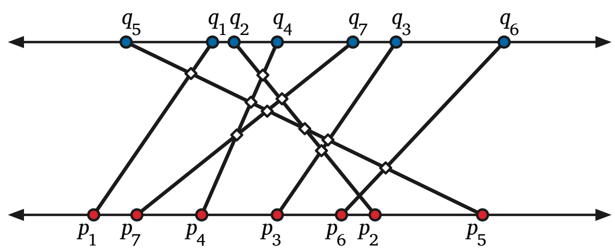
— Golem (feat. Amanda Palmer), "Warsaw is Khelm", *Fresh Off Boat* (2006)

1. An *inversion* in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward).

   Describe and analyze a divide-and-conquer algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. Assume all the elements of the input array are distinct.

2. Suppose you are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Create a set of n line segments by connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time. *[Hint: Use your solution to problem 1.]*

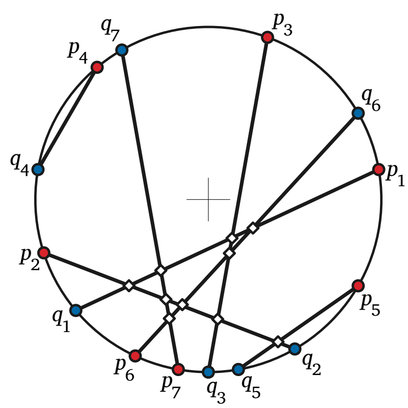
   Assume a reasonable representation for the input points, and assume the x-coordinates of the input points are distinct. For example, for the input shown below, your algorithm should return the number 10.



Ten intersecting pairs of segments with endpoints on parallel lines.

3. Now suppose you are given two sets $\{p_1, p_2, \ldots, p_n\}$ and $\{q_1, q_2, \ldots, q_n\}$ of n points *on the unit circle*. Connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect in $O(n \log^2 n)$ time. *[Hint: Use your solution to problem 2.]*

   Assume a reasonable representation for the input points, and assume all input points are distinct. For example, for the input shown below, your algorithm should return the number 10.



Ten intersecting pairs of segments with endpoints on a circle.

4. **To think about later:** Solve problem 3 in $O(n \log n)$ time.

1. A *longest common subsequence* of a set of strings $\{A_i\}$ is a longest string that is a subsequence of A_i for each i. For example, `alrit` is a longest common subsequence of strings

   `algorithm` and `altruistic`.

   Given two strings $A[1..n]$ and $B[1..n]$, describe and analyze a dynamic programming algorithm that computes the length of a longest common subsequence of the two strings in $O(n^2)$ time.

2. Describe and analyze a dynamic programming algorithm that computes the length of a longest common subsequence of three strings $A[1..n]$, $B[1..n]$, and $C[1..n]$ in $O(n^3)$ time. *[Hint: Try* ***not*** *to use your solution to problem 1 directly.]*

3. A *lucky-10 number* is a string $D[1..n]$ of digits from 1 to 9 (no zeros), such that the i-th digit and the last i-th digit sum up to 10; in another words, $D[i] + D[n - i + 1] = 10$ for all i. For example,

   `3141592648159697` and `11599`

   are both lucky-10 numbers. Given a string of digits $D[1..n]$, describe and analyze a dynamic programming algorithm that computes the length of a longest lucky-10 subsequence of the string. *[Hint: Try to use your solution to problem 1* ***directly****.]*

4. **To think about later:** Can you solve problem 1 in $O(n)$ space?

1. A ***vertex cover*** of a graph is a subset S of the vertices such that every vertex v either belongs to S or has a neighbor in S. In other words, the vertices in S cover all the edges. Finding the minimum size of a vertex cover is NP-hard, but in trees it can be found using dynamic programming.

   Given a tree T and non-negative weight $w(v)$ for each vertex v, describe an algorithm computing the minimum weight of a vertex cover of T.

2. Suppose you are given an unparenthesized mathematical expression containing n numbers, where the only operators are $+$ and $-$; for example:

$$1+3-2-5+1-6+7$$

   You can change the value of the expression by adding parentheses in different positions. For example:

$$1+3-2-5+1-6+7=-1$$
$$(1+3-(2-5))+(1-6)+7=9$$
$$(1+(3-2))-(5+1)-(6+7)=-17$$

   Design an algorithm that, given a list of integers separated by $+$ and $-$ signs, determines the maximum possible value the expression can take by adding parentheses.

   You can only insert parentheses immediately before and immediately after numbers; in particular, you are not allowed to insert implicit multiplication as in $1+3(-2)(-5)+1-6+7=33$.

3. Fix an arbitrary sequence $c_1 < c_2 < \cdots < c_k$ of coin values, all in cents. We have an infinite number of coins of each denomination. Describe a dynamic programming algorithm to determine, given an arbitrary non-negative integer x, the least number of coins whose total value is x. For simplicity, you may assume that $c_1 = 1$.

   To think about later after learning "greedy algorithms":

   (a) Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

   (b) Suppose that the available coins have the values $c^0, c^1, \ldots, c^k$ for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

   (c) Describe a set of 4 coin values for which the greedy algorithm does ***not*** yield an optimal solution.

Note: All the questions in this session are taken from past CS473 midterms.

1. (Fall 2006) **Multiple Choice:** Each of the questions on this page has one of the following five answers: For each question, write the letter that corresponds to your answer.

$\boxed{\text{A: } \Theta(1) \quad \text{B: } \Theta(\log n) \quad \text{C: } \Theta(n) \quad \text{D: } \Theta(n \log n) \quad \text{E: } \Theta(n)}$

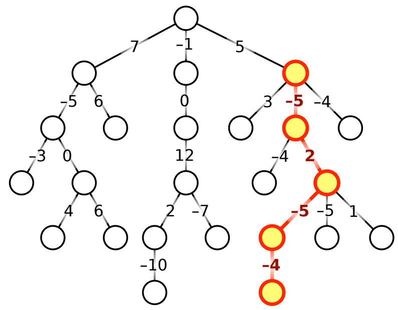
   (a) What is $\frac{5}{n} + \frac{n}{5}$?
   (b) What is $\sum_{i=1}^{n} \frac{n}{i}$?
   (c) What is $\sum_{i=1}^{n} \frac{i}{n}$?
   (d) How many bits are required to represent the nth Fibonacci number in binary?
   (e) What is the solution to the recurrence $T(n) = 2T(n/4) + \Theta(n)$?
   (f) What is the solution to the recurrence $T(n) = 16T(n/4) + \Theta(n)$?
   (g) What is the solution to the recurrence $T(n) = T(n-1) + \frac{1}{n^2}$?
   (h) What is the worst-case time to search for an item in a binary search tree?
   (i) What is the worst-case running time of quicksort?
   (j) What is the running time of the fastest possible algorithm to solve Sudoku puzzles? A Sudoku puzzle consists of a 9×9 grid of squares, partitioned into nine 3×3 sub-grids; some of the squares contain digits between 1 and 9. The goal of the puzzle is to enter digits into the blank squares, so that each digit between 1 and 9 appears exactly once in each row, each column, and each 3×3 sub-grid. The initial conditions guarantee that the solution is unique.

2							4	
	7		5					
				1		9		
6		4			2			
	8						5	
			9			3		7
		1		4				
					3		8	
	5							6

A Sudoku puzzle. **Don't try to solve this during the exam!**

2. (Spring 2010) Let T be a rooted tree with integer weights on its edges, which could be positive, negative, or zero. The weight of a path in T is the sum of the weights of its edges. Describe and analyze an algorithm to compute the minimum weight of any path from a node in T down to one of its descendants. It is not necessary to compute the actual minimum-weight path; just its weight. For example, given the tree shown below, your algorithm should return the number -12.

3. (Fall 2006) Suppose you are given an array $A[1..n]$ of n distinct integers, sorted in increasing order. Describe and analyze an algorithm to determine whether there is an index i such that $A[i] = i$, in $o(n)$ time. [Hint: Yes, that's little-oh of n. What can you say about the sequence $A[i] - i$?]



The minimum-weight downward path in this tree has weight −12.

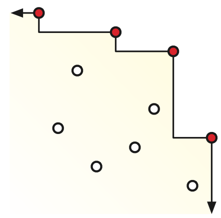
4. (Spring 2010 and Spring 2004) Describe and analyze efficient algorithms to solve the following problems:

   (a) Given a set of n integers, does it contain two elements a, b such that $a + b = 0$?

   (b) Given a set of n integers, does it contain three elements a, b, c such that $a + b = c$?

1. What is the *exact* expected number of leaves in a treap with n nodes?

2. Recall question 5 from Midterm 1:

   > Suppose you are given a set P of n points in the plane. A point $p \in P$ is *maximal* in P if no other point in P is both above and to the right of P. Intuitively, the maximal points define a "staircase" with all the other points of P below it.
   >
   > 
   >
   > A set of ten points, four of which are maximal.
   >
   > Describe and analyze an algorithm to compute the number of maximal points in P in $O(n \log n)$ time. For example, given the ten points shown above, your algorithm should return the integer 4.

   Suppose the points in P are generated independently and uniformly at random in the unit square $[0,1]^2$. What is the *exact* expected number of maximal points in P?

3. Suppose you want to write an app for your new Pebble smart watch that monitors the global Twitter stream and selects a small sample of *random* tweets. You will not know when the stream ends until your app attempts to read the next tweet and receives the error message FAILWHALE. The Pebble has only a small amount of memory, far too little to store the entire stream.

   (a) Describe an algorithm that, as soon as the stream ends, returns a single tweet chosen uniformly at random from the stream. Prove your algorithm is correct. (You may assume that the stream contains at least one tweet.)

   (b) Now fix an arbitrary positive integer k. Describe an algorithm that picks k tweets uniformly at random from the stream. Prove your algorithm is correct. (You may assume that the stream contains at least k tweets.)

Recall the following elementary data structures from CS 225.

- A *stack* supports the following operations.
  - PUSH pushes an element on top of the stack.
  - POP removes the top element from a stack.
  - ISEMPTY checks if a stack is empty.
- A *queue* supports the following operations.
  - PUSH adds an element to the back of the queue.
  - PULL removes an element from the front of the queue.
  - ISEMPTY checks if a queue is empty.
- A *deque*, or double-ended queue, supports the following operations.
  - PUSH adds an element to the back of the queue.
  - PULL removes an element from the back of the queue.
  - CUT adds an element from the front of the queue.
  - POP removes an element from the front of the queue.
  - ISEMPTY checks if a queue is empty.

Suppose you have a stack implementation that supports all stack operations in constant time.

1. Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that each queue operation runs in $O(1)$ amortized time.
2. Describe how to implement a deque using three stacks and $O(1)$ additional memory, so that each deque operation runs in $O(1)$ amortized time.

1. Let P be a set of n points in the plane. Recall from the midterm that the *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.

   (a) Describe and analyze a data structure that stores the staircase of a set of points, and an algorithm $\text{ABOVE?}(x, y)$ that returns TRUE if the point (x, y) is above the staircase, or FALSE otherwise. Your data structure should use $O(n)$ space, and your ABOVE? algorithm should run in $O(\log n)$ time.

   (b) Describe and analyze a data structure that maintains the staircase of a set of points as new points are inserted. Specifically, your data structure should support a function $\text{INSERT}(x, y)$ that adds the point (x, y) to the underlying point set and returns TRUE or FALSE to indicate whether the staircase of the set has changed. Your data structure should use $O(n)$ space, and your INSERT algorithm should run in $O(\log n)$ amortized time.

2. An *ordered stack* is a data structure that stores a sequence of items and supports the following operations.

   - $\text{ORDEREDPUSH}(x)$ removes all items smaller than x from the beginning of the sequence and then adds x to the beginning of the sequence.
   - POP deletes and returns the first item in the sequence (or NULL if the sequence is empty).

   Suppose we implement an ordered stack with a simple linked list, using the obvious ORDEREDPUSH and POP algorithms. Prove that if we start with an empty data structure, the amortized cost of each ORDEREDPUSH or POP operation is $O(1)$.

3. Consider the following solution for the union-find problem, called *union-by-weight*. Each set leader $\bar{x}$ stores the number of elements of its set in the field weight($\bar{x}$). Whenever we UNION two sets, the leader of the *smaller* set becomes a new child of the leader of the *larger* set (breaking ties arbitrarily).

```
MAKESET(x)
  parent(x) ← x
  weight(x) ← 1

FIND(x)
  while x ≠ parent(x)
      x ← parent(x)
  return x

UNION(x, y)
  x̄ ← FIND(x)
  ȳ ← FIND(y)
  if weight(x̄) > weight(ȳ)
      parent(ȳ) ← x̄
      weight(x̄) ← weight(x̄) + weight(ȳ)
  else
      parent(x̄) ← ȳ
      weight(x̄) ← weight(x̄) + weight(ȳ)
```

   Prove that if we use union-by-weight, the *worst-case* running time of $\text{FIND}(x)$ is $O(\log n)$, where n is the cardinality of the set containing x.

1. Let G be an undirected graph.

   (a) Suppose we start with two coins on two arbitrarily chosen nodes. At every step, each coin must move to an adjacent node. Describe an algorithm to compute the minimum number of steps to reach a configuration that two coins are on the same node.

   (b) Now suppose there are three coins, numbered 0, 1, and 2. Again we start with an arbitrary coin placement with all three coins facing up. At each step, we move each coin to an adjacent node at each step. Moreover, for every integer i, we flip coin i mod 3 at the ith step. Describe an algorithm to compute the minimum number of steps to reach a configuration that all three coins are on the same node and all facing up. What is the running time of your algorithm?

2. Let G be a directed acyclic graph with a unique source s and a unique sink t.

   (a) A *Hamiltonian path* in G is a directed path in G that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

   (b) Suppose several nodes in G are marked to be *important*; also an integer k is given. Design an algorithm which computes all the nodes that can reach t through at least k important nodes.

   (c) Suppose the edges in G have real weights. Describe an algorithm to find a path from s to t with maximum total weight.

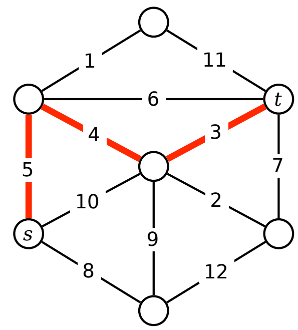
   (d) Suppose the vertices of G have labels from a fixed finite alphabet, and let $A[1..\ell]$ be a string over the same alphabet. Any directed path in G has a label, which is obtained by concatenating the labels of its vertices. Describe an algorithm to find the longest path in G whose labels are a subsequence of A.

3. Let G be a directed graph with a special source that has an edge to each other node in graph, and denote $scc(G)$ as the strong component graph of G. Let S and S' be two strongly connected components in G with $S \to S'$ an arc in $scc(G)$. (That is, if there is an arc between node $u \in S$ and $v \in S'$, then it must be $u \to v$.) Consider a fixed depth-first search performed on G starting at s; we define post$(\cdot)$ as the post-order numbering of the search.

   (a) Prove or disprove that we have $\text{post}(u) > \text{post}(u')$ for any $u \in S$ and $u' \in S'$.

   (b) Prove or disprove that we have $\max_{u \in S} \text{post}(u) > \max_{u' \in S'} \text{post}(u')$.

1. Consider a path between two vertices s and t in an undirected weighted graph G. The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t. (If there are no paths from s to t, the bottleneck distance between s and t is ∞.)

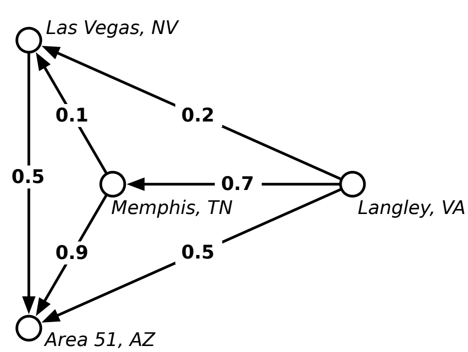


Describe an algorithm to compute the bottleneck distance between *every* pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.

2. Let G be a directed graph with (possibly negative) edge weights, and let s be an arbitrary vertex of G. Suppose for each vertex v we are given a real number $d(v)$. Describe and analyze an algorithm to determine whether the numbers $d(v)$ on vertices are the shortest path distances from s to each vertex v. Do not assume that the graph G has no negative cycles.

3. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What rout should hs take so that he has the least chance of being abducted?

   More formally, you are given a directed graph G, possibly with cycles, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.



For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of begin abducted![1]

[1]That's how they got Elvis, you know.

Almost all these review problems from from past midterms.

1. [Fall 2002, Spring 2004] Suppose we want to maintain a set X of numbers, under the following operations:

   - INSERT(x): Add x to the set (if it isn't already there).
   - PRINT&DELETEBETWEEN(a, b): Print every element $x \in X$ such that $a \leq x \leq b$, in order from smallest to largest, and then delete those elements from X.

   For example, if the current set is $\{1, 5, 3, 4, 8\}$, then PRINT&DELETEBETWEEN($4, 6$) prints the numbers 4 and 5 and changes the set to $\{1, 3, 8\}$.

   Describe and analyze a data structure that supports these two operations, each in $O(\log n)$ *amortized* time, where n is the maximum number of elements in X.

2. [Spring 2004] Consider a random walk on a path with vertices numbered $1, 2, \ldots, n$ from left to right. We start at vertex 1. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex n.

   Prove that the probability that the walk ends by falling off the *left* end of the path is exactly $n/(n+1)$. [Hint: Set up a recurrence and verify that $n/(n+1)$ satisfies it.]

3. [Fall 2006] ***Prove or disprove*** each of the following statements.

   (a) Let G be an arbitrary undirected graph with arbitrary distinct weights on the edges. The minimum spanning tree of G includes the lightest edge in every cycle in G.

   (b) Let G be an arbitrary undirected graph with arbitrary distinct weights on the edges. The minimum spanning tree of G excludes the heaviest edge in every cycle in G.

4. [Fall 2012] Let $G = (V, E)$ be a connected undirected graph. For any vertices u and v, let $d_G(u, v)$ denote the length of the shortest path in G from u to v. For any sets of vertices A and B, let $d_G(A, B)$ denote the length of the shortest path in G from any vertex in A to any vertex in B:

   $$d_G(A, B) = \min_{u \in A} \min_{v \in B} d_G(u, v).$$

   Describe and analyze a fast algorithm to compute $d_G(A, B)$, given the graph G and subsets A and B as input. You do not need to prove that your algorithm is correct.

5. Let G and H be directed acyclic graphs, whose vertices have labels from some fixed alphabet, and let $A[1..\ell]$ be a string over the same alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices.

   (a) Describe an algorithm to find the longest string that is both a label of a directed path in G and the label of a directed path in H.

   (b) Describe an algorithm to find the longest string that is both a *subsequence* of the label of a directed path in G and *subsequence* of the label of a directed path in H.

1. The Island of Sodor is home to a large number of towns and villages, connected by an extensive rail network. Recently, several cases of a deadly contagious disease (either swine flu or zombies; reports are unclear) have been reported in the village of Ffarquhar. The controller of the Sodor railway plans to close down certain railway stations to prevent the disease from spreading to Tidmouth, his home town. No trains can pass through a closed station. To minimize expense (and public notice), he wants to close down as few stations as possible. However, he cannot close the Ffarquhar station, because that would expose him to the disease, and he cannot close the Tidmouth station, because then he couldn't visit his favorite pub.

   Describe and analyze an algorithm to find the minimum number of stations that must be closed to block all rail travel from Ffarquhar to Tidmouth. The Sodor rail network is represented by an undirected graph, with a vertex for each station and an edge for each rail connection between two stations. Two special vertices f and t represent the stations in Ffarquhar and Tidmouth.

2. Given an undirected graph $G = (V, E)$, with three vertices u, v, and w, describe and analyze an algorithm to determine whether there is a path from u to w that passes through v.

3. Suppose you have already computed a maximum flow f^* in a flow network G with *integer* edge capacities.

   (a) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is increased by 1.

   (b) Describe and analyze an algorithm to update the maximum flow after the capacity of a single edge is decreased by 1.

   Both algorithms should be significantly faster than recomputing the maximum flow from scratch.

1. An *American graph* is a directed graph with each vertex colored *red*, *white*, or *blue*. An *American Hamiltonian path* is a Hamiltonian path that cycles between red, white, and blue vertices; that is, every edge goes from red to white, or white to blue, or blue to red. The AMERICANHAMILTONIANPATH problem asks whether there is an American Hamiltonian path in an American graph.

   (a) Prove that AMERICANHAMILTONIANPATH is NP-complete by reducing from HAMILTONIANPATH.

   (b) In the opposite direction, reduce AMERICANHAMILTONIANPATH to HAMILTONIANPATH.

2. Given a graph G, the DEG17SPANNINGTREE problem asks whether G has a spanning tree in which each vertex of the spanning tree has degree at most 17. Prove that DEG17SPANNINGTREE is NP-complete.

3. Two graphs are *isomorphic* if one can be transformed into the other by relabeling the vertices. Consider the following related decision problems:

   - GRAPHISOMORPHISM: Given two graphs G and H, determine whether G and H are isomorphic.
   - EVENGRAPHISOMORPHISM: Given two graphs G and H, such that every vertex of G and H have even degree, determine whether G and H are isomorphic.
   - SUBGRAPHISOMORPHISM: Given two graphs G and H, determine whether G is isomorphic to a subgraph of H.

   (a) Describe a polynomial time reduction from GRAPHISOMORPHISM to EVENGRAPHISOMORPHISM.

   (b) Describe a polynomial time reduction from GRAPHISOMORPHISM to SUBGRAPHISOMORPHISM.

1. Prove that the following problem is NP-hard.

   SETCOVER: Given a collection of sets $\{S_1, \ldots, S_m\}$, find the smallest sub-collection of S_i's that contains all the elements of $\bigcup_i S_i$.

2. Given an undirected graph G and a subset of vertices S, a *Steiner tree* of S in G is a subtree of G that contains every vertex in S. If S contains every vertex of G, a Steiner tree is just a spanning tree; if S contains exactly two vertices, any path between them is a Steiner tree.

   Given a graph G, a vertex subset S, and an integer k, the *Steiner tree problem* requires us to decide whether there is a Steiner tree of S in G with at most k edges. Prove that the Steiner tree problem is NP-hard. *[Hint: Reduce from* VERTEXCOVER*, or* SETCOVER*, or* 3SAT*.]*

3. Let G be a directed graph whose edges are colored red and white. A *Canadian Hamiltonian path* is a Hamiltonian path whose edges are alternately red and white. The CANADIANHAMILTONIANPATH problem ask us to find a Canadian Hamiltonian path in a graph G. (Two weeks ago we looked for Hamiltonian paths that cycled through colors on the *vertices* instead of edges.)

   (a) Prove that CANADIANHAMILTONIANPATH is NP-Complete.

   (b) Reduce CANADIANHAMILTONIANPATH to HAMILTONIANPATH.

This exam lasts 120 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. Each of these ten questions has one of the following five answers:

   A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$

   (a) What is $\dfrac{n^5 - 3n^3 - 5n + 4}{4n^3 - 2n^2 + n - 7}$?

   (b) What is $\displaystyle\sum_{i=1}^{n} i$?

   (c) What is $\displaystyle\sum_{i=1}^{n} \sqrt{\frac{n}{i}}$?

   (d) How many bits are required to write the integer n^{10} in binary?

   (e) What is the solution to the recurrence $E(n) = E(n/2) + E(n/4) + E(n/8) + 16n$?

   (f) What is the solution to the recurrence $F(n) = 6F(n/6) + 6n$?

   (g) What is the solution to the recurrence $G(n) = 9G(\lceil n/3 \rceil + 1) + n$?

   (h) The *total path length* of a binary tree is the sum of the depths of all nodes. What is the total path length of an n-node binary tree in the worst case?

   (i) Consider the following recursive function, defined in terms of a fixed array $X[1\,..\,n]$:

   $$WTF(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \max\left\{ \begin{array}{c} 2 \cdot \big[X[i] \neq X[j]\big] + WTF(i+1, j-1) \\ 1 + WTF(i+1, j) \\ 1 + WTF(i, j-1) \end{array} \right\} & \text{otherwise} \end{cases}$$

   How long does it take to compute $WTF(1,n)$ using dynamic programming?

   (j) Voyager 1 recently became the first made-made object to reach interstellar space. Currently the spacecraft is about 18 billion kilometers (roughly 60,000 light seconds) from Earth, traveling outward at approximately 17 kilometers per second (approximately 1/18000 of the speed of light). Voyager carries a golden record containing over 100 digital images abd approximately one hour of sound recordings. In digital form, the recording would require about 1 gigabyte. Voyager can transmit data back to Earth at approximately 1400 bits per second. Suppose the engineers at JPL sent instructions to Voyager 1 to send the complete contents of the Golden Record back to Earth; how many seconds would they have to wait to receive the entire record?

2. You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates; each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the *same* party or not simply by introducing them to each other—members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

   Suppose more than half of the delegates belong to the same political party. Describe and analyze an efficient algorithm that identifies all members of this majority party.

3. Recall that a *tree* is a connected undirected graph with no cycles. ***Prove*** that in any tree, the number of nodes is exactly one more than the number of edges.

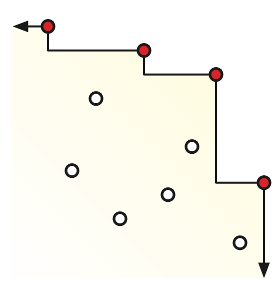
4. Next spring break, you and your friends decide to take a road trip, but before you leave, you decide to figure out *exactly* how much money to bring for gasoline. Suppose you compile a list of all gas stations along your planned route, containing the following information:

   - A sorted array $\mathit{Dist}[0\,..\,n]$, where $\mathit{Dist}[0] = 0$ and $\mathit{Dist}[i]$ is the number of miles from the beginning of your route to the ith gas station. Your route ends at the nth gas station.
   - A second array $\mathit{Price}[1\,..\,n]$, where $\mathit{Price}[i]$ is the price of one gallon of gasoline at the ith gas station. (Unlike in real life, these prices do not change over time.)

   You start the trip with a full tank of gas. Whenever you buy gas, you must completely fill your tank. Your car holds exactly 10 gallons of gas and travels exactly 25 miles per gallon; thus, starting with a full tank, you can travel exactly 250 miles before your car dies. Finally, $\mathit{Dist}[i+1] < \mathit{Dist}[i]+250$ for every index i, so the trip is possible.

   Describe and analyze an algorithm to determine the minimum amount of money you must spend on gasoline to guarantee that you can drive the entire route.

5. Suppose you are given a set P of n points in the plane. A point $p \in P$ is *maximal* in P if no other point in P is both above and to the right of P. Intuitively, the maximal points define a "staircase" with all the other points of P below it.

   A set of ten points, four of which are maximal.

   Describe and analyze an algorithm to compute the number of maximal points in P in $O(n \log n)$ time. For example, given the ten points shown above, your algorithm should return the integer 4.

This exam lasts 120 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. Each of these ten questions has one of the following five answers:

A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$

(a) What is $\dfrac{n^5 - 3n^3 - 5n + 4}{3n^4 - 2n^2 + n - 7}$?

(b) What is $\displaystyle\sum_{i=1}^{n} \frac{n}{i}$?

(c) What is $\displaystyle\sum_{i=1}^{n} \frac{i}{n}$?

(d) How many bits are required to write the integer 10^n in binary?

(e) What is the solution to the recurrence $E(n) = E(n/3) + E(n/4) + E(n/5) + n/6$?

(f) What is the solution to the recurrence $F(n) = 16\,F(n/4 + 2) + n$?

(g) What is the solution to the recurrence $G(n) = G(n/2) + 2G(n/4) + n$?

(h) The *total path length* of a binary tree is the sum of the depths of all nodes. What is the total path length of a perfectly balanced n-node binary tree?

(i) Consider the following recursive function, defined in terms of two fixed arrays $A[1 \mathinner{..} n]$ and $B[1 \mathinner{..} n]$:

$$WTF(i,j) = \begin{cases} 0 & \text{if } i > j \\ \max \left\{ \begin{array}{c} (A[i] - B[j])^2 + WTF(i+1, j-1) \\ A[i]^2 + WTF(i+1, j) \\ B[i]^2 + WTF(i, j-1) \end{array} \right\} & \text{otherwise} \end{cases}$$

How long does it take to compute $WTF(1, n)$ using dynamic programming?

(j) Voyager 1 recently became the first made-made object to reach interstellar space. Currently the spacecraft is about 18 billion kilometers (roughly 60,000 light seconds) from Earth, traveling outward at approximately 17 kilometers per second (approximately 1/18000 of the speed of light). Voyager carries a golden record containing over 100 digital images and approximately one hour of sound recordings. In digital form, the recording would require about 1 gigabyte. Voyager can transmit data back to Earth at approximately 1400 bits per second. Suppose the engineers at JPL sent instructions to Voyager 1 to send the complete contents of the Golden Record back to Earth; how many seconds would they have to wait to receive the entire record?

2. Suppose we are given an array $A[0..n+1]$ with fencepost values $A[0] = A[n+1] = -\infty$. We say that an element $A[x]$ is a *local maximum* if it is less than or equal to its neighbors, or more formally, if $A[x-1] \le A[x]$ and $A[x] \ge A[x+1]$. For example, there are five local maxima in the following array:

$-\infty$	6	7	2	1	3	7	5	4	9	9	3	4	8	6	$-\infty$

We can obviously find a local maximum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that returns the index of one local maximum in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array **must** have at least one local maximum. Why?]*

3. ***Prove*** that in any binary tree, the number of nodes with no children (leaves) is exactly one more than the number of nodes with two children. (Remember that a binary tree can have nodes with only one child.)

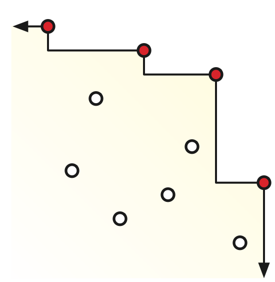
4. A string x is a *supersequence* of a string y if we can obtain x by inserting zero or more letters into y, or equivalently, if y is a subsequence of x. For example, the string **D**YN**AM**IC**P**ROG**RA**MMIN**G** is a supersequence of the string DAMPRAG.

A *palindrome* is any string that is exactly the same as its reversal, like I, DAD, HANNAH, AIBOHPHOBIA (fear of palindromes), or the empty string.

Describe and analyze an algorithm to find the length of the shortest supersequence of a given string that is also a palindrome.

For example, the 11-letter string E**HE**C**ADAC**E**HE** is the shortest palindrome supersequence of HEADACHE, so given the string HEADACHE as input, your algorithm should output the number 11.

5. Suppose you are given a set P of n points in the plane. A point $p \in P$ is *maximal* in P if no other point in P is both above and to the right of P. Intuitively, the maximal points define a "staircase" with all the other points of P below it.



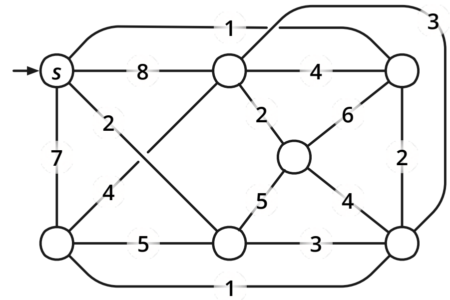
A set of ten points, four of which are maximal.

Describe and analyze an algorithm to compute the number of maximal points in P in $O(n \log n)$ time. For example, given the ten points shown above, your algorithm should return the integer 4.

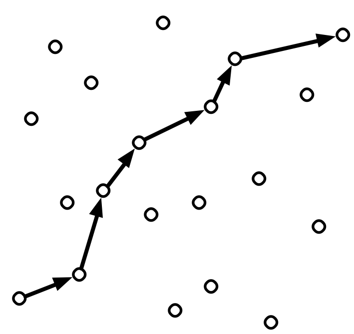
This exam lasts 120 minutes.
Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the following spanning trees in the weighted graph pictured below. Some of these subproblems have more than one correct answer.

   (a) A depth-first spanning tree rooted at s
   (b) A breadth-first spanning tree rooted at s
   (c) A shortest-path tree rooted at s
   (d) A minimum spanning tree
   (e) A *maximum* spanning tree



2. A ***polygonal path*** is a sequence of line segments joined end-to-end; the endpoints of these line segments are called the ***vertices*** of the path. The ***length*** of a polygonal path is the sum of the lengths of its segments. A polygonal path with vertices $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ is ***monotonically increasing*** if $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for every index i—informally, each vertex of the path is above and to the right of its predecessor.



A monotonically increasing polygonal path with seven vertices through a set of points

Suppose you are given a set S of n points in the plane, represented as two arrays $X[1..n]$ and $Y[1..n]$. Describe and analyze an algorithm to compute the length of the maximum-length monotonically increasing path with vertices in S. Assume you have a subroutine $\textsc{Length}(x, y, x', y')$ that returns the length of the segment from (x, y) to (x', y').

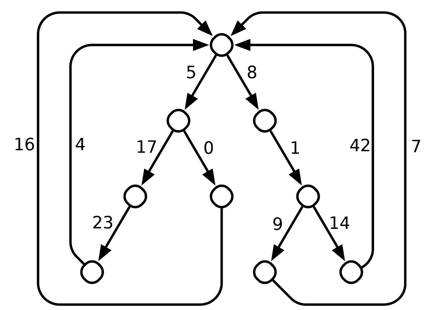
3. Suppose you are maintaining a circular array $X[0..n-1]$ of counters, each taking a value from the set $\{0,1,2\}$. The following algorithm increments one of the counters; if the counter overflows, the algorithm resets it 0 and recursively increments its two neighbors.

```
INCREMENT(i):
  X[i] ← X[i] + 1
  if X[i] = 3
      X[i] ← 0
      INCREMENT((i − 1) mod n)
      INCREMENT((i + 1) mod n)
```

   (a) Suppose $n = 5$ and $X = [2,2,2,2,2]$. What does X contain after we call INCREMENT(3)?

   (b) Suppose all counters are initially 0. ***Prove*** that INCREMENT runs in $O(1)$ amortized time.

4. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has non-negative weight.



A looped tree.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path from an arbitrary vertex s to another arbitrary vertex t, in a looped tree with n vertices?

   (b) Describe and analyze a faster algorithm. Your algorithm should compute the actual shortest path, not just its length.

5. Consider the following algorithm for finding the smallest element in an unsorted array:

```
RANDOMMIN(A[1..n]):
  min ← ∞
  for i ← 1 to n in random order
      if A[i] < min
          min ← A[i]    (⋆)
  return min
```

   Assume the elements of A are all distinct.

   (a) In the worst case, how many times does RANDOMMIN execute line $(\star)$?

   (b) What is the probability that line $(\star)$ is executed during the *last* iteration of the for loop?

   (c) What is the *exact* expected number of executions of line $(\star)$?

This exam lasts 180 minutes.
Write your answers in the separate answer booklet.
Please return this question handout and your cheat sheets with your answers.

1. Suppose you are given a sorted array of n distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, you are given an array $A[1..n]$ such that the prefix $A[1..k]$ is sorted in increasing order, the suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$. For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	−4	0	2	5	7	8

Describe and analyze an algorithm to determine if the given array contains a given number x. For example, given the previous array and the number 17 as input, your algorithm should return FALSE. The index k is ***NOT*** part of the input.

2. You are hired as a cyclist for the Giggle Highway View project, which will provide street-level images along the entire US national highway system. As a pilot project, you are asked to ride the Giggle Highway-View Fixed-Gear Carbon-Fiber Bicycle from "the Giggleplex" in Portland, Oregon to "Giggleburg" in Williamsburg, Brooklyn, New York.

You are a hopeless caffeine addict, but like most Giggle employees you are also a coffee snob; you only drink independently roasted organic shade-grown single-origin espresso. After each espresso shot, you can bike up to L miles before suffering a caffeine-withdrawal migraine.

Giggle helpfully provides you with a map of the United States, in the form of an undirected graph G, whose vertices represent coffee shops that sell independently roasted organic shade-grown single-origin espresso, and whose edges represent highway connections between them. Each edge e is labeled with the length $\ell(e)$ of the corresponding stretch of highway. Naturally, there are espresso stands at both Giggle offices, represented by two specific vertices s and t in the graph G.

(a) Describe and analyze an algorithm to determine whether it is possible to bike from the Giggleplex to Giggleburg without suffering a caffeine-withdrawal migraine.

(b) You discover that by wearing a more expensive fedora, you can increase the distance L that you can bike between espresso shots. Describe and analyze and algorithm to find the minimum value of L that allows you to bike from the Giggleplex to Giggleburg without suffering a caffeine-withdrawal migraine.

3. Suppose you are given a collection of up-trees representing a partition of the set $\{1, 2, \ldots, n\}$ into subsets. **You have no idea how these trees were constructed.** You are also given an array $node[1..n]$, where $node[i]$ is a pointer to the up-tree node containing element i. Your task is to create a new array $label[1..n]$ using the following algorithm:

```
LabelEverything:
  for i ← 1 to n
      label[i] ← Find(node[i])
```

Recall that there are two natural ways to implement Find: simple pointer-chasing and pointer-chasing with path compression. Pseudocode for both methods is shown below.

```
Find(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x
```

Without path compression

```
Find(x):
  if x ≠ parent(x)
      parent(x) ← Find(parent(x))
  return parent(x)
```

With path compression

(a) What is the worst-case running time of LabelEverything if we implement Find *without* path compression?

(b) ***Prove*** that if we implement Find using path compression, LabelEverything runs in $O(n)$ time in the worst case.

4. Congratulations! You have successfully conquered Camelot, transforming the former battle-scarred kingdom with an anarcho-syndicalist commune, where citizens take turns to act as a sort of executive-officer-for-the-week, but with all the decisions of that officer ratified at a special bi-weekly meeting, by a simple majority in the case of purely internal affairs, but by a two-thirds majority, in the case of more major. . . .

   As a final symbolic act, you order the Round Table (surprisingly, an actual circular table) to be split into pizza-like wedges and distributed to the citizens of Camelot as trophies. Each citizen has submitted a request for an angular wedge of the table, specified by two angles—for example, Sir Robin the Brave might request the wedge from 23.17° to 42°. Each citizen will be happy if and only if they receive *precisely* the wedge that they requested. Unfortunately, some of these ranges overlap, so satisfying *all* the citizens' requests is simply impossible. Welcome to politics.

   Describe and analyze an algorithm to find the maximum number of requests that can be satisfied.

5. The NSA has established several monitoring stations around the country, each one conveniently hidden in the back of a Starbucks. Each station can monitor up to 42 cell-phone towers, but can only monitor cell-phone towers within a 20-mile radius. To ensure that every cell-phone call is recorded even if some stations malfunction, the NSA requires each cell-phone tower to be monitored by at least 3 different stations.

   Suppose you know that there are n cell-phone towers and m monitoring stations, and you are given a function $\textsc{Distance}(i, j)$ that returns the distance between the ith tower and the jth station in $O(1)$ time. Describe and analyze an algorithm that either computes a valid assignment of cell-phone towers to monitoring stations, or reports correctly that there is no such assignment (in which case the NSA will build another Starbucks).

6. Consider the following closely related problems:

   - HamiltonianPath: Given an undirected graph G, determine whether G contains a ***path*** that visits every vertex of G exactly once.
   - HamiltonianCycle: Given an undirected graph G, determine whether G contains a ***cycle*** that visits every vertex of G exactly once.

   Describe a polynomial-time reduction from HamiltonianPath to HamiltonianCycle. ***Prove*** your reduction is correct. *[Hint: A polynomial-time reduction is allowed to call the black-box subroutine more than once.]*

7. An array $X[1..n]$ of distinct integers is ***wobbly*** if it alternates between increasing and decreasing: $X[i] < X[i+1]$ for every odd index i, and $X[i] > X[i+1]$ for every even index i. For example, the following 16-element array is wobbly:

12	13	0	16	13	31	5	7	−1	23	8	10	−4	37	17	42

   Describe and analyze an algorithm that permutes the elements of a given array to make the array wobbly.

You may use the following algorithms as black boxes:

RANDOM(k): Given any positive integer k, return an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, k\}$ in $O(1)$ time.

ORLINMAXFLOW(V, E, c, s, t): Given a directed graph $G = (V, E)$, a capacity function $c \colon E \to \mathbb{R}^+$, and vertices s and t, return a maximum (s, t)-flow in G in $O(VE)$ time. If the capacities are integral, so is the returned maximum flow.

Any other algorithm that we described in class.

You may assume the following problems are NP-hard:

CIRCUITSAT: Given a boolean circuit, are there any input values that make the circuit output TRUE?

PLANARCIRCUITSAT: Given a boolean circuit drawn in the plane so that no two wires cross, are there any input values that make the circuit output TRUE?

3SAT: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

MAXINDEPENDENTSET: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MAXCLIQUE: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MINVERTEXCOVER: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MINSETCOVER: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MINHITTINGSET: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3COLOR: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

HAMILTONIANCYCLE: Given a graph G, is there a cycle in G that visits every vertex exactly once?

HAMILTONIANPATH: Given a graph G, is there a path in G that visits every vertex exactly once?

TRAVELINGSALESMAN: Given a graph G with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

STEINERTREE: Given an undirected graph G with some of the vertices marked, what is the minimum number of edges in a subtree of G that contains every marked vertex?

SUBSETSUM: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

PARTITION: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3PARTITION: Given a set X of $3n$ positive integers, can X be partitioned into n three-element subsets, all with the same sum?

DRAUGHTS: Given an $n \times n$ international draughts configuration, what is the largest number of pieces that can (and therefore must) be captured in a single move?

DOGE: Such N. Many P. Wow.

This exam lasts 180 minutes.
Write your answers in the separate answer booklet.
Please return this question handout and your cheat sheets with your answers.

1. Suppose you are given a sorted array of n distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, you are given an array $A[1..n]$ such that the prefix $A[1..k]$ is sorted in increasing order, the suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$. Describe and analyze an algorithm to compute the unknown integer k.

   For example, given the following array as input, your algorithm should output the integer 10.

9	13	16	18	19	23	28	31	37	42	−4	0	2	5	7	8

2. You are hired as a cyclist for the Giggle Highway View project, which will provide street-level images along the entire US national highway system. As a pilot project, you are asked to ride the Giggle Highway-View Fixed-Gear Carbon-Fiber Bicycle from "the Giggleplex" in Portland, Oregon to "Giggleburg" in Williamsburg, Brooklyn, New York.

   You are a hopeless caffeine addict, but like most Giggle employees you are also a coffee snob; you only drink independently roasted organic shade-grown single-origin espresso. After each espresso shot, you can bike up to L miles before suffering a caffeine-withdrawal migraine.

   Giggle helpfully provides you with a map of the United States, in the form of an undirected graph G, whose vertices represent coffee shops that sell independently roasted organic shade-grown single-origin espresso, and whose edges represent highway connections between them. Each edge e is labeled with the length $\ell(e)$ of the corresponding stretch of highway. Naturally, there are espresso stands at both Giggle offices, represented by two specific vertices s and t in the graph G.

   (a) Describe and analyze an algorithm to determine whether it is possible to bike from the Giggleplex to Giggleburg without suffering a caffeine-withdrawal migraine.

   (b) When you report to your supervisor (whom Giggle recently hired away from competitor Yippee!) that the ride is impossible, she demands to look at your map. "Oh, I see the problem; there are no *Starbucks* on this map!" As you look on in horror, she hands you an updated graph G' that includes a vertex for every Starbucks location in the United States, helpfully marked in Starbucks Green (Pantone® 3425 C).

   Describe and analyze an algorithm to find the minimum number of Starbucks locations you must visit to bike from the Giggleplex to Giggleburg without suffering a caffeine-withdrawal migraine. More formally, your algorithm should find the minimum number of green vertices on any path in G' from s to t that uses only edges of length at most L.

3. Suppose you are given a collection of up-trees representing a partition of the set $\{1, 2, \ldots, n\}$ into subsets. **You have no idea how these trees were constructed.** You are also given an array $node[1..n]$, where $node[i]$ is a pointer to the up-tree node containing element i. Your task is to create a new array $label[1..n]$ using the following algorithm:

```
LabelEverything:
  for i ← 1 to n
      label[i] ← Find(node[i])
```

Recall that there are two natural ways to implement Find: simple pointer-chasing and pointer-chasing with path compression. Pseudocode for both methods is shown below.

```
Find(x):
  while x ≠ parent(x)
      x ← parent(x)
  return x
```

Without path compression

```
Find(x):
  if x ≠ parent(x)
      parent(x) ← Find(parent(x))
  return parent(x)
```

With path compression

(a) What is the worst-case running time of LabelEverything if we implement Find *without* path compression?

(b) ***Prove*** that if we implement Find using path compression, LabelEverything runs in $O(n)$ time in the worst case.

4. Congratulations! You have successfully conquered Camelot, transforming the former battle-scarred kingdom with an anarcho-syndicalist commune, where citizens take turns to act as a sort of executive-officer-for-the-week, but with all the decisions of that officer ratified at a special bi-weekly meeting, by a simple majority in the case of purely internal affairs, but by a two-thirds majority, in the case of more major. . . .

   As a final symbolic act, you order the Round Table (surprisingly, an actual circular table) to be split into pizza-like wedges and distributed to the citizens of Camelot as trophies. Each citizen has submitted a request for an angular wedge of the table, specified by two angles—for example, Sir Robin the Brave might request the wedge from 17° to 42°. Each citizen will be happy if and only if they receive *precisely* the wedge that they requested. Unfortunately, some of these ranges overlap, so satisfying *all* the citizens' requests is simply impossible. Welcome to politics.

   Describe and analyze an algorithm to find the maximum number of requests that can be satisfied.

5. The NSA has established several monitoring stations around the country, each one conveniently hidden in the back of a Starbucks. Each station can monitor up to 42 cell-phone towers, but can only monitor cell-phone towers within a 20-mile radius. To ensure that every cell-phone call is recorded even if some stations malfunction, the NSA requires each cell-phone tower to be monitored by at least 3 different stations.

   Suppose you know that there are n cell-phone towers and m monitoring stations, and you are given a function $\textsc{Distance}(i, j)$ that returns the distance between the ith tower and the jth station in $O(1)$ time. Describe and analyze an algorithm that either computes a valid assignment of cell-phone towers to monitoring stations, or reports correctly that there is no such assignment (in which case the NSA will build another Starbucks).

6. Consider the following closely related problems:

   - HamiltonianPath: Given an undirected graph G, determine whether G contains a ***path*** that visits every vertex of G exactly once.
   - HamiltonianCycle: Given an undirected graph G, determine whether G contains a ***cycle*** that visits every vertex of G exactly once.

   Describe a polynomial-time reduction from HamiltonianCycle to HamiltonianPath. ***Prove*** your reduction is correct. *[Hint: A polynomial-time reduction is allowed to call the black-box subroutine more than once.]*

7. An array $X[1..n]$ of distinct integers is ***wobbly*** if it alternates between increasing and decreasing: $X[i] < X[i+1]$ for every odd index i, and $X[i] > X[i+1]$ for every even index i. For example, the following 16-element array is wobbly:

12	13	0	16	13	31	5	7	−1	23	8	10	−4	37	17	42

   Describe and analyze an algorithm that permutes the elements of a given array to make the array wobbly.

You may use the following algorithms as black boxes:

Random(k): Given any positive integer k, return an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, k\}$ in $O(1)$ time.

OrlinMaxFlow(V, E, c, s, t): Given a directed graph $G = (V, E)$, a capacity function $c \colon E \to \mathbb{R}^+$, and vertices s and t, return a maximum (s, t)-flow in G in $O(VE)$ time. If the capacities are integral, so is the returned maximum flow.

Any other algorithm that we described in class.

You may assume the following problems are NP-hard:

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

PlanarCircuitSat: Given a boolean circuit drawn in the plane so that no two wires cross, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

HamiltonianCycle: Given a graph G, is there a cycle in G that visits every vertex exactly once?

HamiltonianPath: Given a graph G, is there a path in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

SteinerTree: Given an undirected graph G with some of the vertices marked, what is the minimum number of edges in a subtree of G that contains every marked vertex?

SubsetSum: Given a set X of positive integers and an integer k, does X have a subset whose elements sum to k?

Partition: Given a set X of positive integers, can X be partitioned into two subsets with the same sum?

3Partition: Given a set X of $3n$ positive integers, can X be partitioned into n three-element subsets, all with the same sum?

Draughts: Given an $n \times n$ international draughts configuration, what is the largest number of pieces that can (and therefore must) be captured in a single move?

Doge: Such N. Many P. Wow.

“CS 374” Fall 2014 — Homework 0

Due Tuesday, September 2, 2014 at noon

• • • Some important course policies • • •

- **Each student must submit individual solutions for this homework.** You may use any source at your disposal—paper, electronic, or human—but you ***must*** cite ***every*** source that you use. See the academic integrity policies on the course web site for more details. For all future homeworks, groups of up to three students will be allowed to submit joint solutions.

- **Submit your solutions on standard printer/copier paper.** At the top of each page, please clearly print your name and NetID, and indicate your registered discussion section. Use both sides of the paper. If you plan to write your solutions by hand, please print the last three pages of this homework as templates. If you plan to typeset your homework, you can find a LaTeX template on the course web site; well-typeset homework will get a small amount of extra credit.

- **Submit your solutions in the drop boxes outside 1404 Siebel.** There is a separate drop box for each numbered problem. Don’t staple your entire homework together. Don’t give your homework to Jeff in class; he is fond of losing important pieces of paper.

- **Avoid the Three Deadly Sins!** There are a few dangerous writing (and thinking) habits that will trigger an automatic zero on any homework or exam problem. Yes, we are completely serious.
  - Give complete solutions, not just examples.
  - Declare all your variables.
  - Never use weak induction.

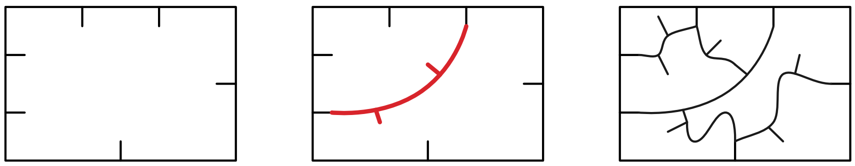
- Answering any homework or exam problem (or subproblem) in this course with “I don’t know” *and nothing else* is worth 25% partial credit. We will accept synonyms like “No idea” or “WTF”, but you must write *something*.

See the course web site for more information.

If you have any questions about these policies,
please don’t hesitate to ask in class, in office hours, or on Piazza.

---

1. The Terminal Game is a two-person game played with pen and paper. The game begins by drawing a rectangle with n "terminals" protruding into the rectangles, for some positive integer n, as shown in the figure below. On a player's turn, she selects two terminals, draws a simple curve from one to the other without crossing any other curve (or itself), and finally draws a new terminal on each side of the curve. A player loses if it is her turn and no moves are possible, that is, if no two terminals may be connected without crossing at least one other curve.



The initial setup. The first turn. No more moves.

Analyze this game, answering the following questions (and any more that you determine the answers to): When is it better to play first, and when it is better to play second? Is there always a winning strategy? What is the smallest number of moves in which you can defeat your opponent? Prove your answers are correct.

2. Herr Professor Doktor Georg von den Dschungel has a 23-node binary tree, in which each node is labeled with a unique letter of the German alphabet, which is just like the English alphabet with four extra letters: **Ä**, **Ö**, **Ü**, and **ß**. (Don't confuse these with **A**, **O**, **U**, and **B**!) Preorder and postorder traversals of the tree visit the nodes in the following order:

   - Preorder: **B K Ü E H L Z I Ö R C ß T S O A Ä D F M N U G**
   - Postorder: **H I Ö Z R L E C Ü S O T A ß K D M U G N F Ä B**

   (a) List the nodes in Professor von den Dschungel's tree in the order visited by an inorder traversal.

   (b) Draw Professor von den Dschungel's tree.

3. Recursively define a set L of strings over the alphabet $\{\mathtt{0},\mathtt{1}\}$ as follows:

   - The empty string ε is in L.
   - For any two strings x and y in L, the string $\mathtt{0}x\mathtt{1}y\mathtt{0}$ is also in L.
   - These are the only strings in L.

   (a) Prove that the string **000010101010010100** is in L.

   (b) Prove by induction that every string in L has exactly twice as many **0**s as **1**s.

   (c) Give an example of a string with exactly twice as many **0**s as **1**s that is *not* in L.

   Let $\#(a,w)$ denote the number of times symbol a appears in string w; for example,

$$\#(\mathtt{0},\mathtt{000010101010010100}) = 12 \qquad \text{and} \qquad \#(\mathtt{1},\mathtt{000010101010010100}) = 6.$$

   You may assume without proof that $\#(a,xy) = \#(a,x) + \#(a,y)$ for any symbol a and any strings x and y.

4. **This is an extra credit problem. Submit your solutions in the drop box for problem 2 (but don't staple your solutions for 2 and 4 together).**

   A *perfect riffle shuffle*, also known as a *Faro shuffle*, is performed by cutting a deck of cards exactly in half and then *perfectly* interleaving the two halves. There are two different types of perfect shuffles, depending on whether the top card of the resulting deck comes from the top half or the bottom half of the original deck. An *out-shuffle* leaves the top card of the deck unchanged. After an in-shuffle, the original top card becomes the second card from the top. For example:

$$\mathit{OutShuffle}(\mathrm{A}\spadesuit\ 2\spadesuit\ 3\spadesuit\ 4\spadesuit\ 5\heartsuit\ 6\heartsuit\ 7\heartsuit\ 8\heartsuit) = \mathrm{A}\spadesuit\ 5\heartsuit\ 2\spadesuit\ 6\heartsuit\ 3\spadesuit\ 7\heartsuit\ 4\spadesuit\ 8\heartsuit$$

$$\mathit{InShuffle}(\mathrm{A}\spadesuit\ 2\spadesuit\ 3\spadesuit\ 4\spadesuit\ 5\heartsuit\ 6\heartsuit\ 7\heartsuit\ 8\heartsuit) = 5\heartsuit\ \mathrm{A}\spadesuit\ 6\heartsuit\ 2\spadesuit\ 7\heartsuit\ 3\spadesuit\ 8\heartsuit\ 4\spadesuit$$

   (If you are unfamiliar with playing cards, please refer to the Wikipedia article https://en.wikipedia.org/wiki/Standard_52-card_deck.)

   Suppose we start with a deck of 2^n distinct cards, for some non-negative integer n. What is the effect of performing exactly n perfect in-shuffles on this deck? Prove your answer is correct!

"CS 374" Fall 2014 — Homework 1

Due Tuesday, September 9, 2014 at noon

Groups of up to three students may submit common solutions for each problem in this homework and in all future homeworks. You are responsible for forming you own groups; you are welcome to advertise for group members on Piazza. You need not use the same group for every homework, or even for every problem in a single homework. Please clearly print the names and NetIDs of each of your group members at the top of each submitted solution, along with ***one*** discussion section where we should return your graded work. If you submit hand-written solutions, please use the last three pages of this homework as templates.

---

1. Give regular expressions for each of the following languages over the alphabet $\{0, 1\}$. You do not need to prove your answers are correct.

   (a) All strings with an odd number of 1s.

   (b) All strings with at most three 0s.

   (c) All strings that do not contain the substring 010.

   (d) All strings in which every occurrence of the substring 00 occurs before every occurrence of the substring 11.

2. Recall that the ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:

$$w^R = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x \end{cases}$$

   The reversal L^R of a language L is defined as the set of reversals of all strings in L:

$$L^R := \left\{ w^R \;\middle|\; w \in L \right\}$$

   (a) Prove that $(L^*)^R = (L^R)^*$ for every language L.

   (b) Prove that the reversal of any regular language is also a regular language. (You may assume part (a) even if you haven't proved it yet.)

   You may assume the following facts without proof:

   - $L^* \bullet L^* = L^*$ for every language L.
   - $(w^R)^R = w$ for every string w.
   - $(x \bullet y)^R = y^R \bullet x^R$ for all strings x and y.

   [Hint: Yes, all three proofs use induction, but induction on what? And yes, all ***three*** *proofs.]*

3. Describe context-free grammars for each of the following languages over the alphabet $\{0, 1\}$. Explain *briefly* why your grammars are correct; in particular, describe *in English* the language generated by each non-terminal in your grammars. (We are not looking for full formal proofs of correctness, but convincing evidence that *you* understand why your answers are correct.)

   (a) The set of all strings with more than twice as many 0s as 1s.

   (b) The set of all strings that are *not* palindromes.

   *(c) **[Extra credit]** The set of all strings that are *not* equal to ww for any string w.
   [Hint: $a + b = b + a$.]

“CS 374” Fall 2014 — Homework 2

Due Tuesday, September 16, 2014 at noon

Groups of up to three students may submit common solutions for each problem in this homework and in all future homeworks. You are responsible for forming you own groups; you are welcome to advertise for group members on Piazza. You need not use the same group for every homework, or even for every problem in a single homework. Please clearly print the names and NetIDs of each of your group members at the top of each submitted solution, along with ***one*** discussion section where we should return your graded work. If you submit hand-written solutions, please use the last three pages of this homework as templates.

---

1. ***C comments*** are the set of strings over alphabet $\Sigma = \{*, /, \mathtt{A}, \diamond, \hookleftarrow\}$ that form a proper comment in the C program language and its descendants, like C++ and Java. Here $\hookleftarrow$ represents the newline character, $\diamond$ represents any other whitespace character (like the space and tab characters), and A represents any non-whitespace character other than $*$ or /.[1] There are two types of C comments:

   - Line comments: Strings of the form `// ...` $\hookleftarrow$.
   - Block comments: Strings of the form `/* ... */`.

   Following the C99 standard, we explicitly disallow ***nesting*** comments of the same type. A line comment starts with `//` and ends at the first $\hookleftarrow$ after the opening `//`. A block comment starts with `/*` and ends at the the first `*/` completely after the opening `/*`; in particular, every block comment has at least two $*$s. For example, the following strings are all valid C comments:

   - `/***/`
   - `//`$\diamond$`//`$\diamond\hookleftarrow$
   - `/*///`$\diamond$`*`$\diamond\hookleftarrow$`**/`
   - `/*`$\diamond$`//`$\diamond\hookleftarrow\diamond$`*/`

   On the other hand, the following strings are *not* valid C comments:

   - `/*/`
   - `//`$\diamond$`//`$\diamond\hookleftarrow\diamond\hookleftarrow$
   - `/*`$\diamond$`/*`$\diamond$`*/`$\diamond$`*/`

   (a) Describe a DFA that accepts the set of all C comments.

   (b) Describe a DFA that accepts the set of all strings composed entirely of blanks($\diamond$), newlines($\hookleftarrow$), and C comments.

   You must explain *in English* how your DFAs work. Drawings or formal descriptions without English explanations will receive no credit, even if they are correct.

2. Construct a DFA for the following language over alphabet $\{0, 1\}$:

$$L = \left\{ w \in \{0,1\}^* \;\middle|\; \begin{array}{l} \text{the number represented by binary string } w \text{ is divisible} \\ \text{by 19, but the length of } w \text{ is not a multiple of 23} \end{array} \right\}.$$

You must explain *in English* how your DFA works. A formal description without an English explanation will receive no credit, even if it is correct. Don't even try to draw the DFA.

3. Prove that each of the following languages is *not* regular.

(a) $\left\{ w \in \{0\}^* \;\middle|\; \text{length of } w \text{ is a perfect square; that is, } |w| = k^2 \text{ for some integer } k \right\}$.

(b) $\left\{ w \in \{0,1\}^* \;\middle|\; \text{the number represented by } w \text{ as a binary string is a perfect square} \right\}$.

*4. **[Extra credit]** Suppose L is a regular language which guarantees to contain at least one palindrome. Prove that if an n-state DFA M accepts L, then L contains a palindrome of length polynomial in n. What is the polynomial bound you get?

---

[1]The actual C commenting syntax is considerably more complex than described here, because of character and string literals.

- The opening `/*` or `//` of a comment must not be inside a string literal (`"..."`) or a (multi-)character literal (`'...'`).
- The opening double-quote of a string literal must not be inside a character literal (`'"'`) or a comment.
- The closing double-quote of a string literal must not be escaped (`\"`)
- The opening single-quote of a character literal must not be inside a string literal (`"...'..."`) or a comment.
- The closing single-quote of a character literal must not be escaped (`\'`)
- A backslash escapes the next symbol if and only if it is not itself escaped (`\\`) or inside a comment.

For example, the string `"/*\\\"*/"/*"/*\"/*"*/` is a valid string literal (representing the 5-character string `/*\"\*/`, which is itself a valid block comment!) followed immediately by a valid block comment. For this homework question, just pretend that the characters `'`, `"`, and `\` don't exist.

The C++ commenting is even more complicated, thanks to the addition of *raw* string literals. Don't ask.

Some C and C++ compilers do support nested block comments, in violation of the language specification. A few other languages, like OCaml, explicitly allow nesting comments.

"CS 374" Fall 2014 — Homework 3

Due Tuesday, September 23, 2014 at noon

- As usual, groups of up to three students may submit common solutions for this assignment. Each group should submit exactly ***one*** solution for each problem. Please clearly print the names and NetIDs of each of your group members at the top of each submitted solution, along with ***one*** discussion section where we should return your graded work. If you submit hand-written solutions, please use the last three pages of this handout as templates.
- If a question asks you to construct an NFA, you are welcome to use ε-transitions.

---

1. For each of the following regular expressions, describe or draw two finite-state machines:
   - An NFA that accepts the same language, using Thompson's algorithm (described in class and in the notes)
   - An equivalent DFA, using the incremental subset construction described in class. For each state in your DFA, identify the corresponding subset of states in your NFA. Your DFA should have no unreachable states.

   (a) $(\mathtt{01} + \mathtt{10})^*(\mathtt{0} + \mathtt{1} + \varepsilon)$

   (b) $\mathtt{1}^* + (\mathtt{10})^* + (\mathtt{100})^*$

2. Prove that for any regular language L, the following languages are also regular:

   (a) $\text{SUBSTRINGS}(L) := \{x \mid wxy \in L \text{ for some } w, y \in \Sigma^*\}$

   (b) $\text{HALF}(L) := \{w \mid ww \in L\}$

   [Hint: Describe how to transform a DFA for L into NFAs for $\text{SUBSTRINGS}(L)$ *and* $\text{HALF}(L)$*. What do your NFAs have to guess? Don't forget to explain* ***in English*** *how your NFAs work.]*

3. Which of the following languages over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$ are regular and which are not? Prove your answers are correct. Recall that Σ^+ denotes the set of all *nonempty* strings over Σ.

   (a) $\{wxw \mid w, x \in \Sigma^+\}$

   (b) $\{wxx \mid w, x \in \Sigma^+\}$

   (c) $\{wxwy \mid w, x, y \in \Sigma^+\}$

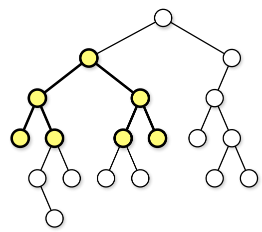
   (d) $\{wxxy \mid w, x, y \in \Sigma^+\}$

“CS 374” Fall 2014 — Homework 4

Due Tuesday, October 7, 2014 at noon

---

1. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return both the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

2. Consider the following cruel and unusual sorting algorithm.

```
CRUEL(A[1..n]):
  if n > 1
      CRUEL(A[1..n/2])
      CRUEL(A[n/2+1..n])
      UNUSUAL(A[1..n])
```

```
UNUSUAL(A[1..n]):
  if n = 2
      if A[1] > A[2]                  ⟨⟨the only comparison!⟩⟩
          swap A[1] ↔ A[2]
  else
      for i ← 1 to n/4                ⟨⟨swap 2nd and 3rd quarters⟩⟩
          swap A[i+n/4] ↔ A[i+n/2]
      UNUSUAL(A[1..n/2])              ⟨⟨recurse on left half⟩⟩
      UNUSUAL(A[n/2+1..n])            ⟨⟨recurse on right half⟩⟩
      UNUSUAL(A[n/4+1..3n/4])         ⟨⟨recurse on middle half⟩⟩
```

Notice that the comparisons performed by the algorithm do not depend at all on the values in the input array; such a sorting algorithm is called ***oblivious***. Assume for this problem that the input size n is always a power of 2.

(a) Prove by induction that CRUEL correctly sorts any input array. *[Hint: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough?]*

(b) Prove that CRUEL would *not* correctly sort if we removed the for-loop from UNUSUAL.

(c) Prove that CRUEL would *not* correctly sort if we swapped the last two lines of UNUSUAL.

(d) What is the running time of UNUSUAL? Justify your answer.

(e) What is the running time of CRUEL? Justify your answer.

3. In the early 20th century, a German mathematician developed a variant of the Towers of Hanoi game, which quickly became known in the American literature as "Liberty Towers".[1] In this variant, there is a row of $k \geq 3$ pegs, numbered in order from 1 to k. In a single turn, for any index i, you can move the smallest disk on peg i to either peg $i-1$ or peg $i+1$, subject to the usual restriction that you cannot place a bigger disk on a smaller disk. Your mission is to move a stack of n disks from peg 1 to peg k.

   (a) Describe and analyze a recursive algorithm for the case $k = 3$. ***Exactly*** how many moves does your algorithm perform?

   (b) Describe and analyze a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^3)$ moves. To simplify the algorithm, assume that n is a power of 2. *[Hint: Use part (a).]*

   (c) **[Extra credit]** Describe and analyze a recursive algorithm for the case $k = n+1$ that requires at most $O(n^2)$ moves. Do *not* assume that n is a power of 2. *[Hint: Don't use part (a).]*

   (d) **[Extra credit]** Describe and analyze a recursive algorithm for the case $k = \sqrt{n} + 1$ that requires at most a polynomial number of moves. To simplify the algorithm, assume that n is a power of 4. What polynomial bound do you get? *[Hint: Use part (a)!]*

   ★(e) **[Extra extra credit]** Describe and analyze a recursive algorithm for arbitrary n and k. How small must k be (as a function of n) so that the number of moves is bounded by a polynomial in n? (This is actually an open research problem, a phrase which here means "Nobody knows the best answer.")

[1]No, not really. During World War I, many German-derived or Germany-related names were changed to more patriotic variants. For example, sauerkraut became "liberty cabbage", hamburgers became "liberty sandwiches", frankfurters became "Liberty sausages" or "hot dogs", German measles became "liberty measles", dachsunds became "liberty pups", German shepherds became "Alsatians", and pinochle (the card game) became "Liberty". For more recent anti-French examples, see "freedom fries", "freedom toast", and "liberty lip lock". Americans are weird.

“CS 374” Fall 2014 — Homework 5

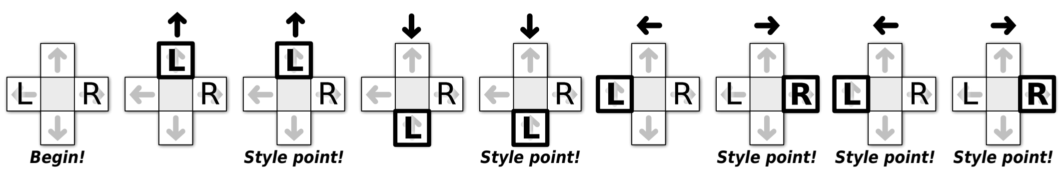
Due Tuesday, October 14, 2014 at noon

---

1. ***Dance Dance Revolution*** is a dance video game, first introduced in Japan by Konami in 1998. Players stand on a platform marked with four arrows, pointing forward, back, left, and right, arranged in a cross pattern. During play, the game plays a song and scrolls a sequence of n arrows (←, ↑, ↓, or →) from the bottom to the top of the screen. At the precise moment each arrow reaches the top of the screen, the player must step on the corresponding arrow on the dance platform. (The arrows are timed so that you’ll step with the beat of the song.)

   You are playing a variant of this game called “Vogue Vogue Revolution”, where the goal is to play perfectly but move as little as possible. When an arrow reaches the top of the screen, if one of your feet is already on the correct arrow, you are awarded one style point for maintaining your current pose. If neither foot is on the right arrow, you must move one (and *only* one) of your feet from its current location to the correct arrow on the platform. If you ever step on the wrong arrow, or fail to step on the correct arrow, or move more than one foot at a time, or move either foot when you are already standing on the correct arrow, or insult Beyoncé, all your style points are immediately taken away and you lose.

   How should you move your feet to maximize your total number of style points? For purposes of this problem, assume you always start with you left foot on ← and you right foot on →, and that you’ve memorized the entire sequence of arrows. For example, if the sequence is ↑↑↓↓←→←→, you can earn 5 style points by moving you feet as shown below:



   Describe and analyze an efficient algorithm to find the maximum number of style points you can earn during a given VVR routine. Your input is an array $Arrow[1\,..\,n]$ containing the sequence of arrows.

2. Recall that a *palindrome* is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`.

   Any string can be decomposed into a sequence of palindrome substrings. For example, the string `BUBBASEESABANANA` (“Bubba sees a banana.”) can be broken into palindromes in the following ways (among many others):

   `BUB` • `BASEESAB` • `ANANA`

   `B` • `U` • `BB` • `A` • `SEES` • `ABA` • `NAN` • `A`

   `B` • `U` • `BB` • `A` • `SEES` • `A` • `B` • `ANANA`

   `B` • `U` • `B` • `B` • `A` • `S` • `E` • `E` • `S` • `A` • `B` • `ANA` • `N` • `A`

   Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string `BUBBASEESABANANA`, your algorithm would return the integer 3.

3. Suppose you are given a DFA $M = (\{\mathtt{0}, \mathtt{1}\}, Q, s, A, \delta)$ and a binary string $w \in \{\mathtt{0}, \mathtt{1}\}^*$.

   (a) Describe and analyze an algorithm that computes the longest subsequence of w that is accepted by M, or correctly reports that M does not accept any subsequence of w.

   *(b) **[Extra credit]** Describe and analyze an algorithm that computes the *shortest supersequence* of w that is accepted by M, or correctly reports that M does not accept any supersequence of w. (Recall that a string x is a supersequence of w if and only if w is a subsequence of x.)

   Analyze both of your algorithms in terms of the parameters $n = |w|$ and $k = |Q|$.

Rubric (for all dynamic programming problems): As usual, a score of x on the following 10-point scale corresponds to a score of $\lceil x/3 \rceil$ on the 4-point homework scale.

- 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
  - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Automatic zero if the English description is missing.**
  - + 1 point for stating how to call your function to get the final answer.
  - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
  - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
- 4 points for details of the dynamic programming algorithm
  - + 1 point for describing the memoization data structure
  - + 2 points for describing a correct evaluation order; a clear picture is sufficient. If you use nested loops, be sure to specify the nesting order.
  - + 1 point for time analysis
- It is *not* necessary to state a space bound.
- For problems that ask for an algorithm that computes an optimal *structure*—such as a subset, partition, subsequence, or tree—an algorithm that computes only the *value* or *cost* of the optimal structure is sufficient for full credit.
- Official solutions usually include pseudocode for the final iterative dynamic programming algorithm, ***but this is not required for full credit***. If your solution includes iterative pseudocode, you do not need to separately describe the recurrence, data structure, or evaluation order. (But you still need to describe the underlying recursive function in English.)
- Official solutions will provide target time bounds. Algorithms that are faster than this target are worth more points; slower algorithms are worth fewer points, typically by 2 or 3 points for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.

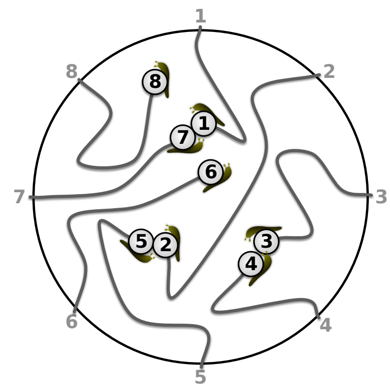
  We rarely include these target time bounds in the actual questions, because when we do include them, significantly more students turn in algorithms that meet the target time bound but don't work (earning 0/10) instead of correct algorithms that are slower than the target time bound (earning 8/10).

“CS 374” Fall 2014 — Homework 6

Due Tuesday, October 21, 2014 at noon

---

1. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.



The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates.
The organizers must pay $M[3,4] + M[2,5] + M[1,7]$.

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i,j] = M[j,i]$ is the reward to be paid if snails i and j meet. Rewards may be positive, negative, or zero.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

2. Consider a weighted version of the class scheduling problem, where different classes offer different number of credit hours, which are of course totally unrelated to the duration of the class lectures. Given arrays $S[1..n]$ of start times, an array $F[1..n]$ of finishing times, and an array $H[1..n]$ of credit hours as input, your goal is to choose a set of non-overlapping classes with the largest possible number of credit hours.

   (a) Prove that the greedy algorithm described in class — Choose the class that ends first and recurse — does *not* always return the best schedule.

   (b) Describe an efficient algorithm to compute the best schedule.

   In addition to submitting a solution on paper as usual, please *individually* submit an electronic solution for this problem on CrowdGrader. Please see the course web page for detailed instructions.

3. Suppose you have just purchased a new type of hybrid car that uses fuel extremely efficiently, but can only travel 100 miles on a single battery. The car's fuel is stored in a single-use battery, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and can only be installed by licensed battery-replacement technicians. Thus, even if you decide to replace your battery early, you must still pay full price for the new battery to be installed. Moreover, because these batteries are in high demand, no one can afford to own more than one battery at a time.

   Suppose you are trying to get from San Francisco to New York City on the new Inter-Continental Super-Highway, which runs in a direct line between these two cities. There are several fueling stations along the way; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every fueling station on the ICSH. Given this information, how do you decide the best places to stop for fuel?

   More formally, suppose you are given two arrays $D[1..n]$ and $C[1..n]$, where $D[i]$ is the distance from the start of the highway to the ith station, and $C[i]$ is the cost to replace your battery at the ith station. Assume that your trip starts and ends at fueling stations (so $D[1] = 0$ and $D[n]$ is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

   (a) Describe and analyze a greedy algorithm to find the minimum number of refueling stops needed to complete your trip. Don't forget to prove that your algorithm is correct.

   (b) But what you really want to minimize is the total *cost* of travel. Show that your greedy algorithm in part (a) does *not* produce an optimal solution when extended to this setting.

   (c) Describe an efficient algorithm to compute the locations of the fuel stations you should stop at to minimize the total cost of travel.

“CS 374” Fall 2014 — Homework 7

Due Tuesday, October 28, 2014 at noon

---

1. You are standing next to a water pond, and you have three empty jars. Each jar holds a positive integer number of gallons; the capacities of the three jars may or may not be different. You want one of the jars (which one doesn't matter) to contain exactly k gallons of water, for some integer k. You are only allowed to perform the following operations:

   (a) Fill a jar with water from the pond until the jar is full.

   (b) Empty a jar of water by pouring water into the pond.

   (c) Pour water from one jar to another, until either the first jar is empty or the second jar is full, whichever happens first.

   For example, suppose your jars hold 6, 10, and 15 gallons. Then you can put 13 gallons of water into the third jar in six steps:

   - Fill the third jar from the pond.
   - Fill the first jar from the third jar. (Now the third jar holds 9 gallons.)
   - Empty the first jar into the pond.
   - Fill the second jar from the pond.
   - Fill the first jar from the second jar. (Now the second jar holds 4 gallons.)
   - Empty the second jar into the third jar.

   Describe an efficient algorithm that finds the minimum number of operations required to obtain a jar containing exactly k gallons of water, or reports correctly that obtaining exactly k gallons of water is impossible, given the capacities of the three jars and a positive integer k as input. For example, given the four numbers $6, 10, 15$ and 13 as input, your algorithm should return the number 6 (for the sequence of operations listed above).

2. Consider a directed graph G, where each edge is colored either red, white, or blue. A walk[1] in G is called a *French flag walk* if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \to v_1 \to \cdots \to v_k$ is a French flag path if, for every integer i, the edge $v_i \to v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

   Describe an efficient algorithm to find all vertices in a given edge-colored directed graph G that can be reached from a given vertex v through a French flag walk.

3. Suppose we are given a directed acyclic graph G where every edge e has a positive integer weight $w(e)$, along with two specific vertices s and t and a positive integer W.

   (a) Describe an efficient algorithm to find the *longest* path (meaning the largest number of edges) from s to t in G with total weight at most W. *[Hint: Use dynamic programming.]*

   (b) **[Extra credit]** Solve part (a) with a running time that does not depend on W.

---

[1] Recall that a ***walk*** in G is a sequence of vertices $v_0 \to v_1 \to \cdots \to v_k$, such that $v_{i-1} \to v_i$ is an edge in G for every index i. A ***path*** is a walk in which no vertex appears more than once.

"CS 374" Fall 2014 — Homework 8

Due Tuesday, November 4, 2014 at noon

---

1. After a grueling algorithms midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Champaign-Urbana is currently suffering from a plague of zombies, so even though the bus stops have fences that *supposedly* keep the zombies out, you'd still like to spend as little time waiting at bus stops as possible. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between buses at least once.

   Describe and analyze an algorithm to determine a sequence of bus rides from Siebel to your home, that minimizes the total time you spend waiting at bus stops. You can assume that there are b different bus lines, and each bus stops n times per day. Assume that the buses run exactly on schedule, that you have an accurate watch, and that walking between bus stops is too dangerous to even contemplate.

2. It is well known that the global economic collapse of 2017 was caused by computer scientists indiscriminately abusing weaknesses in the currency exchange market. *Arbitrage* was a money-making scheme that takes advantage of inconsistencies in currency exchange rates. Suppose a currency trader with $1,000,000 discovered that 1 US dollar could be traded for 120 Japanese yen, 1 yen could be traded for 0.01 euros, and 1 euro could be traded for 1.2 US dollars. Then by converting his money from dollars to yen, then from yen to euros, and finally from euros back to dollars, the trader could instantly turn his $1,000,000 into $1,440,000! The cycle of currencies $\$ \to ¥ \to € \to \$$ was called an ***arbitrage cycle***. Finding and exploiting arbitrage cycles before the prices were corrected required extremely fast algorithms. Of course, now that the entire world uses plastic bags as currency, such abuse is impossible.

   Suppose n different currencies are traded in the global currency market. You are given a two-dimensional array $Exch[1..n, 1..n]$ of exchange rates between every pair of currencies; for all indices i and j, one unit of currency i buys $Exch[i, j]$ units of currency j. (Do *not* assume that $Exch[i, j] \cdot Exch[j, i] = 1$.)

   (a) Describe an algorithm that computes an array $Most[1..n]$, where $Most[i]$ is the largest amount of currency i that you can obtain by trading, starting with one unit of currency 1, assuming there are no arbitrage cycles.

   (b) Describe an algorithm to determine whether the given array of currency exchange rates creates an arbitrage cycle.

3. Describe and analyze and algorithm to find the *second smallest spanning tree* of a given undirected graph G with weighted edges, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree. Because the minimum spanning tree is haunted, or something.

"CS 374" Fall 2014 — Homework 9

Due Tuesday, November 18, 2014 at noon

The following questions ask you to describe various Turing machines. In each problem, give *both* a formal description of your Turing machine in terms of specific states, tape symbols, and transition functions *and* explain in English how your Turing machine works. In particular:

- Clearly specify what variant of Turing machine you are using: Number of tapes, number of heads, allowed head motions, halting conditions, and so on.
- Include the type signature of your machine's transition function. The standard model uses a transition function whose signature is $\delta: Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.
- If necessary, break your Turing machine into smaller functional pieces, and describe those pieces separately (both formally and in English).
- Use state names that convey their meaning/purpose.

---

1. Describe a Turing machine that computes the function $\lceil \log_2 n \rceil$. Given the string 1^n as input, for any positive integer n, your machine should return the string $1^{\lceil \log_2 n \rceil}$ as output. For example, given the input string 1111111111111 (thirteen 1s), your machine should output the string 1111, because $2^3 < 13 \le 2^4$.

2. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current cell to its *P*arent, its *L*eft child, or to its *R*ight child. Thus, the transition function of such a machine has the form $\delta: Q \times \Gamma \to Q \times \Gamma \times \{P, L, R\}$. The input string is initially given along the left spine of the tape.

   Prove that any binary-tree Turing machine can be simulated by a standard Turing machine. That is, given any binary-tree Turing machine $M = (\Gamma, \square, \Sigma, Q, \mathsf{start}, \mathsf{accept}, \mathsf{reject}, \delta)$, describe a standard Turing machine $M' = (\Gamma', \square', \Sigma, Q', \mathsf{start}', \mathsf{accept}', \mathsf{reject}', \delta')$ that accepts and rejects exactly the same input strings as M. Be sure to describe how a single transition of M is simulated by M'.

   In addition to submitting paper solutions, please also electronically submit your solution to this problem on CrowdGrader.

3. **[Extra credit]**

   A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, formally describe a tag-Turing machine M' that accepts and rejects exactly the same strings as M. Be sure to describe how a single transition of M is simulated by M'.

“CS 374” Fall 2014 — Homework 10

Due Tuesday, December 2, 2014 at noon

---

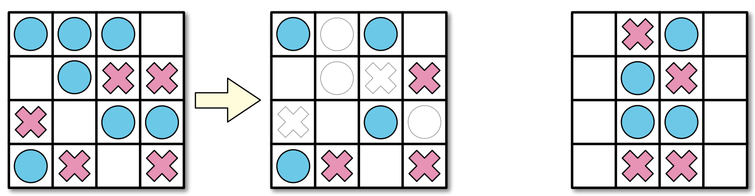
1. Consider the following problem, called BoxDepth: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?

   (a) Describe a polynomial-time reduction from BoxDepth to MaxClique.

   (b) Describe and analyze a polynomial-time algorithm for BoxDepth. *[Hint: Don't try to optimize the running time; $O(n^3)$ is good enough.]*

   (c) Why don't these two results imply that P=NP?

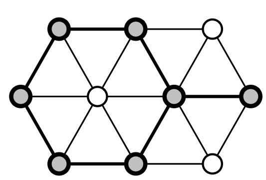
2. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.

   A solvable puzzle and one of its many solutions. An unsolvable puzzle.

   Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether this puzzle can be solved.

3. A subset S of vertices in an undirected graph G is called ***triangle-free*** if, for every triple of vertices $u, v, w \in S$, at least one of the three edges uv, uw, vw is *absent* from G. Prove that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.

   A triangle-free subset of 7 vertices.
   This is ***not*** the largest triangle-free subset in this graph.

   In addition to submitting paper solutions, please also electronically submit your solution to this problem on CrowdGrader.

4. **[Extra credit]** Describe a direct polynomial-time reduction from 4Color to 3Color. (This is significantly harder than the opposite direction, which you'll see in lab on Wednesday. Don't go through the Cook-Levin Theorem.)

"CS 374" Fall 2014 ✧ Homework 11

Due Tuesday, December 9, 2014 at noon

---

1. Recall that w^R denotes the reversal of string w; for example, $\texttt{TURING}^R = \texttt{GNIRUT}$. Prove that the following language is undecidable.

$$\textsc{RevAccept} := \left\{ \langle M \rangle \;\middle|\; M \text{ accepts } \langle M \rangle^R \right\}$$

2. Let M be a Turing machine, let w be an arbitrary input string, and let s be an integer. We say that ***M accepts w in space s*** if, given w as input, M accesses only the first s cells on the tape and eventually accepts.

   (a) Prove that the following language is decidable:

$$\left\{ \langle M, w \rangle \;\middle|\; M \text{ accepts } w \text{ in space } |w|^2 \right\}$$

   (b) Prove that the following language is undecidable:

$$\left\{ \langle M \rangle \;\middle|\; M \text{ accepts at least one string } w \text{ in space } |w|^2 \right\}$$

3. ***[Extra credit]*** For each of the following languages, either prove that the language is decidable, or prove that the language is undecidable.

   (a) $L_0 = \left\{ \langle M \rangle \;\middle|\; \text{given any input string, } M \text{ eventually leaves its } \texttt{start} \text{ state} \right\}$

   (b) $L_1 = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_0 \right\}$

   (c) $L_2 = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_1 \right\}$

   (d) $L_3 = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_2 \right\}$

   (e) $L_4 = \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_3 \right\}$

1. Prove that every non-negative integer can be represented as the sum of distinct powers of 2. ("Write it in binary" is not a proof; it's just a restatement of what you have to prove.)

2. Suppose you and your 8-year-old cousin Elmo decide to play a game with a rectangular bar of chocolate, which has been scored into an $n \times m$ grid of squares. You and Elmo alternate turns. On each turn, you or Elmo choose one of the available pieces of chocolate and break it along one of the grid lines into two smaller rectangles. Thus, at all times, each piece of chocolate is an $a \times b$ rectangle for some positive integers a and b; in particular, a 1×1 piece cannot be broken into smaller pieces. The game ends when all the pieces are individual squares. The winner is the player who breaks the last piece.

   Describe a strategy for winning this game. When should you take the first move, and when should you offer it to Elmo? On each turn, how do you decide which piece to break and where? Prove your answers are correct. *[Hint: Let's play a 3 × 3 game. You go first. Oh, and I'm kinda busy right now, so could you just play for me whenever it's my turn? Thanks.]*

3. **[To think about later]** Now consider a variant of the previous chocolate-bar game, where on each turn you can *either* break a piece into two smaller pieces *or eat a* 1×1 *piece*. This game ends when all the chocolate is gone. The winner is the player who eats the last bite of chocolate (*not* the player who eats the *most* chocolate). Describe a strategy for winning this game, and prove that your strategy works.

These lab problems ask you to prove some simple claims about recursively-defined string functions and concatenation. In each case, we want a self-contained proof by induction that relies on the formal recursive definitions, *not* on intuition. In particular, your proofs must refer to the formal recursive definition of string concatenation:

$$w \bullet z := \begin{cases} z & \text{if } w = \varepsilon \\ a \cdot (x \bullet z) & \text{if } w = ax \text{ for some symbol } a \text{ and some string } x \end{cases}$$

You may also use any of the following facts, which we proved in class:

Lemma 1: Concatenating nothing does nothing: For every string w, we have $w \bullet \varepsilon = w$.

Lemma 2: Concatenation adds length: $|w \bullet x| = |w| + |x|$ for all strings w and x.

Lemma 3: Concatenation is associative: $(w \bullet x) \bullet y = w \bullet (x \bullet y)$ for all strings w, x, and y.

---

1. Let $\#(a, w)$ denote the number of times symbol a appears in string w; for example,

$$\#(\mathtt{0}, \mathtt{000010101010010100}) = 12 \qquad \text{and} \qquad \#(\mathtt{1}, \mathtt{000010101010010100}) = 6.$$

   (a) Give a formal recursive definition of $\#(a, w)$.

   (b) Prove by induction that $\#(a, w \bullet z) = \#(a, w) + \#(a, z)$ for any symbol a and any strings w and z.

2. The ***reversal*** $\boldsymbol{w^R}$ of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

   (a) Prove that $(w \bullet x)^R = x^R \bullet w^R$ for all strings w and x.

   (b) Prove that $(w^R)^R = w$ for every string w.

Give regular expressions that describe each of the following languages over the alphabet {**0**,**1**}. We won't get to all of these in section.

1. All strings containing at least three **0**s.

2. All strings containing at least two **0**s and at least one **1**.

3. All strings containing the substring **000**.

4. All strings *not* containing the substring **000**.

5. All strings in which every run of **0**s has length at least 3.

6. All strings such that every substring **000** appears after every **1**.

7. Every string except **000**. *[Hint: Don't try to be clever.]*

8. All strings w such that *in every prefix of w*, the number of **0**s and **1**s differ by at most 1.

⋆9. All strings w such that *in every prefix of w*, the number of **0**s and **1**s differ by at most 2.

★10. All strings in which the substring **000** appears an even number of times.
(For example, **0001000** and **0000** are in this language, but **00000** is not.)

Jeff showed the context-free grammars in class on Tuesday; in each example, the grammar itself is on the left; the explanation for each non-terminal is on the right.

- Properly nested strings of parentheses.

$S \to \varepsilon \mid S\,\mathtt{(}S\mathtt{)}$	properly nested parentheses

  Here is a different grammar for the same language:

$S \to \varepsilon \mid \mathtt{(}S\mathtt{)} \mid SS$	properly nested parentheses

- $\{\mathtt{0}^m\mathtt{1}^n \mid m \neq n\}$. This is the set of all binary strings composed of some number of 0s followed by a different number of 1s.

$S \to A \mid B$	all strings $\mathtt{0}^m\mathtt{1}^n$ where $m \neq n$
$A \to \mathtt{0}A \mid \mathtt{0}C$	all strings $\mathtt{0}^m\mathtt{1}^n$ where $m > n$
$B \to B\mathtt{1} \mid C\mathtt{1}$	all strings $\mathtt{0}^m\mathtt{1}^n$ where $m < m$
$C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$	all strings $\mathtt{0}^n\mathtt{1}^n$ for some integer n

---

Give context-free grammars for each of the following languages. For each grammar, describe *in English* the language for each non-terminal, and in the examples above. As usual, we won't get to all of these in section.

1. Binary palindromes: Strings over $\{\mathtt{0},\mathtt{1}\}$ that are equal to their reversals. For example: **00111100** and **0100010**, but not **01100**.

2. $\{\mathtt{0}^{2n}\mathtt{1}^n \mid n \geq 0\}$

3. $\{\mathtt{0}^m\mathtt{1}^n \mid m \neq 2n\}$

4. $\{\mathtt{0},\mathtt{1}\}^* \setminus \{\mathtt{0}^{2n}\mathtt{1}^n \mid n \geq 0\}$

5. Strings of properly nested parentheses **()**, brackets **[]**, and braces **{}**. For example, the string **([]){}** is in this language, but the string **([)]** is not, because the left and right delimiters don't match.

6. Strings over $\{\mathtt{0},\mathtt{1}\}$ where the number of 0s is equal to the number of 1s.

7. Strings over $\{\mathtt{0},\mathtt{1}\}$ where the number of 0s is *not* equal to the number of 1s.

Construct DFA that accept each of the following languages over the alphabet $\{0, 1\}$. We won’t get to all of these in section.

---

1. (a) $(0+1)^*$
   (b) $\emptyset$
   (c) $\{\epsilon\}$

2. Every string except **000**.

3. All strings containing the substring **000**.

4. All strings *not* containing the substring **000**.

5. All strings in which the reverse of the string is the binary representation of a integer divisible by 3.

6. All strings w such that *in every prefix of w*, the number of **0**s and **1**s differ by at most 2.

Prove that each of the following languages is not regular.

1. Binary palindromes: Strings over $\{0, 1\}$ that are equal to their reversals. For example: 00111100 and 0100010, but not 01100. *[Hint: We did this in class.]*

2. $\{0^{2n}1^n \mid n \geq 0\}$

3. $\{0^m 1^n \mid m \neq 2n\}$

4. Strings over $\{0, 1\}$ where the number of 0s is exactly twice the number of 1s.

5. Strings of properly nested parentheses (), brackets [], and braces {}. For example, the string ([]){} is in this language, but the string ([)] is not, because the left and right delimiters don't match.

6. $\left\{0^{2^n} \;\middle|\; n \geq 0\right\}$ — Strings of 0s whose length is a power of 2.

7. Strings of the form $w_1\#w_2\#\cdots\#w_n$ for some $n \geq 2$, where each substring w_i is a string in $\{0, 1\}^*$, and some pair of substrings w_i and w_j are equal.

For each of the following languages over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, either prove the language is regular (by giving an equivalent regular expression, DFA, or NFA) or prove that the language is not regular (using a fooling set argument). Exactly half of these languages are regular.

1. $\{\mathtt{0}^n\mathtt{1}\mathtt{0}^n \mid n \geq 0\}$
2. $\{\mathtt{0}^n\mathtt{1}\mathtt{0}^n w \mid n \geq 0 \text{ and } w \in \Sigma^*\}$
3. $\{w\mathtt{0}^n\mathtt{1}\mathtt{0}^n x \mid w \in \Sigma^* \text{ and } n \geq 0 \text{ and } x \in \Sigma^*\}$
4. Strings in which the number of 0s and the number of 1s differ by at most 2.
5. Strings such that *in every prefix*, the number of 0s and the number of 1s differ by at most 2.
6. Strings such that *in every substring*, the number of 0s and the number of 1s differ by at most 2.

Here are several problems that are easy to solve in $O(n)$ time, essentially by brute force. Your task is to design algorithms for these problems that are significantly faster.

1. (a) Suppose $A[1..n]$ is an array of n distinct integers, sorted so that $A[1] < A[2] < \cdots < A[n]$. Each integer $A[i]$ could be positive, negative, or zero. Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists..

   (b) Now suppose $A[1..n]$ is a sorted array of n distinct ***positive*** integers. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy.]*

2. Suppose we are given an array $A[1..n]$ such that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a ***local minimum*** if both $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are exactly six local minima in the following array:

9	**7**	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9
	▲			▲				▲		▲	▲			▲	

   Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 5, because $A[5]$ is a local minimum. *[Hint: With the given boundary conditions, any array **must** contain at least one local minimum. Why?]*

3. (a) Suppose you are given two sorted arrays $A[1..n]$ and $B[1..n]$ containing distinct integers. Describe a fast algorithm to find the median (meaning the nth smallest element) of the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..8] = [2, 4, 5, 8, 17, 19, 21, 23]$$

   your algorithm should return the integer 9. *[Hint: What can you learn by comparing one element of A with one element of B?]*

   (b) ***To think about on your own:*** Now suppose you are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe a fast algorithm to find the kth smallest element in the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..5] = [2, 5, 7, 17, 19] \qquad k = 6$$

   your algorithm should return the integer 7.

Here are several problems that are easy to solve in $O(n)$ time, essentially by brute force. Your task is to design algorithms for these problems that are significantly faster, and ***prove*** that your algorithm is correct.

1. (a) Suppose $A[1..n]$ is an array of n distinct integers, sorted so that $A[1] < A[2] < \cdots < A[n]$. Each integer $A[i]$ could be positive, negative, or zero. Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists..

   (b) Now suppose $A[1..n]$ is a sorted array of n distinct ***positive*** integers. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy.]*

2. Suppose we are given an array $A[1..n]$ such that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a ***local minimum*** if both $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are exactly six local minima in the following array:

9	**7**	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9
	▲			▲				▲		▲	▲			▲	

   Describe and analyze a fast algorithm that returns the index of one local minimum. For example, given the array above, your algorithm could return the integer 5, because $A[5]$ is a local minimum. *[Hint: With the given boundary conditions, any array **must** contain at least one local minimum. Why?]*

3. (a) Suppose you are given two sorted arrays $A[1..n]$ and $B[1..n]$ containing distinct integers. Describe a fast algorithm to find the median (meaning the nth smallest element) of the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..8] = [2, 4, 5, 8, 17, 19, 21, 23]$$

   your algorithm should return the integer 9. *[Hint: What can you learn by comparing one element of A with one element of B?]*

   (b) ***To think about on your own:*** Now suppose you are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe a fast algorithm to find the kth smallest element in the union $A \cup B$. For example, given the input

$$A[1..8] = [0, 1, 6, 9, 12, 13, 18, 20] \qquad B[1..5] = [2, 5, 7, 17, 19] \qquad k = 6$$

   your algorithm should return the integer 7.

A ***subsequence*** of a sequence (for example, an array, linked list, or string), obtained by removing zero or more elements and keeping the rest in the same sequence order. A subsequence is called a ***substring*** if its elements are contiguous in the original sequence. For example:

- `SUBSEQUENCE`, `UBSEQU`, and the empty string ε are all substrings of the string `SUBSEQUENCE`;
- `SBSQNC`, `UEQUE`, and `EEE` are all subsequences of `SUBSEQUENCE` but not substrings;
- `QUEUE`, `SSS`, and `FOOBAR` are not subsequences of `SUBSEQUENCE`.

Describe recursive backtracking algorithms for the following problems. *Don't worry about running times.*

1. Given an array $A[1..n]$ of integers, compute the length of a ***longest increasing subsequence***. A sequence $B[1..\ell]$ is *increasing* if $B[i] > B[i-1]$ for every index $i \geq 2$. For example, given the array

$$\langle 3, \underline{\mathbf{1}}, \underline{\mathbf{4}}, 1, \underline{\mathbf{5}}, 9, 2, \underline{\mathbf{6}}, 5, 3, 5, \underline{\mathbf{8}}, \underline{\mathbf{9}}, 7, 9, 3, 2, 3, 8, 4, 6, 2, 7\rangle$$

your algorithm should return the integer 6, because $\langle 1, 4, 5, 6, 8, 9\rangle$ is a longest increasing subsequence (one of many).

2. Given an array $A[1..n]$ of integers, compute the length of a ***longest decreasing subsequence***. A sequence $B[1..\ell]$ is *decreasing* if $B[i] < B[i-1]$ for every index $i \geq 2$. For example, given the array

$$\langle 3, 1, 4, 1, 5, \underline{\mathbf{9}}, 2, \underline{\mathbf{6}}, 5, 3, \underline{\mathbf{5}}, 8, 9, 7, 9, 3, 2, 3, 8, \underline{\mathbf{4}}, 6, \underline{\mathbf{2}}, 7\rangle$$

your algorithm should return the integer 5, because $\langle 9, 6, 5, 4, 2\rangle$ is a longest decreasing subsequence (one of many).

3. Given an array $A[1..n]$ of integers, compute the length of a ***longest alternating subsequence***. A sequence $B[1..\ell]$ is *alternating* if $B[i] < B[i-1]$ for every even index $i \geq 2$, and $B[i] > B[i-1]$ for every odd index $i \geq 3$. For example, given the array

$$\langle \underline{\mathbf{3}}, \underline{\mathbf{1}}, \underline{\mathbf{4}}, \underline{\mathbf{1}}, \underline{\mathbf{5}}, 9, \underline{\mathbf{2}}, \underline{\mathbf{6}}, \underline{\mathbf{5}}, 3, 5, \underline{\mathbf{8}}, 9, \underline{\mathbf{7}}, \underline{\mathbf{9}}, \underline{\mathbf{3}}, 2, 3, \underline{\mathbf{8}}, \underline{\mathbf{4}}, \underline{\mathbf{6}}, \underline{\mathbf{2}}, \underline{\mathbf{7}}\rangle$$

your algorithm should return the integer 17, because $\langle 3, 1, 4, 1, 5, 2, 6, 5, 8, 7, 9, 3, 8, 4, 6, 2, 7\rangle$ is a longest alternating subsequence (one of many).

A ***subsequence*** of a sequence (for example, an array, a linked list, or a string), obtained by removing zero or more elements and keeping the rest in the same sequence order. A subsequence is called a ***substring*** if its elements are contiguous in the original sequence. For example:

- **SUBSEQUENCE**, **UBSEQU**, and the empty string ε are all substrings of the string **SUBSEQUENCE**;
- **SBSQNC**, **UEQUE**, and **EEE** are all subsequences of **SUBSEQUENCE** but not substrings;
- **QUEUE**, **SSS**, and **FOOBAR** are not subsequences of **SUBSEQUENCE**.

Describe and analyze ***dynamic programming*** algorithms for the following problems. For the first three, use the backtracking algorithms you developed on Wednesday.

1. Given an array $A[1..n]$ of integers, compute the length of a longest ***increasing*** subsequence of A. A sequence $B[1..\ell]$ is *increasing* if $B[i] > B[i-1]$ for every index $i \geq 2$.

2. Given an array $A[1..n]$ of integers, compute the length of a longest ***decreasing*** subsequence of A. A sequence $B[1..\ell]$ is *decreasing* if $B[i] < B[i-1]$ for every index $i \geq 2$.

3. Given an array $A[1..n]$ of integers, compute the length of a longest ***alternating*** subsequence of A. A sequence $B[1..\ell]$ is *alternating* if $B[i] < B[i-1]$ for every even index $i \geq 2$, and $B[i] > B[i-1]$ for every odd index $i \geq 3$.

4. Given an array $A[1..n]$ of integers, compute the length of a longest ***convex*** subsequence of A. A sequence $B[1..\ell]$ is *convex* if $B[i] - B[i-1] > B[i-1] - B[i-2]$ for every index $i \geq 3$.

5. Given an array $A[1..n]$, compute the length of a longest ***palindrome*** subsequence of A. Recall that a sequence $B[1..\ell]$ is a *palindrome* if $B[i] = B[\ell - i + 1]$ for every index i.

Basic steps in developing a dynamic programming algorithm

1. **Formulate the problem recursively.** This is the hard part. There are two distinct but equally important things to include in your formulation.

   (a) **Specification.** First, give a clear and precise English description of the problem you are claiming to solve. Don't describe *how* to solve the problem at this stage; just describe *what* the problem actually is. Otherwise, the reader has no way to know what your recursive algorithm is *supposed* to compute.

   (b) **Solution.** Second, give a clear recursive formula or algorithm for the whole problem in terms of the answers to smaller instances of *exactly* the same problem. It generally helps to think in terms of a recursive definition of your inputs and outputs. If you discover that you need a solution to a *similar* problem, or a slightly *related* problem, you're attacking the wrong problem; go back to step 1.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

   (a) **Identify the subproblems.** What are all the different ways can your recursive algorithm call itself, starting with some initial input? For example, the argument to RecFibo is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2}, we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

   (d) **Identify dependencies between subproblems.** Except for the base cases, every recursive subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

   (e) **Find a good evaluation order.** Order the subproblems so that each subproblem comes *after* the subproblems it depends on. Typically, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. ***Be careful!***

   (f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence.

1. It’s almost time to show off your flippin’ sweet dancing skills! Tomorrow is the big dance contest you’ve been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You’ve obtained an advance copy of the the list of n songs that the judges will play during the contest, in chronological order.

   You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly $Score[k]$ points, but then you will be physically unable to dance for the next $Wait[k]$ songs (that is, you cannot dance to songs $k+1$ through $k+Wait[k]$). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

   Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.

2. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string **BANANAANANAS** is a shuffle of the strings **BANANA** and **ANANAS** in several different ways.

   $${}^{\texttt{BANANA}}{}_{\texttt{ANANAS}} \qquad {}^{\texttt{BAN}}{}_{\texttt{ANA}}{}^{\texttt{ANA}}{}_{\texttt{NAS}} \qquad {}^{\texttt{B}}{}_{\texttt{AN}}{}^{\texttt{AN}}{}_{\texttt{A}}{}^{\texttt{A}}{}_{\texttt{NA}}{}^{\texttt{NA}}{}_{\texttt{S}}$$

   Similarly, the strings **PRODGYRNAMAMMIINCG** and **DYPRONGARMAMMICING** are both shuffles of **DYNAMIC** and **PROGRAMMING**:

   $${}_{\texttt{PRO}}{}^{\texttt{D}}{}_{\texttt{G}}{}^{\texttt{Y}}{}_{\texttt{R}}{}^{\texttt{NAM}}{}_{\texttt{AMMI}}{}^{\texttt{I}}{}_{\texttt{N}}{}^{\texttt{C}}{}_{\texttt{G}} \qquad {}^{\texttt{DY}}{}_{\texttt{PRO}}{}^{\texttt{N}}{}_{\texttt{G}}{}^{\texttt{A}}{}_{\texttt{R}}{}^{\texttt{M}}{}_{\texttt{AMM}}{}^{\texttt{IC}}{}_{\texttt{ING}}$$

   Describe and analyze an efficient algorithm to determine, given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, whether C is a shuffle of A and B.

Basic steps in developing a dynamic programming algorithm

1. **Formulate the problem recursively.** This is the hard part. There are two distinct but equally important things to include in your formulation.

   (a) **Specification.** First, give a clear and precise English description of the problem you are claiming to solve. Don't describe *how* to solve the problem at this stage; just describe *what* the problem actually is. Otherwise, the reader has no way to know what your recursive algorithm is *supposed* to compute.

   (b) **Solution.** Second, give a clear recursive formula or algorithm for the whole problem in terms of the answers to smaller instances of *exactly* the same problem. It generally helps to think in terms of a recursive definition of your inputs and outputs. If you discover that you need a solution to a *similar* problem, or a slightly *related* problem, you're attacking the wrong problem; go back to step 1.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

   (a) **Identify the subproblems.** What are all the different ways can your recursive algorithm call itself, starting with some initial input? For example, the argument to RecFibo is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2}, we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

   (d) **Identify dependencies between subproblems.** Except for the base cases, every recursive subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

   (e) **Find a good evaluation order.** Order the subproblems so that each subproblem comes *after* the subproblems it depends on. Typically, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. ***Be careful!***

   (f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence.

1. Suppose you are given a sequence of non-negative integers separated by + and × signs; for example:

$$2 \times 3 + 0 \times 6 \times 1 + 4 \times 2$$

You can change the value of this expression by adding parentheses in different places. For example:

$$\begin{aligned}
&2 \times (3 + (0 \times (6 \times (1 + (4 \times 2))))) = 6 \\
&(((((2 \times 3) + 0) \times 6) \times 1) + 4) \times 2 = 80 \\
&((2 \times 3) + (0 \times 6)) \times (1 + (4 \times 2)) = 108 \\
&(((2 \times 3) + 0) \times 6) \times ((1 + 4) \times 2) = 360
\end{aligned}$$

Describe and analyze an algorithm to compute, given a list of integers separated by + and × signs, the largest possible value we can obtain by inserting parentheses.

Your input is an array $A[0..2n]$ where each $A[i]$ is an integer if i is even and + or × if i is odd. Assume any arithmetic operation in your algorithm takes $O(1)$ time.

Basic steps in developing a dynamic programming algorithm

1. **Formulate the problem recursively.** This is the hard part. There are two distinct but equally important things to include in your formulation.

   (a) **Specification.** First, give a clear and precise English description of the problem you are claiming to solve. Don't describe *how* to solve the problem at this stage; just describe *what* the problem actually is. Otherwise, the reader has no way to know what your recursive algorithm is *supposed* to compute.

   (b) **Solution.** Second, give a clear recursive formula or algorithm for the whole problem in terms of the answers to smaller instances of *exactly* the same problem. It generally helps to think in terms of a recursive definition of your inputs and outputs. If you discover that you need a solution to a *similar* problem, or a slightly *related* problem, you're attacking the wrong problem; go back to step 1.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

   (a) **Identify the subproblems.** What are all the different ways can your recursive algorithm call itself, starting with some initial input? For example, the argument to RecFibo is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2}, we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

   (d) **Identify dependencies between subproblems.** Except for the base cases, every recursive subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

   (e) **Find a good evaluation order.** Order the subproblems so that each subproblem comes *after* the subproblems it depends on. Typically, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. ***Be careful!***

   (f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence.

Recall the class scheduling problem described in lecture on Tuesday. We are given two arrays $S[1..n]$ and $F[1..n]$, where $S[i] < F[i]$ for each i, representing the start and finish times of n classes. Your goal is to find the largest number of classes you can take without ever taking two classes simultaneously. We showed in class that the following greedy algorithm constructs an optimal schedule:

Choose the course that *ends first*, discard all conflicting classes, and recurse.

But this is not the only greedy strategy we could have tried. For each of the following alternative greedy algorithms, either prove that the algorithm always constructs an optimal schedule, or describe a small input example for which the algorithm does not produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control).

[Hint: Exactly three of these greedy strategies actually work.]

1. Choose the course x that *ends last*, discard classes that conflict with x, and recurse.

2. Choose the course x that *starts first*, discard all classes that conflict with x, and recurse.

3. Choose the course x that *starts last*, discard all classes that conflict with x, and recurse.

4. Choose the course x with *shortest duration*, discard all classes that conflict with x, and recurse.

5. Choose a course x that *conflicts with the fewest other courses*, discard all classes that conflict with x, and recurse.

6. If no classes conflict, choose them all. Otherwise, discard the course with *longest duration* and recurse.

7. If no classes conflict, choose them all. Otherwise, discard a course that *conflicts with the most other courses* and recurse.

8. Let x be the class with the *earliest start time*, and let y be the class with the *second earliest start time*.
   - If x and y are disjoint, choose x and recurse on everything but x.
   - If x completely contains y, discard x and recurse.
   - Otherwise, discard y and recurse.

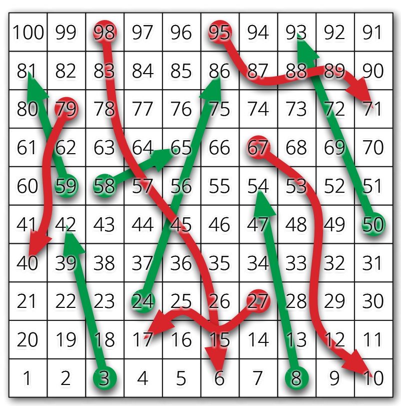
9. If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that *ends last*, discard all classes that conflict with y, and recurse.

For each of the problems below, transform the input into a graph and apply a standard graph algorithm that you've seen in class. Whenever you use a standard graph algorithm, you ***must*** provide the following information. (I recommend actually using a bulleted list.)

- What are the vertices?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that problem?
- What is the running time of your entire algorithm, *including* the time to build the graph, *as a function of the original input parameters*?

---

1. ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.



A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k (typically 6). If the token ends the move at the *top* end of a snake, you ***must*** slide the token down to the bottom of that snake. If the token ends the move at the *bottom* end of a ladder, you ***may*** move the token up to the top of that ladder.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

2. Let G be a connected undirected graph. Suppose we start with two coins on two arbitrarily chosen vertices of G. At every step, each coin *must* move to an adjacent vertex. Describe and analyze an algorithm to compute the minimum number of steps to reach a configuration where both coins are on the same vertex, or to report correctly that no such configuration is reachable. The input to your algorithm consists of a graph $G = (V, E)$ and two vertices $u, v \in V$ (which may or may not be distinct).

For each of the problems below, transform the input into a graph and apply a standard graph algorithm that you've seen in class. Whenever you use a standard graph algorithm, you ***must*** provide the following information. (I recommend actually using a bulleted list.)

- What are the vertices?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that problem?
- What is the running time of your entire algorithm, *including* the time to build the graph, *as a function of the original input parameters*?

---

1. Inspired by the previous lab, you decided to organize a Snakes and Ladders competition with n participants. In this competition, each game of Snakes and Ladders involves three players. After the game is finished, they are ranked first, second and third. Each player may be involved in any (non-negative) number of games, and the number needs not be equal among players.

   At the end of the competition, m games have been played. You realized that you had forgotten to implement a proper rating system, and therefore decided to produce the overall ranking of all n players as you see fit. However, to avoid being too suspicious, if player A ranked better than player B in any game, then A must rank better than B in the overall ranking.

   You are given the list of players involved and the ranking in each of the m games. Describe and analyze an algorithm to produce an overall ranking of the n players that satisfies the condition, or correctly reports that it is impossible.

2. There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way e has an associated toll of c_e dollars, where c_e is a positive integer. A teleport-way can be used multiple times, but the toll must be paid every time it is used.

   Judy wants to travel from galaxy u to galaxy v, but teleportation is not very pleasant and she would like to minimize the number of times she needs to teleport. However, she wants the total cost to be a multiple of five dollars, because carrying small bills is not pleasant either.

   (a) Describe and analyze an algorithm to compute the smallest number of times Judy needs to teleport to travel from galaxy u to galaxy v while the total cost is a multiple of five dollars.

   (b) Solve (a), but now assume that Judy has a coupon that allows her to waive the toll once.

Suppose we are given both an undirected graph G with weighted edges and a minimum spanning tree T of G.

1. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \in T$ is decreased.

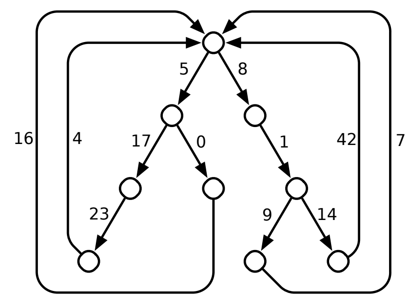
2. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \notin T$ is increased.

3. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \in T$ is increased.

4. Describe an efficient algorithm to update the minimum spanning tree when the weight of one edge $e \notin T$ is decreased.

In all cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. Of course, we could just recompute the minimum spanning tree from scratch in $O(E \log V)$ time, but you can do better.

1. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has non-negative weight.



A looped tree.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?

   (b) Describe and analyze a faster algorithm.

2. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

   Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights. *[Hint: Build an appropriate graph from the input data and apply Dijkstra's algorithm.]*

Describe Turing machines that compute the following functions.

In particular, specify the transition functions $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ for each machine either by writing out a table or by drawing a graph. Recall that $\delta(p, \$) = (q, @, +1)$ means that if the Turing machine is in state p and reads the symbol $ from the tape, then it will change to state q, write the symbol @ to the tape, and move one step to the right. In a *drawing* of a Turing machine, this transition is indicated by an edge from p to q with the label "$/@, +1".

Give your states short mnemonic names that suggest their purpose. Naming your states well won't just make it easier to understand; it will also make it easier to design.

---

1. DOUBLE: Given a string $w \in \{0, 1\}^*$ as input, return the string ww as output.

2. POWER: Given a string of the form 1^n as input, return the string 1^{2^n} as output.

Describe how to simulate an arbitrary Turing machine to make it ***error-tolerant***. Specifically, given an arbitrary Turing machine M, describe a new Turing machine M' that accepts and rejects exactly the same strings as M, even though an evil pixie named Lenny will move the head of M' to an ***arbitrary*** location on the tape some finite number of *unknown* times during the execution of M'.

You do not have to describe M' in complete detail, but do give enough details that a seasoned Turing machine programmer could work out the remaining mechanical details.

As stated, this problem has no solution! If M halts on all inputs after a finite number of steps, then Lenny can make any substring of the input string completely invisible to M. For example, if the true input string is `INPUT-STRING`, Lenny can make M believe the input string is actually `IMPING`, by moving the head to the second `I` whenever it tries to move to `R`, and by moving the head to `P` when it tries to move to `U`. Because M halts after a finite number of steps, Lenny only has a finite number of opportunities to move the head.

In fact, with more care, Lenny can make M think the input string is *any* string that uses only symbols from the actual input string; if the true input string is `INPUT-STRING`, Lenny can make M believe the input string is actually `GRINNING-PUTIN-IS-GRINNING`.)

However, there are several different ways to rescue the problem. For each of the following restrictions on Lenny's behavior, and for any Turing machine M, one can design a Turing machine M' that simulates M despite Lenny's interference.

- Lenny can move the head only a *bounded* number of times. For example: Lenny can move the head at most 374 times.
- Whenever Lenny moves the head, he changes the state of the machine to a special error state `lenny`.
- Whenever Lenny moves the head, he moves it to the left end of the tape.
- Whenever Lenny moves the head, he moves it to a blank cell to the right of all non-blank cells.
- Whenever Lenny moves the head, he moves it to a cell containing a particular symbol in the input alphabet, say `0`.

Describe algorithms for the following problems. The input for each problem is string $\langle M, w\rangle$ that encodes a standard (one-tape, one-track, one-head) Turing machine M whose tape alphabet is $\{\mathtt{0}, \mathtt{1}, \square\}$ and a string $w \in \{\mathtt{0}, \mathtt{1}\}^*$.

1. Does M accept w after at most $|w|^2$ steps?

2. If we run M with input w, does M ever move its head to the right?

2½. If we run M with input w, does M ever move its head to the right twice in a row?

2¾. If we run M with input w, does M move its head to the right more than $2^{|w|}$ times?

3. If we run M with input w, does M ever change a symbol on the tape?

3½. If we run M with input w, does M ever change a $\square$ on the tape to either $\mathtt{0}$ or $\mathtt{1}$?

4. If we run M with input w, does M ever leave its start state?

In contrast, as we will see later, the following problems are all undecidable!

1. Does M accept w?

1½. If we run M with input w, does M ever halt?

2. If we run M with input w, does M ever move its head to the right three times in a row?

3. If we run M with input w, does M ever change a $\square$ on the tape to $\mathtt{1}$?

3½. If we run M with input w, does M ever change either $\mathtt{0}$ or $\mathtt{1}$ on the tape to $\square$?

4. If we run M with input w, does M ever reenter its start state?

1. Suppose you are given a magic black box that somehow answers the following decision problem in *polynomial time*:

    - INPUT: A boolean circuit K with n inputs and one output .
    - OUTPUT: TRUE if there are input values $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ that make K output TRUE, and FALSE otherwise.

    Using this black box as a subroutine, describe an algorithm that solves the following related search problem *in polynomial time*:

    - INPUT: A boolean circuit K with n inputs and one output.
    - OUTPUT: Input values $x_1, x_2, \ldots, x_n \in \{\text{TRUE}, \text{FALSE}\}$ that make K output TRUE, or NONE if there are no such inputs.

    [Hint: You can use the magic box more than once.]

2. Formally, ***valid 3-coloring*** of a graph $G = (V, E)$ is a function $c : V \to \{1, 2, 3\}$ such that $c(u) \neq c(v)$ for all $uv \in E$. Less formally, a valid 3-coloring assigns each vertex a color, which is either red, green, or blue, such that the endpoints of every edge have different colors.

    Suppose you are given a magic black box that somehow answers the following problem *in polynomial time*:

    - INPUT: An undirected graph G.
    - OUTPUT: TRUE if G has a valid 3-coloring, and FALSE otherwise.

    Using this black box as a subroutine, describe an algorithm that solves the **3-*coloring problem*** *in polynomial time*:

    - INPUT: An undirected graph G.
    - OUTPUT: A valid 3-coloring of G, or NONE if there is no such coloring.

    *[Hint: You can use the magic box more than once. The input to the magic box is a graph and **only** a graph, meaning **only** vertices and edges.]*

Proving that a problem X is NP-hard requires several steps:

- Choose a problem Y that you already know is NP-hard.
- Describe an algorithm to solve Y, using an algorithm for X as a subroutine. Typically this algorithm has the following form: Given an instance of Y, transform it into an instance of X, and then call the magic black-box algorithm for X.
- Prove that your algorithm is correct. This almost always requires two separate steps:
  - Prove that your algorithm transforms "good" instances of Y into "good" instances of X.
  - Prove that your algorithm transforms "bad" instances of Y into "bad" instances of X. Equivalently: Prove that if your transformation produces a "good" instance of X, then it was given a "good" instance of Y.
- Argue that your algorithm for Y runs in polynomial time.

---

1. Recall the following kColor problem: Given an undirected graph G, can its vertices be colored with k colors, so that every edge touches vertices with two different colors?

   (a) Describe a direct polynomial-time reduction from 3Color to 4Color.

   (b) Prove that kColor problem is NP-hard for any $k \geq 3$.

2. Recall that a *Hamiltonian cycle* in a graph G is a cycle that goes through every vertex of G exactly once. Now, a ***tonian cycle*** in a graph G is a cycle that goes through at least *half* of the vertices of G, and a ***Hamilhamiltonian circuit*** in a graph G is a closed walk that goes through every vertex in G exactly *twice*.

   (a) Prove that it is NP-hard to determine whether a given graph contains a tonian cycle.

   (b) Prove that it is NP-hard to determine whether a given graph contains a Hamilhamiltonian circuit.

Proving that a problem X is NP-hard requires several steps:

- Choose a problem Y that you already know is NP-hard.
- Describe an algorithm to solve Y, using an algorithm for X as a subroutine. Typically this algorithm has the following form: Given an instance of Y, transform it into an instance of X, and then call the magic black-box algorithm for X.
- Prove that your algorithm is correct. This almost always requires two separate steps:
  - Prove that your algorithm transforms "good" instances of Y into "good" instances of X.
  - Prove that your algorithm transforms "bad" instances of Y into "bad" instances of X. Equivalently: Prove that if your transformation produces a "good" instance of X, then it was given a "good" instance of Y.
- Argue that your algorithm for Y runs in polynomial time.

Recall that a *Hamiltonian cycle* in a graph G is a cycle that visits every vertex of G exactly once.

1. In class on Thursday, Jeff proved that it is NP-hard to determine whether a given *directed* graph contains a Hamiltonian cycle. Prove that it is NP-hard to determine whether a given *undirected* graph contains a Hamiltonian cycle.

2. A *double Hamiltonian circuit* in a graph G is a closed walk that goes through every vertex in G exactly *twice*. Prove that it is NP-hard to determine whether a given *undirected* graph contains a double Hamiltonian circuit.

Proving that a language L is undecidable by reduction requires several steps:

- Choose a language L' that you already know is undecidable. Typical choices for L' include:

$$\begin{aligned}
\textsc{Accept} &:= \left\{\langle M,w\rangle \;\middle|\; M \text{ accepts } w\right\}\\
\textsc{Reject} &:= \left\{\langle M,w\rangle \;\middle|\; M \text{ rejects } w\right\}\\
\textsc{Halt} &:= \left\{\langle M,w\rangle \;\middle|\; M \text{ halts on } w\right\}\\
\textsc{Diverge} &:= \left\{\langle M,w\rangle \;\middle|\; M \text{ diverges on } w\right\}\\
\textsc{NeverAccept} &:= \left\{\langle M\rangle \mid \textsc{Accept}(M)=\varnothing\right\}\\
\textsc{NeverReject} &:= \left\{\langle M\rangle \mid \textsc{Reject}(M)=\varnothing\right\}\\
\textsc{NeverHalt} &:= \left\{\langle M\rangle \mid \textsc{Halt}(M)=\varnothing\right\}\\
\textsc{NeverDiverge} &:= \left\{\langle M\rangle \mid \textsc{Diverge}(M)=\varnothing\right\}
\end{aligned}$$

- Describe an algorithm (really a Turing machine) M' that decides L', using a Turing machine M that decides L as a black box. Typically this algorithm has the following form:

  Given a string w, transform it into another string x,
  such that M accepts x if and only if $w \in L'$.

- Prove that your Turing machine is correct. This almost always requires two separate steps:
  - Prove that if M accepts w then $w \in L'$.
  - Prove that if M rejects w then $w \notin L'$.

Prove that the following languages are undecidable:

1. $\textsc{AcceptIllini} := \left\{\langle M\rangle \;\middle|\; M \text{ accepts the string } \texttt{ILLINI}\right\}$
2. $\textsc{AcceptThree} := \left\{\langle M\rangle \;\middle|\; M \text{ accepts exactly three strings}\right\}$
3. $\textsc{AcceptPalindrome} := \left\{\langle M\rangle \;\middle|\; M \text{ accepts at least one palindrome}\right\}$

Prove that the following languages are undecidable *using Rice's Theorem*:

Rice's Theorem. *Let $\mathscr{X}$ be any nonempty proper subset of the set of acceptable languages. The language* $\textsc{AcceptIn}\mathscr{X} := \left\{\langle M\rangle \,\middle|\, \textsc{Accept}(M) \in \mathscr{X}\right\}$ *is undecidable.*

1. $\textsc{AcceptRegular} := \left\{\langle M\rangle \,\middle|\, \textsc{Accept}(M) \text{ is regular}\right\}$
2. $\textsc{AcceptIllini} := \left\{\langle M\rangle \,\middle|\, M \text{ accepts the string } \texttt{ILLINI}\right\}$
3. $\textsc{AcceptPalindrome} := \left\{\langle M\rangle \,\middle|\, M \text{ accepts at least one palindrome}\right\}$
4. $\textsc{AcceptThree} := \left\{\langle M\rangle \,\middle|\, M \text{ accepts exactly three strings}\right\}$
5. $\textsc{AcceptUndecidable} := \left\{\langle M\rangle \,\middle|\, \textsc{Accept}(M) \text{ is undecidable }\right\}$

To think about later. Which of the following languages are undecidable? How do you prove it?

1. $\textsc{Accept}\{\{\varepsilon\}\} := \left\{\langle M\rangle \,\middle|\, M \text{ only accepts the string } \varepsilon\text{, i.e. } \textsc{Accept}(M) = \{\varepsilon\}\right\}$
2. $\textsc{Accept}\{\varnothing\} := \left\{\langle M\rangle \,\middle|\, M \text{ does not accept any strings, i.e. } \textsc{Accept}(M) = \varnothing\right\}$
3. $\textsc{Accept}\varnothing := \left\{\langle M\rangle \,\middle|\, \textsc{Accept}(M) \text{ is not an acceptable language}\right\}$
4. $\textsc{Accept=Reject} := \left\{\langle M\rangle \,\middle|\, \textsc{Accept}(M) = \textsc{Reject}(M) \right\}$
5. $\textsc{Accept}{\neq}\textsc{Reject} := \left\{\langle M\rangle \,\middle|\, \textsc{Accept}(M) \neq \textsc{Reject}(M) \right\}$
6. $\textsc{Accept}{\cup}\textsc{Reject} := \left\{\langle M\rangle \,\middle|\, \textsc{Accept}(M) \cup \textsc{Reject}(M) = \Sigma^* \right\}$

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "True" if the statement is ***always*** true and "False" otherwise. Each correct answer is worth +1 point; each incorrect answer is worth −½ point; checking "I don't know" is worth +¼ point; and flipping a coin is (on average) worth +¼ point. You do ***not*** need to prove your answer is correct.

   Read each statement *very* carefully. Some of these are deliberately subtle.

   (a) If $2+2=5$, then Jeff is the Queen of England.

   (b) For all languages L_1 and L_2, the language $L_1 \cup L_2$ is regular.

   (c) For all languages $L \subseteq \Sigma^*$, if L is not regular, then $\Sigma^* \setminus L$ cannot be represented by a regular expression.

   (d) For all languages L_1 and L_2, if $L_1 \subseteq L_2$ and L_1 is regular, then L_2 is regular.

   (e) For all languages L_1 and L_2, if $L_1 \subseteq L_2$ and L_1 is not regular, then L_2 is not regular.

   (f) For all languages L, if L is regular, then L has no infinite fooling set.

   (g) The language $\left\{ \mathtt{0}^m \mathtt{1}^n \;\middle|\; 0 \le m+n \le 374 \right\}$ is regular.

   (h) The language $\left\{ \mathtt{0}^m \mathtt{1}^n \;\middle|\; 0 \le m-n \le 374 \right\}$ is regular.

   (i) For every language L, if the language $L^R = \left\{ w^R \;\middle|\; w \in L \right\}$ is regular, then L is also regular. (Here w^R denotes the reversal of string w; for example, $(\mathtt{BACKWARD})^R = \mathtt{DRAWKCAB}$.)

   (j) Every context-free language is regular.

2. Let L be the set of strings in $\{\mathtt{0}, \mathtt{1}\}^*$ in which every run of consecutive $\mathtt{0}$s has even length and every run of consecutive $\mathtt{1}$s has odd length.

   (a) Give a regular expression that represents L.

   (b) Construct a DFA that recognizes L.

   You do ***not*** need to prove that your answers are correct.

3. For each of the following languages over the alphabet $\{\mathtt{0}, \mathtt{1}\}$, either ***prove*** that the language is regular or ***prove*** that the language is not regular. ***Exactly one of these two languages is regular.***

   (a) The set of all strings in which the substrings $\mathtt{00}$ and $\mathtt{11}$ appear the same number of times.

   (b) The set of all strings in which the substrings $\mathtt{01}$ and $\mathtt{10}$ appear the same number of times.

   For example, both of these languages contain the string $\mathtt{1100001101101}$.

4. Consider the following recursive function:

$$stutter(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ aa \cdot stutter(x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

For example, $stutter(\texttt{00101}) = \texttt{0000110011}$.

Prove that for any regular language L, the following languages are also regular.

(a) $\textsc{Stutter}(L) := \{stutter(w) \mid w \in L\}$.

(b) $\textsc{Stutter}^{-1}(L) := \{w \mid stutter(w) \in L\}$.

5. Recall that string concatenation and string reversal are formally defined as follows:

$$w \bullet y := \begin{cases} y & \text{if } w = \varepsilon \\ a \cdot (x \bullet y) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

Prove that $(w \bullet x)^R = x^R \bullet w^R$, for all strings w and x. Your proof should be complete, concise, formal, and self-contained.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. For each statement below, check "True" if the statement is ***always*** true and "False" otherwise. Each correct answer is worth +1 point; each incorrect answer is worth −½ point; checking "I don't know" is worth +¼ point; and flipping a coin is (on average) worth +¼ point. You do ***not*** need to prove your answer is correct.

   Read each statement *very* carefully. Some of these are deliberately subtle.

   (a) If $2+2=5$, then Jeff is not the Queen of England.

   (b) For all languages L, the language L^* is regular.

   (c) For all languages $L \subseteq \Sigma^*$, if L is can be represented by a regular expression, then $\Sigma^* \setminus L$ can also be represented by a regular expression.

   (d) For all languages L_1 and L_2, if L_2 is regular and $L_1 \subseteq L_2$, then L_1 is regular.

   (e) For all languages L_1 and L_2, if L_2 is not regular and $L_1 \subseteq L_2$, then L_1 is not regular.

   (f) For all languages L, if L is not regular, then every fooling set for L is infinite.

   (g) The language $\left\{\mathtt{0}^m\mathtt{1}\mathtt{0}^n \;\middle|\; 0 \le n-m \le 374\right\}$ is regular.

   (h) The language $\left\{\mathtt{0}^m\mathtt{1}\mathtt{0}^n \;\middle|\; 0 \le n+m \le 374\right\}$ is regular.

   (i) For every language L, if L is not regular, then the language $L^R = \left\{w^R \;\middle|\; w \in L\right\}$ is also not regular. (Here w^R denotes the reversal of string w; for example, $(\mathtt{BACKWARD})^R = \mathtt{DRAWKCAB}$.)

   (j) Every context-free language is regular.

2. Let L be the set of strings in $\{\mathtt{0},\mathtt{1}\}^*$ in which every run of consecutive $\mathtt{0}$s has odd length and the total number of $\mathtt{1}$s is even.

   For example, the string $\mathtt{11110000010111000}$ is in L, because it has eight $\mathtt{1}$s and three runs of consecutive $\mathtt{0}$s, with lengths 5, 1, and 3.

   (a) Give a regular expression that represents L.

   (b) Construct a DFA that recognizes L.

   You do ***not*** need to prove that your answers are correct.

3. For each of the following languages over the alphabet $\{\mathtt{0},\mathtt{1}\}$, either ***prove*** that the language is regular or ***prove*** that the language is not regular. ***Exactly one of these two languages is regular.***

   (a) The set of all strings in which the substrings $\mathtt{10}$ and $\mathtt{01}$ appear the same number of times.

   (b) The set of all strings in which the substrings $\mathtt{00}$ and $\mathtt{01}$ appear the same number of times.

   For example, both of these languages contain the string $\mathtt{1100001101101}$.

4. Consider the following recursive function:

$$odds(w) := \begin{cases} w & \text{if } |w| \leq 1 \\ a \cdot odds(x) & \text{if } w = abx \text{ for some } a, b \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

Intuitively, *odds* removes every other symbol from the input string, starting with the second symbol. For example, $odds(\texttt{0101110}) = \texttt{0010}$.

Prove that for any regular language L, the following languages are also regular.

(a) $\textsc{Odds}(L) := \left\{ odds(w) \,\middle|\, w \in L \right\}$.

(b) $\textsc{Odds}^{-1}(L) := \left\{ w \,\middle|\, odds(w) \in L \right\}$.

5. Recall that string concatenation and string reversal are formally defined as follows:

$$w \bullet y := \begin{cases} y & \text{if } w = \varepsilon \\ a \cdot (x \bullet y) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

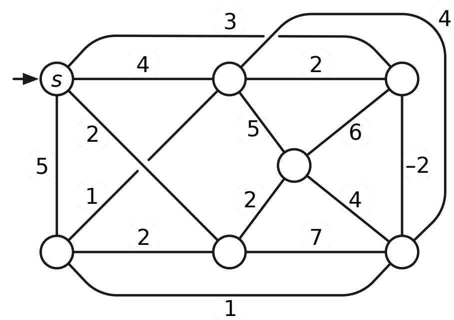
Prove that $(w \bullet x)^R = x^R \bullet w^R$, for all strings w and x. Your proof should be complete, concise, formal, and self-contained. You may assume the following identities, which we proved in class:

- $w \bullet (x \bullet y) = (w \bullet x) \bullet y$ for all strings w, x, and y.
- $|w \bullet x| = |w| + |x|$ for all strings w and x.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the edges of the following spanning trees of the weighted graph pictured below. (Pretend that the person grading your exam has bad eyesight.) Some of these subproblems have more than one correct answer. Yes, that edge on the right has negative weight.

   (a) A depth-first spanning tree rooted at s
   (b) A breadth-first spanning tree rooted at s
   (c) A shortest-path tree rooted at s
   (d) A minimum spanning tree



2. An array $A[0..n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3

is bitonic, but

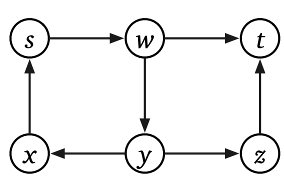
3	6	9	8	7	5	1	2	4

is *not* bitonic.

   Describe and analyze an algorithm to find the index of the *smallest* element in a given bitonic array $A[0..n-1]$ in $O(\log n)$ time. You may assume that the numbers in the input array are distinct. For example, given the first array above, your algorithm should return 6, because $A[6] = 1$ is the smallest element in that array.

3. Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Describe and analyze an algorithm to determine if there is a walk in G from s to t (possibly repeating vertices and/or edges) whose length is divisible by 3.

   For example, given the graph below, with the indicated vertices s and t, your algorithm should return TRUE, because the walk $s \to w \to y \to x \to s \to w \to t$ has length 6.



   [Hint: Build a (different) graph.]

4. The new swap-puzzle game *Candy Swap Saga XIII* involves n cute animals numbered 1 through n. Each animal holds one of three types of candy: circus peanuts, Heath bars, and Cioccolateria Gardini chocolate truffles. You also have a candy in your hand; at the start of the game, you have a circus peanut.

   To earn points, you visit each of the animals in order from 1 to n. For each animal, you can either keep the candy in your hand or exchange it with the candy the animal is holding.

   - If you swap your candy for another candy of the *same* type, you earn one point.
   - If you swap your candy for a candy of a *different* type, you lose one point. (Yes, your score can be negative.)
   - If you visit an animal and decide not to swap candy, your score does not change.

   You *must* visit the animals in order, and once you visit an animal, you can never visit it again.

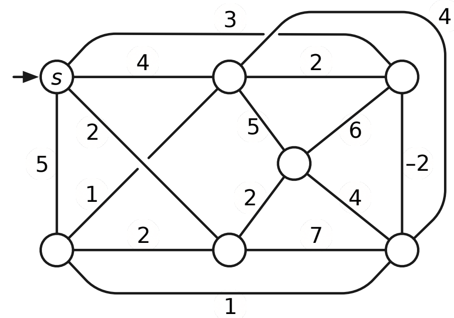
   Describe and analyze an efficient algorithm to compute your maximum possible score. Your input is an array $C[1..n]$, where $C[i]$ is the type of candy that the ith animal is holding.

5. Let G be a directed graph with weighted edges, and let s be a vertex of G. Suppose every vertex $v \neq s$ stores a pointer $pred(v)$ to another vertex in G. Describe and analyze an algorithm to determine whether these predecessor pointers correctly define a single-source shortest path tree rooted at s. Do ***not*** assume that G has no negative cycles.

Write your answers in the separate answer booklet.
Please return this question sheet and your cheat sheet with your answers.

1. ***Clearly*** indicate the edges of the following spanning trees of the weighted graph pictured below. (Pretend that the person grading your exam has bad eyesight.) Some of these subproblems have more than one correct answer. Yes, that edge on the right has negative weight.

   (a) A depth-first spanning tree rooted at s
   (b) A breadth-first spanning tree rooted at s
   (c) A shortest-path tree rooted at s
   (d) A minimum spanning tree



2. Farmers Boggis, Bunce, and Bean have set up an obstacle course for Mr. Fox. The course consists of a row of n booths, each with an integer painted on the front with bright red paint, which could be positive, negative, or zero. Let $A[i]$ denote the number painted on the front of the ith booth. Everyone has agreed to the following rules:

   - At each booth, Mr. Fox ***must*** say either "Ring!" or "Ding!".
   - If Mr. Fox says "Ring!" at the ith booth, he earns a reward of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox pays a penalty of $-A[i]$ chickens.)
   - If Mr. Fox says "Ding!" at the ith booth, he pays a penalty of $A[i]$ chickens. (If $A[i] < 0$, Mr. Fox earns a reward of $-A[i]$ chickens.)
   - Mr. Fox is forbidden to say the same word more than three times in a row. For example, if he says "Ring!" at booths 6, 7, and 8, then he *must* say "Ding!" at booth 9.
   - All accounts will be settled at the end; Mr. Fox does not actually have to carry chickens through the obstacle course.
   - If Mr. Fox violates any of the rules, or if he ends the obstacle course owing the farmers chickens, the farmers will shoot him.

   Describe and analyze an algorithm to compute the largest number of chickens that Mr. Fox can earn by running the obstacle course, given the array $A[1..n]$ of booth numbers as input.

3. Let G be a directed graph with weighted edges, and let s be a vertex of G. Suppose every vertex $v \neq s$ stores a pointer $pred(v)$ to another vertex in G. Describe and analyze an algorithm to determine whether these predecessor pointers correctly define a single-source shortest path tree rooted at s. Do ***not*** assume that G has no negative cycles.

4. An array $A[0..n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3	is bitonic, but

3	6	9	8	7	5	1	2	4	is *not* bitonic.

Describe and analyze an algorithm to find the index of the *smallest* element in a given bitonic array $A[0..n-1]$ in $O(\log n)$ time. You may assume that the numbers in the input array are distinct. For example, given the first array above, your algorithm should return 6, because $A[6] = 1$ is the smallest element in that array.

5. Suppose we are given an undirected graph G in which every *vertex* has a positive weight.

   (a) Describe and analyze an algorithm to find a *spanning tree* of G with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.)

   (b) Describe and analyze an algorithm to find a *path* in G from one given vertex s to another given vertex t with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.)

"CS 374": Algorithms and Models of Computation, Fall 2014
Final Exam — Version A — December 16, 2014

Name:	
NetID:	
Section:	1 2 3

#	1	2	3	4	5	6	Total
Score							
Max	20	10	10	10	10	10	70
Grader							

- ***Don't panic!***
- Please print your name and your NetID and circle your discussion section in the boxes above.
- This is a closed-book, closed-notes, closed-electronics exam. If you brought anything except your writing implements and your two double-sided 8½" × 11" cheat sheets, please put it away for the duration of the exam. In particular, you may not use *any* electronic devices.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear.
- **You have 180 minutes.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.** Alternatively, feel free to tear out the blank pages and use them as scratch paper.
- **Please return your cheat sheets and all scratch paper with your answer booklet.**
- If you use a greedy algorithm, you must prove that it is correct to receive credit. **Otherwise, proofs are required only if we specifically ask for them.**
- As usual, answering any (sub)problem with "I don't know" (and nothing else) is worth 25% partial credit. **Yes, even for problem 1.** Correct, complete, but suboptimal solutions are *always* worth more than 25%. A blank answer is not the same as "I don't know".
- ***Good luck!*** And have a great winter break!

---

1. For each of the following questions, indicate ***every*** correct answer by marking the "Yes" box, and indicate ***every*** incorrect answer by marking the "No" box. **Assume P $\neq$ NP.** If there is any other ambiguity or uncertainty, mark the "No" box. For example:

~~Yes~~	No	$2+2=4$
Yes	~~No~~	$x+y=5$
Yes	~~No~~	3SAT can be solved in polynomial time.
~~Yes~~	No	Jeff is not the Queen of England.

There are 40 yes/no choices altogether, each worth ½ point.

(a) Which of the following statements is true for ***every*** language $L \subseteq \{0, 1\}^*$?

Yes	No	L is non-empty.
Yes	No	L is decidable or L is infinite (or both).
Yes	No	L is accepted by some DFA with 42 states if and only if L is accepted by some NFA with 42 states.
Yes	No	If L is regular, then $L \in$ NP.
Yes	No	L is decidable if and only if its complement $\overline{L}$ is undecidable.

(b) Which of the following computational models can be simulated by a deterministic Turing machine with three read/write heads, with at most polynomial slow-down in time, assuming P $\neq$ NP?

Yes	No	A Java program
Yes	No	A deterministic Turing machine with one head
Yes	No	A deterministic Turing machine with 3 tapes, each with 5 heads
Yes	No	A nondeterministic Turing machine with one head
Yes	No	A nondeterministic finite-state automaton (NFA)

(c) Which of the following languages are decidable?

Yes	No	$\varnothing$
Yes	No	$\{0^n1^n0^n1^n \mid n \geq 0\}$
Yes	No	$\{\langle M\rangle \mid M$ is a Turing machine$\}$
Yes	No	$\{\langle M\rangle \mid M$ accepts $\langle M\rangle \bullet \langle M\rangle\}$
Yes	No	$\{\langle M\rangle \mid M$ accepts a finite number of non-palindromes$\}$
Yes	No	$\{\langle M\rangle \mid M$ accepts $\varnothing\}$
Yes	No	$\{\langle M, w\rangle \mid M$ accepts $w^R\}$
Yes	No	$\{\langle M, w\rangle \mid M$ accepts w after at most $\lvert w\rvert^2$ transitions$\}$
Yes	No	$\{\langle M, w\rangle \mid M$ changes a blank on the tape to a non-blank, given input $w\}$
Yes	No	$\{\langle M, w\rangle \mid M$ changes a non-blank on the tape to a blank, given input $w\}$

(d) Let M be a standard Turing machine (with a single one-track tape and a single head) that ***decides*** the regular language 0^*1^*. Which of the following ***must*** be true?

Yes	No	Given an empty initial tape, M eventually halts.
Yes	No	M accepts the string 1111.
Yes	No	M rejects the string 0110.
Yes	No	M moves its head to the right at least once, given input 1100.
Yes	No	M moves its head to the right at least once, given input 0101.
Yes	No	M never accepts before reading a blank.
Yes	No	For some input string, M moves its head to the left at least once.
Yes	No	For some input string, M changes at least one symbol on the tape.
Yes	No	M always halts.
Yes	No	If M accepts a string w, it does so after at most $O(\lvert w\rvert^2)$ steps.

(e) Consider the following pair of languages:

- $\text{HamiltonianPath} := \{G \mid G \text{ contains a Hamiltonian path}\}$
- $\text{Connected} := \{G \mid G \text{ is connected}\}$

Which of the following ***must*** be true, assuming P≠NP?

Yes	No	$\text{Connected} \in \text{NP}$
Yes	No	$\text{HamiltonianPath} \in \text{NP}$
Yes	No	HamiltonianPath is decidable.
Yes	No	There is no polynomial-time reduction from HamiltonianPath to Connected.
Yes	No	There is no polynomial-time reduction from Connected to HamiltonianPath.

(f) Suppose we want to prove that the following language is undecidable.

$$\text{AlwaysHalts} := \{\langle M\rangle \mid M \text{ halts on every input string}\}$$

Rocket J. Squirrel suggests a reduction from the standard halting language

$$\text{Halt} := \{\langle M, w\rangle \mid M \text{ halts on inputs } w\}.$$

Specifically, suppose there is a Turing machine AH that decides AlwaysHalts. Rocky claims that the following Turing machine H decides Halt. Given an arbitrary encoding $\langle M, w\rangle$ as input, machine H writes the encoding $\langle M'\rangle$ of a new Turing machine M' to the tape and passes it to AH, where M' implements the following algorithm:

```
M'(x):
  if M accepts w
    reject
  if M rejects w
    accept
```

Which of the following statements is true for all inputs $\langle M, w\rangle$?

Yes	No	If M accepts w, then M' halts on every input string.
Yes	No	If M diverges on w, then M' halts on every input string.
Yes	No	If M accepts w, then AH accepts $\langle M'\rangle$.
Yes	No	If M rejects w, then H rejects $\langle M, w\rangle$.
Yes	No	H decides the language Halt. (That is, Rocky's reduction is correct.)

2. A ***relaxed 3-coloring*** of a graph G assigns each vertex of G one of three colors (for example, red, green, and blue), such that ***at most one*** edge in G has both endpoints the same color.

   (a) Give an example of a graph that has a relaxed 3-coloring, but does not have a proper 3-coloring (where every edge has endpoints of different colors).

   (b) ***Prove*** that it is NP-hard to determine whether a given graph has a relaxed 3-coloring.

3. Give a complete, formal, self-contained description of a DFA that accepts all strings in $\{0, 1\}^*$ containing at least ten 0s and at most ten 1s. Specifically:

   (a) What are the states of your DFA?

   (b) What is the start state of your DFA?

   (c) What are the accepting states of your DFA?

   (d) What is your DFA's transition function?

4. Suppose you are given three strings $A[1..n]$, $B[1..n]$, and $C[1..n]$. Describe and analyze an algorithm to find the maximum length of a common subsequence of all three strings. For example, given the input strings

$$A = \texttt{AxxBxxCDxEF}, \qquad B = \texttt{yyABCDyEyFy}, \qquad C = \texttt{zAzzBCDzEFz},$$

your algorithm should output the number 6, which is the length of the longest common subsequence `ABCDEF`.

5. For each of the following languages over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, either ***prove*** that the language is regular, or ***prove*** that the language is not regular.

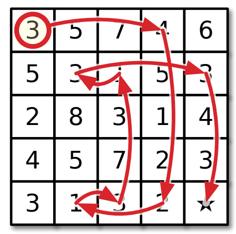
   (a) $\left\{www \;\middle|\; w \in \Sigma^*\right\}$

   (b) $\left\{wxw \;\middle|\; w, x \in \Sigma^*\right\}$

6. A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner. On each turn, you are allowed to move the token up, down, left, or right; the distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right. However, you are never allowed to move the token off the edge of the board.

   Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the maze shown below, your algorithm would return the number 8.

3	5	7	4	6
5	3	1	5	3
2	8	3	1	4
4	5	7	2	3
3	1	3	2	★



A 5×5 number maze that can be solved in eight moves.

(scratch paper)

(scratch paper)

You may assume the following problems are NP-hard:

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

HamiltonianPath: Given an undirected graph G, is there a path in G that visits every vertex exactly once?

HamiltonianCycle: Given an undirected graph G, is there a cycle in G that visits every vertex exactly once?

DirectedHamiltonianCycle: Given an directed graph G, is there a directed cycle in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G (either directed or undirected) with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

Draughts: Given an $n \times n$ international draughts configuration, what is the largest number of pieces that can (and therefore must) be captured in a single move?

Super Mario: Given an $n \times n$ level for Super Mario Brothers, can Mario reach the castle?

You may assume the following languages are undecidable:

$$\textsc{SelfReject} := \{\langle M\rangle \mid M \text{ rejects } \langle M\rangle\}$$
$$\textsc{SelfAccept} := \{\langle M\rangle \mid M \text{ accepts } \langle M\rangle\}$$
$$\textsc{SelfHalt} := \{\langle M\rangle \mid M \text{ halts on } \langle M\rangle\}$$
$$\textsc{SelfDiverge} := \{\langle M\rangle \mid M \text{ does not halt on } \langle M\rangle\}$$
$$\textsc{Reject} := \{\langle M, w\rangle \mid M \text{ rejects } w\}$$
$$\textsc{Accept} := \{\langle M, w\rangle \mid M \text{ accepts } w\}$$
$$\textsc{Halt} := \{\langle M, w\rangle \mid M \text{ halts on } w\}$$
$$\textsc{Diverge} := \{\langle M, w\rangle \mid M \text{ does not halt on } w\}$$
$$\textsc{NeverReject} := \{\langle M\rangle \mid \textsc{Reject}(M) = \varnothing\}$$
$$\textsc{NeverAccept} := \{\langle M\rangle \mid \textsc{Accept}(M) = \varnothing\}$$
$$\textsc{NeverHalt} := \{\langle M\rangle \mid \textsc{Halt}(M) = \varnothing\}$$
$$\textsc{NeverDiverge} := \{\langle M\rangle \mid \textsc{Diverge}(M) = \varnothing\}$$

"CS 374": Algorithms and Models of Computation, Fall 2014

Final Exam (Version B) — December 16, 2014

Name:	
NetID:	
Section:	1 2 3

#	1	2	3	4	5	6	Total
Score							
Max	20	10	10	10	10	10	70
Grader							

- ***Don't panic!***
- Please print your name and your NetID and circle your discussion section in the boxes above.
- This is a closed-book, closed-notes, closed-electronics exam. If you brought anything except your writing implements and your two double-sided 8½" × 11" cheat sheets, please put it away for the duration of the exam. In particular, you may not use *any* electronic devices.
- **Please read the entire exam before writing anything.** Please ask for clarification if any question is unclear.
- **You have 180 minutes.**
- If you run out of space for an answer, continue on the back of the page, or on the blank pages at the end of this booklet, **but please tell us where to look.** Alternatively, feel free to tear out the blank pages and use them as scratch paper.
- **Please return your cheat sheets and all scratch paper with your answer booklet.**
- If you use a greedy algorithm, you must prove that it is correct to receive credit. **Otherwise, proofs are required only if we specifically ask for them.**
- As usual, answering any (sub)problem with "I don't know" (and nothing else) is worth 25% partial credit. **Yes, even for problem 1.** Correct, complete, but suboptimal solutions are *always* worth more than 25%. A blank answer is not the same as "I don't know".
- ***Good luck!*** And have a great winter break!

---

1. For each of the following questions, indicate ***every*** correct answer by marking the "Yes" box, and indicate ***every*** incorrect answer by marking the "No" box. **Assume P $\neq$ NP.** If there is any other ambiguity or uncertainty, mark the "No" box. For example:

~~Yes~~	No	$2+2=4$
Yes	~~No~~	$x+y=5$
Yes	~~No~~	3SAT can be solved in polynomial time.
~~Yes~~	No	Jeff is not the Queen of England.

There are 40 yes/no choices altogether, each worth ½ point.

(a) Which of the following statements is true for ***every*** language $L \subseteq \{0,1\}^*$?

Yes	No	L is non-empty.
Yes	No	L is decidable or L is infinite (or both).
Yes	No	L is accepted by some DFA with 42 states if and only if L is accepted by some NFA with 42 states.
Yes	No	If L is regular, then $L \in$ NP.
Yes	No	L is decidable if and only if its complement $\overline{L}$ is undecidable.

(b) Which of the following computational models can simulate a deterministic Turing machine with three read/write heads, with at most polynomial slow-down in time, assuming P $\neq$ NP?

Yes	No	A C++ program
Yes	No	A deterministic Turing machine with one head
Yes	No	A deterministic Turing machine with 3 tapes, each with 5 heads
Yes	No	A nondeterministic Turing machine with one head
Yes	No	A nondeterministic finite-state automaton (NFA)

(c) Which of the following languages are decidable?

Yes	No	$\varnothing$
Yes	No	$\{ww \mid w \text{ is a palindrome}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ is a Turing machine}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts } \langle M\rangle \bullet \langle M\rangle\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts an infinite number of palindromes}\}$
Yes	No	$\{\langle M\rangle \mid M \text{ accepts } \varnothing\}$
Yes	No	$\{\langle M, w\rangle \mid M \text{ accepts } www\}$
Yes	No	$\{\langle M, w\rangle \mid M \text{ accepts } w \text{ after } \textbf{at least } \lvert w\rvert^2 \text{ transitions}\}$
Yes	No	$\{\langle M, w\rangle \mid M \text{ changes a non-blank on the tape to a blank, given input } w\}$
Yes	No	$\{\langle M, w\rangle \mid M \text{ changes a blank on the tape to a non-blank, given input } w\}$

(d) Let M be a standard Turing machine (with a single one-track tape and a single head) such that $\text{Accept}(M)$ is the regular language `0`*`1`*. Which of the following ***must*** be true?

Yes	No	Given an empty initial tape, M eventually halts.
Yes	No	M accepts the string `1111`.
Yes	No	M rejects the string `0110`.
Yes	No	M moves its head to the right at least once, given input `1100`.
Yes	No	M moves its head to the right at least once, given input `0101`.
Yes	No	M must read a blank before it accepts.
Yes	No	For some input string, M moves its head to the left at least once.
Yes	No	For some input string, M changes at least one symbol on the tape.
Yes	No	M always halts.
Yes	No	If M accepts a string w, it does so after at most $O(\lvert w\rvert^2)$ steps.

(e) Consider the following pair of languages:

- $\textsc{HamiltonianPath} := \{G \mid G \text{ contains a Hamiltonian path}\}$
- $\textsc{Connected} := \{G \mid G \text{ is connected}\}$

Which of the following ***must*** be true, assuming P≠NP?

Yes	No	$\textsc{Connected} \in \mathrm{NP}$
Yes	No	$\textsc{HamiltonianPath} \in \mathrm{NP}$
Yes	No	$\textsc{HamiltonianPath}$ is undecidable.
Yes	No	There is a polynomial-time reduction from $\textsc{HamiltonianPath}$ to $\textsc{Connected}$.
Yes	No	There is a polynomial-time reduction from $\textsc{Connected}$ to $\textsc{HamiltonianPath}$.

(f) Suppose we want to prove that the following language is undecidable.

$$\textsc{AlwaysHalts} := \{\langle M\rangle \mid M \text{ halts on every input string}\}$$

Bullwinkle J. Moose suggests a reduction from the standard halting language

$$\textsc{Halt} := \{\langle M, w\rangle \mid M \text{ halts on inputs } w\}.$$

Specifically, suppose there is a Turing machine AH that decides $\textsc{AlwaysHalts}$. Bullwinkle claims that the following Turing machine H decides $\textsc{Halt}$. Given an arbitrary encoding $\langle M, w\rangle$ as input, machine H writes the encoding $\langle M'\rangle$ of a new Turing machine M' to the tape and passes it to AH, where M' implements the following algorithm:

```
M'(x):
  if M accepts w
      reject
  if M rejects w
      accept
```

Which of the following statements is true for all inputs $\langle M, w\rangle$?

Yes	No	If M accepts w, then M' halts on every input string.
Yes	No	If M rejects w, then M' halts on every input string.
Yes	No	If M rejects w, then H rejects $\langle M, w\rangle$.
Yes	No	If M diverges on w, then H diverges on $\langle M, w\rangle$.
Yes	No	H does not correctly decide the language $\textsc{Halt}$. (That is, Bullwinkle's reduction is incorrect.)

---

2. A ***near-Hamiltonian cycle*** in a graph G is a closed walk in G that visits one vertex exactly twice and every other vertex exactly once.

   (a) Give an example of a graph that contains a near-Hamiltonian cycle, but does not contain a Hamiltonian cycle (which visits every vertex exactly once).

   (b) ***Prove*** that it is NP-hard to determine whether a given graph contains a near-Hamiltonian cycle.

3. Give a complete, formal, self-contained description of a DFA that accepts all strings in $\{0,1\}^*$ such that every fifth bit is 0 and the length is *not* divisible by 12. For example, your DFA should accept the strings 11110111101 and 11. Specifically:

   (a) What are the states of your DFA?

   (b) What is the start state of your DFA?

   (c) What are the accepting states of your DFA?

   (d) What is your DFA's transition function?

4. Suppose you are given three strings $A[1..n]$, $B[1..n]$, and $C[1..n]$. Describe and analyze an algorithm to find the maximum length of a common subsequence of all three strings. For example, given the input strings

$$A = \texttt{AxxBxxCDxEF}, \qquad B = \texttt{yyABCDyEyFy}, \qquad C = \texttt{zAzzBCDzEFz},$$

your algorithm should output the number 6, which is the length of the longest common subsequence `ABCDEF`.

5. For each of the following languages over the alphabet $\Sigma = \{0, 1\}$, either ***prove*** that the language is regular, or ***prove*** that the language is not regular.

   (a) $\left\{www \mid w \in \Sigma^*\right\}$

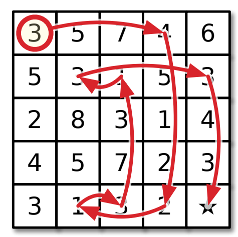
   (b) $\left\{wxw \mid w, x \in \Sigma^*\right\}$

6. A ***number maze*** is an $n \times n$ grid of positive integers. A token starts in the upper left corner; your goal is to move the token to the lower-right corner.

   - On each turn, you are allowed to move the token up, down, left, or right.
   - The distance you may move the token is determined by the number on its current square. For example, if the token is on a square labeled 3, then you may move the token three steps up, three steps down, three steps left, or three steps right.
   - However, you are never allowed to move the token off the edge of the board. In particular, if the current number is too large, you may not be able to move at all.

   Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given number maze, or correctly reports that the maze has no solution. For example, given the maze shown below, your algorithm would return the number 8.

3	5	7	4	6
5	3	1	5	3
2	8	3	1	4
4	5	7	2	3
3	1	3	2	★



A 5×5 number maze that can be solved in eight moves.

(scratch paper)

(scratch paper)

You may assume the following problems are NP-hard:

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three literals per clause, does the formula have a satisfying assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

HamiltonianPath: Given an undirected graph G, is there a path in G that visits every vertex exactly once?

HamiltonianCycle: Given an undirected graph G, is there a cycle in G that visits every vertex exactly once?

DirectedHamiltonianCycle: Given an directed graph G, is there a directed cycle in G that visits every vertex exactly once?

TravelingSalesman: Given a graph G (either directed or undirected) with weighted edges, what is the minimum total weight of any Hamiltonian path/cycle in G?

Draughts: Given an $n \times n$ international draughts configuration, what is the largest number of pieces that can (and therefore must) be captured in a single move?

Super Mario: Given an $n \times n$ level for Super Mario Brothers, can Mario reach the castle?

You may assume the following languages are undecidable:

$$\textsc{SelfReject} := \left\{\langle M\rangle \;\middle|\; M \text{ rejects } \langle M\rangle\right\}$$
$$\textsc{SelfAccept} := \left\{\langle M\rangle \;\middle|\; M \text{ accepts } \langle M\rangle\right\}$$
$$\textsc{SelfHalt} := \left\{\langle M\rangle \;\middle|\; M \text{ halts on } \langle M\rangle\right\}$$
$$\textsc{SelfDiverge} := \left\{\langle M\rangle \;\middle|\; M \text{ does not halt on } \langle M\rangle\right\}$$
$$\textsc{Reject} := \left\{\langle M, w\rangle \;\middle|\; M \text{ rejects } w\right\}$$
$$\textsc{Accept} := \left\{\langle M, w\rangle \;\middle|\; M \text{ accepts } w\right\}$$
$$\textsc{Halt} := \left\{\langle M, w\rangle \;\middle|\; M \text{ halts on } w\right\}$$
$$\textsc{Diverge} := \left\{\langle M, w\rangle \;\middle|\; M \text{ does not halt on } w\right\}$$
$$\textsc{NeverReject} := \left\{\langle M\rangle \;\middle|\; \textsc{Reject}(M) = \varnothing\right\}$$
$$\textsc{NeverAccept} := \left\{\langle M\rangle \;\middle|\; \textsc{Accept}(M) = \varnothing\right\}$$
$$\textsc{NeverHalt} := \left\{\langle M\rangle \;\middle|\; \textsc{Halt}(M) = \varnothing\right\}$$
$$\textsc{NeverDiverge} := \left\{\langle M\rangle \;\middle|\; \textsc{Diverge}(M) = \varnothing\right\}$$