1 Discrete Probability

Before I start discussing randomized *algorithms* at all, I need to give a quick formal overview of the relatively small subset of probability theory that we will actually use. The first two sections of this note are deliberately written more as a review or reference than an introduction, although they do include a few illustrative (and hopefully helpful) examples.

1.1 Discrete Probability Spaces

A ***discrete***[1] ***probability space*** **$(\Omega, \Pr)$** consists of a non-empty countable set Ω, called the ***sample space***, together with a ***probability mass function*** $\Pr : \Omega \to \mathbb{R}$ such that

$$\Pr[\omega] \geq 0 \quad \text{for all } \omega \in \Omega \qquad \text{and} \qquad \sum_{\omega \in \Omega} \Pr[\omega] = 1.$$

The latter condition implies that $\Pr[\omega] \leq 1$ for all $\omega \in \Omega$. I don't know why the probability function is written with brackets instead of parentheses, but that's the standard;[2] just go with it. Here are a few simple examples:

- A fair coin: $\Omega = \{\mathsf{heads}, \mathsf{tails}\}$ and $\Pr[\mathsf{heads}] = \Pr[\mathsf{tails}] = 1/2$.
- A fair six-sided die: $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[\omega] = 1/6$ for all $\omega \in \Omega$.
- A strangely loaded six-sided die: $\Omega = \{1, 2, 3, 4, 5, 6\}$ with $\Pr[\omega] = \omega/21$ for all $\omega \in \Omega$. (For example, $\Pr[4] = 4/21$.)

[1] Correctly defining *continuous* (or otherwise uncountable) probability spaces and continuous random variables requires considerably more care and subtlety than the discrete definitions given here. There is *no* well-defined probability measure satisfying the discrete axioms when Ω is, for instance, an interval on the real line. This way lies the Banach-Tarski paradox.

[2] Or more honestly: one of many standards

- Bart's rock-paper-scissors strategy: $\Omega = \{\mathsf{rock}, \mathsf{paper}, \mathsf{scissors}\}$ and $\Pr[\mathsf{rock}] = 1$ and $\Pr[\mathsf{paper}] = \Pr[\mathsf{scissors}] = 0$.

Other common examples of countable sample spaces include the 52 cards in a standard deck, the 52! permutations of the cards in a standard deck, the natural numbers, the integers, the rationals, the set of all (finite) bit strings, the set of all (finite) rooted trees, the set of all (finite) graphs, and the set of all (finite) execution traces of an algorithm.

The precise choice of probability space is rarely important; we can usually implicitly define Ω to be the set of all possible tuples of values, one for each random variable under discussion.

1.1.1 Events and Probability

Subsets of Ω are usually called ***events***, and individual elements of Ω are usually called ***sample points*** or ***elementary events*** or ***atoms***. However, it is often useful to think of the elements of Ω as possible *states* of a system or *outcomes* of an experiment, and subsets of Ω as *conditions* that some states/outcomes satisfy and others don't.

The probability of an event A, denoted $\mathbf{Pr}[A]$, is defined as the sum of the probabilities of its constituent sample points:

$$\Pr[A] := \sum_{\omega \in A} \Pr[\omega]$$

In particular, we have $\Pr[\varnothing] = 0$ and $\Pr[\Omega] = 1$. Here we are extending (or overloading) the function $\Pr\colon \Omega \to [0,1]$ on atoms to a function $\Pr\colon 2^\Omega \to [0,1]$ on events.

For example, suppose we roll two fair dice, one red and the other blue. The underlying probability space consists of the sample space $\Omega = \{1,2,3,4,5,6\} \times \{1,2,3,4,5,6\}$ and the probabilities $\Pr[\omega] = 1/36$ for all $\omega \in \Omega$.

- The probability of rolling two 5s is $\Pr[\{(5,5)\}] = \Pr[(5,5)] = 1/36$.
- The probability of rolling a total of 6 is

$$\Pr\big[\{(1,5),(2,4),(3,3),(4,2),(5,1)\}\big] = \frac{5}{36}.$$

- The probability that the red die shows a 5 is

$$\Pr\big[\{(5,1),(5,2),(5,3),(5,4),(5,5),(5,6)\}\big] = \frac{1}{6}.$$

- The probability that at least one die shows a 5 is

$$\Pr\big[\{(1,5),(2,5),(3,5),(4,5),(5,1),(5,2),(5,3),(5,4),(5,5),(5,6),(6,5)\}\big] = \frac{11}{36}.$$

- The probability that the red die shows a smaller number than the blue die is

$$\Pr\big[\{(1,2),(1,3),(1,4),(1,5),(1,6),\\ (2,3),(2,4),(2,5),(2,6),(3,4),\\ (3,5),(3,6),(4,5),(4,6),(5,6)\}\big] = \frac{5}{12}.$$

1.1.2 Combining Events

Because they are formally just sets, events can be combined using arbitrary set operations. However, in keeping with the intuition that events are *conditions*, these operations are usually written using Boolean logic notation $\wedge, \vee, \neg$ and vocabulary ("and, or, not") instead of the equivalent set notation $\cap, \cup, \bar{\cdot}$ and vocabulary ("intersection, union, complement"). For example, consider our earlier experiment rolling two fair six-sided dice, one red and the other blue.

$$\begin{aligned}
\Pr[\text{red 5}] &= 1/6 \\
\Pr[\text{two 5s}] &= \Pr[\text{red 5} \wedge \text{blue 5}] = 1/36 \\
\Pr[\text{at least one 5}] &= \Pr[\text{red 5} \vee \text{blue 5}] = 11/36 \\
\Pr[\text{at most one 5}] &= \Pr[\neg(\text{two 5s})] = 1 - \Pr[\text{two 5s}] = 35/36 \\
\Pr[\text{no 5s}] &= \Pr[\neg(\text{at least one 5})] \\
&= 1 - \Pr[\text{at least one 5}] = 25/36 \\
\Pr[\text{exactly one 5}] &= \Pr[\text{at least one 5} \wedge \text{at most one 5}] \\
&= \Pr[\text{red 5} \oplus \text{blue 5}] = 5/18 \\
\Pr[\text{blue 5} \Rightarrow \text{red 5}] &= \Pr[\neg(\text{blue 5}) \vee \text{red 5}] = 31/36
\end{aligned}$$

(As usual, $p \Rightarrow q$ is just shorthand for $\neg p \vee q$; implication does *not* indicate causality!)

For any two events A and B with $\Pr[B] > 0$, the ***conditional probability of A given B*** is defined as

$$\mathbf{Pr[}A \mid B\mathbf{]} := \frac{\Pr[A \wedge B]}{\Pr[B]}.$$

For example, in our earlier red-blue dice experiment:

$$\begin{aligned}
\Pr[\text{blue 5} \mid \text{red 5}] &= \Pr[\text{two 5s} \mid \text{red 5}] = 1/6 \\
\Pr[\text{at most one 5} \mid \text{red 5}] &= \Pr[\text{exactly one 5} \mid \text{red 5}] \\
&= \Pr[\neg(\text{blue 5}) \mid \text{red 5}] = 5/6 \\
\Pr[\text{at least one 5} \mid \text{at most one 5}] &= 2/7 \\
\Pr[\text{at most one 5} \mid \text{at least one 5}] &= 10/11 \\
\Pr[\text{red 5} \mid \text{at least one 5}] &= 6/11 \\
\Pr[\text{red 5} \mid \text{at most one 5}] &= 1/7 \\
\Pr[\text{blue 5} \mid \text{blue 5} \Rightarrow \text{red 5}] &= 1/31 \\
\Pr[\text{red 5} \mid \text{blue 5} \Rightarrow \text{red 5}] &= 6/31 \\
\Pr[\text{blue 5} \Rightarrow \text{red 5} \mid \text{blue 5}] &= 1/6 \\
\Pr[\text{blue 5} \Rightarrow \text{red 5} \mid \text{red 5}] &= 1 \\
\Pr[\text{blue 5} \Rightarrow \text{red 5} \mid \text{red 5} \Rightarrow \text{blue 5}] &= 26/31
\end{aligned}$$

Two events A and B are ***disjoint*** if they are disjoint as sets, meaning $A \cap B = \emptyset$. For example, in our two-dice experiment, the events "red 5" and "blue 5" and "total 5" are pairwise disjoint. Note that it is possible for $\Pr[A \wedge B] = 0$ even when the events A and B are not disjoint; consider the events "Bart plays paper" and "Bart does not play rock".

Two events A and B are ***independent*** if and only if $\Pr[A \wedge B] = \Pr[A] \cdot \Pr[B]$. For example, in our two-dice experiment, the events "red 5" and "blue 5" are independent, but "red 5" and "total 5" are not.

More generally, a countable set of events $\{A_i \mid i \in I\}$ is ***fully*** or ***mutually independent*** if and only if $\Pr[\bigwedge_{i\in I} A_i] = \prod_{i\in I} A_i$. A set of events is ***k-wise independent*** if every subset of k events is fully independent, and ***pairwise independent*** if every pair of events in the set is independent. For example, in our two-dice experiment, the events "red 5" and "blue 5" and "total 7" are pairwise independent, but not mutually independent.

1.1.3 Identities and Inequalities

Fix n arbitrary events $A_1, A_2, \ldots, A_n$ from some sample space Ω. The following observations follow immediately from similar observations about sets.

- ***Union bound:*** For *any* events $A_1, A_2, \ldots A_n$, the definition of probability implies that

$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] \leq \sum_{i=1}^{n} \Pr[A_i].$$

  The expression on the right counts each atom in the union of the events exactly once; the left summation counts each atom once for each event A_i that contains it.

- ***Disjoint union:*** If the events $A_1, A_2, \ldots A_n$ are pairwise disjoint, meaning $A_i \cap A_j = \emptyset$ for all $i \neq j$, the union bound becomes an equation:

$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] = \sum_{i=1}^{n} \Pr[A_i].$$

- ***The principle of inclusion-exclusion*** describes a simple relationship between probabilities of unions (disjunctions) and intersections (conjunctions) of arbitrary events:

$$\Pr[A \vee B] = \Pr[A] + \Pr[B] - \Pr[A \wedge B]$$

  This principle follows directly from elementary boolean algebra and the disjoint union bound:

$$\begin{aligned}
\Pr[A \vee B] + \Pr[A \wedge B] &= \Pr[(A \wedge B) \vee (A \wedge \overline{B}) \vee (\overline{A} \wedge B)] + \Pr[A \wedge B] \\
&= \big(\Pr[A \wedge B] + \Pr[A \wedge \overline{B}] + \Pr[\overline{A} \wedge B]\big) + \Pr[A \wedge B] \\
&= \big(\Pr[A \wedge B] + \Pr[A \wedge \overline{B}]\big) + \big(\Pr[\overline{A} \wedge B] + \Pr[A \wedge B]\big) \\
&= \Pr[A] + \Pr[B]
\end{aligned}$$

  Inclusion-exclusion generalizes inductively any finite number of events as follows:

$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] = 1 - \sum_{I \subseteq [1\,..\,n]} (-1)^{|I|} \Pr\left[\bigwedge_{i\in I} A_i\right]$$

- ***Independent union:*** For any pair A and B of independent events, we have

$$\begin{aligned}
\Pr[A \vee B] &= \Pr[A] + \Pr[B] - \Pr[A \wedge B] && \text{[inclusion-exclusion]} \\
&= \Pr[A] + \Pr[B] - \Pr[A]\Pr[B] && \text{[independence]} \\
&= 1 - (1 - \Pr[A])(1 - \Pr[B]).
\end{aligned}$$

  More generally, if the events $A_1, A_2, \ldots A_n$ are *mutually* independent, then

$$\Pr\left[\bigvee_{i=1}^{n} A_i\right] = 1 - \prod_{i=1}^{n} (1 - \Pr[A_i]).$$

- ***Bayes' Theorem:*** If events A and B both have non-zero probability, the definition of conditional probability immediately implies

$$\frac{\Pr[A \mid B]}{\Pr[A]} = \Pr[A \wedge B] = \frac{\Pr[B \mid A]}{\Pr[B]}.$$

and therefore

$$\Pr[A \mid B] \cdot \Pr[B] = \Pr[B \mid A] \cdot \Pr[A].$$

1.2 Random Variables

Formally, a ***random variable X*** is a function from a sample space Ω (with an associated probability measure) to some other *value set*. For example, the identity function on Ω is a random variable, as is the function that maps everything in Ω to Queen Elizabeth II, or any function mapping Ω to the real numbers. Random variables are almost universally denoted by upper-case letters.

A random variable is not random, nor is it a variable.

The value space of a random variable is commonly described either by an adjective preceding the phrase "random variable" or a noun replacing the word "variable". For example:

- A function from Ω to $\mathbb{Z}$ is called an *integer random variable* or a *random integer*.
- A function from Ω to $\mathbb{R}$ is called an *real random variable* or a *random real number*.
- A function from Ω to $\{0,1\}$ is called an *indicator random variable* or a *random bit*.

Since every integer is a real number, every integer random variable is also a real random variable; similarly, every random bit is also a random real number. Not all random variables are numerical; for example:

- A *random graph* is function from some sample space Ω to the set of all graphs.
- A *random point in the plane* is a function from some sample space Ω to $\mathbb{R}^2$.
- The function mapping every element of Ω to Queen Elizabeth II could be called a *random Queen of England*, or a *random monarch*, or (if certain unlikely conspiracy theories are taken seriously) a *random shape-shifting alien reptile*.

1.2.1 It is a fungus.

Although random variables are formally not variables at all, we typically describe and manipulate them *as if they were* variables representing unknown elements of their value sets, without referring to any particular sample space.

In particular, we can apply arbitrary functions to random variables by composition. Fir any random variable $X : \Omega \to V$ and any function $f : V \to V'$, the function $f(X) := f \circ X$ is a random variable over the value set V'. In particular, if X is a real random variable and α is any real number, then $X + \alpha$ and $\alpha \cdot X$ are also real random variables. More generally, if X and X' are random variables with value sets V and V', then for any function $f : V \times V' \to V''$, the function $f(X, X')$ is a random variable over V'', formally defined as

$$f(X, X')(\omega) := f(X(\omega), X'(\omega)).$$

These definitions extend in the obvious way to functions with an arbitrary number of arguments.

If ϕ is a boolean function or predicate over the value set of X, we implicitly identify the random variable $\phi(X)$ with the event $\{\omega \in \Omega \mid \phi(X(\omega))\}$. For example, if X is an integer random variable, then

$$\Pr[X = x] \;:=\; \Pr\big[\{\omega \mid X(\omega) = x\}\big]$$

and

$$\Pr[X \le x] \;:=\; \Pr\big[\{\omega \mid X(\omega) \le x\}\big]$$

and

$$\Pr[X \text{ is prime}] \;:=\; \Pr\big[\{\omega \mid X(\omega) \text{ is prime}\}\big].$$

In particular, we typically identify the boolean values **TRUE** and **FALSE** with the events Ω and $\emptyset$, respectively. Predicates with more than one random variable are handled similarly; for example,

$$\Pr[X = Y] \;:=\; \Pr\big[\{\omega \mid X(\omega) = Y(\omega)\}\big].$$

1.2.2 Expectation

For any real (or complex or vector) random variable X, the ***expectation of X*** is defined as

$$\mathbf{E}[X] := \sum_x x \cdot \Pr[X = x].$$

This sum is always well-defined, because the set $\{x \mid \Pr[X = x] \neq 0\} \subseteq \Omega$ is countable. For *integer* random variables, the following definition is equivalent:

$$\begin{aligned}\mathrm{E}[X] &= \sum_{x \ge 0} \Pr[X \ge x] - \sum_{x \le 0} \Pr[X \le x] \\ &= \sum_{x \ge 0} \big(\Pr[X \ge x] - \Pr[X \le -x]\big).\end{aligned}$$

If moreover A is an arbitrary event with non-zero probability, then the ***conditional expectation of X given A*** is defined as

$$\mathbf{E}[X \mid A] := \sum_x x \cdot \Pr[X = x \mid A] = \sum_x \frac{x \cdot \Pr[X = x \wedge A]}{\Pr[A]}$$

For any event A with $0 < \Pr[A] < 1$, we immediately have

$$\mathrm{E}[X] \;=\; \mathrm{E}[X \mid A] \cdot \Pr[A] \;+\; \mathrm{E}[X \mid \neg A] \cdot \Pr[\neg A].$$

In particular, for any random variables X and Y, we have

$$\mathrm{E}[X] = \sum_y \mathrm{E}[X \mid Y = y] \cdot \Pr[Y = y].$$

Two random variables X and Y are ***independent*** if, for all x and y, the events $X = x$ and $Y = y$ are independent. If X and Y are independent real random variables, then $\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y]$. (However, this equation does *not* imply that X and Y are independent.) We can extend the notions of full, k-wise, and pairwise independence from events to random variables in similar fashion. In particular, if $X_1, X_2, \ldots, x_n$ are *fully independent* real random variables, then

$$\mathrm{E}\left[\prod_{i=1}^{n} X_i\right] = \prod_{i=1}^{n} \mathrm{E}[X_i].$$

Linearity of expectation refers to the following important fact: The expectation of any weighted sum of random variables is equal to the weighted sum of the expectations of those variables. More formally, for any real random variables $X_1, X_2, \ldots, X_n$ and any real coefficients $\alpha_1, \alpha_2, \ldots, \alpha_n$,

$$\mathrm{E}\left[\sum_{i=1}^{n}(\alpha_i \cdot X_i)\right] = \sum_{i=1}^{n}\big(\alpha_i \cdot \mathrm{E}[X_i]\big).$$

Linearity of expectation does *not* require the variables to be independent.

1.2.3 Examples

Consider once again our experiment with two standard fair six-sided dice, one red and the other blue. We define several random variables:

- R is the value (on the top face) of the red die.
- B is the value (on the top face) of the blue die.
- $S = R + B$ is the total value (on the top faces) of both dice.
- $\mathcal{U} = 7 - R$ is the value on the *bottom* face of the red die.

The variables R and B are independent, as are the variables $\mathcal{U}$ and B, but no other pair of these variables is independent.

$$\begin{aligned}
\mathrm{E}[R] = \mathrm{E}[B] = \mathrm{E}[\mathcal{U}] &= \frac{1+2+3+4+5+6}{6} = \frac{7}{2} \\
\mathrm{E}[R+B] &= \mathrm{E}[R] + \mathrm{E}[B] = 7 && \text{[linearity]} \\
\mathrm{E}[R+\mathcal{U}] &= 7 && \text{[trivial distribution]} \\
\mathrm{E}[R+B+\mathcal{U}] &= \mathrm{E}[R] + \mathrm{E}[B] + \mathrm{E}[\mathcal{U}] = \frac{21}{2} && \text{[linearity]} \\
\mathrm{E}[R \cdot B] &= \mathrm{E}[R] \cdot \mathrm{E}[B] = \frac{49}{4} && \text{[independence]} \\
\mathrm{E}[R \cdot \mathcal{U}] &= \frac{1\cdot 6 + 2\cdot 5 + 3\cdot 4}{3} = \frac{28}{3} \\
\mathrm{E}[R^2] &= \frac{1^2+2^2+3^2+4^2+5^2+6^2}{6} = \frac{91}{6} \\
\mathrm{E}[(R+B)^2] &= \mathrm{E}[R^2] + 2\,\mathrm{E}[RB] + \mathrm{E}[B^2] = \frac{329}{6} && \text{[linearity]} \\
\mathrm{E}[R+B \mid R=6] &= \mathrm{E}[R \mid R=6] + \mathrm{E}[B \mid R=6] && \text{[linearity]} \\
&= 6 + \mathrm{E}[B] = 19/2 && \text{[independence]}
\end{aligned}$$

$$\begin{aligned}
\mathrm{E}[R \mid R+B=6] &= \frac{1+2+3+4+5}{5} = 3 \\
\mathrm{E}[R^2 \mid R+B=6] &= \frac{1^2+2^2+3^2+4^2+5^2}{5} = 11 \\
\mathrm{E}[R+B \mid R\cdot B=6] &= \frac{(1+6)+(2+3)}{2} = 6 \\
\mathrm{E}[R\cdot B \mid R+B=6] &= \frac{(1\cdot 5)+(2\cdot 4)+(3\cdot 3)+(4\cdot 2)+(5\cdot 1)}{5} = 7
\end{aligned}$$

1.3 Common Probability Distributions

A ***probability distribution*** assigns a probability to each possible value of a random variable. More formally, $X : \Omega \to V$ is a random variable over some probability space $(\Omega, \Pr)$, the probability distribution of X is the function $P : V \to [0,1]$ such that

$$P(x) = \Pr[X = x] = \sum \{\Pr(\omega) \mid X(\omega) = x\}.$$

The ***support*** of a probability distribution is the set of values with non-zero probability; this is a subset of the value set V. The following table summarizes several of the most useful discrete probability distributions.

name	intuition	parameters	support	$\Pr[X = x]$	$\mathrm{E}[X]$
trivial	Good ol' Rock, nothing beats that!	—	singleton set $\{a\}$	1	a
uniform	fair die roll	—	finite set $S \neq \varnothing$	$\frac{1}{\lvert S\rvert}$	$\frac{\sum S}{\lvert S\rvert}$
Bernoulli	biased coin flip	$0 \le p \le 1$	$\{0,1\}$	$\begin{cases} p & \text{if } x = 1 \\ 1-p & \text{if } x = 0 \end{cases}$	p
binomial	n biased coin flips	$0 \le p \le 1$ $n \ge 0$	$[0\,..\,n]$	$\binom{n}{x} p^x (1-p)^{n-x}$	np
geometric	#tails before first head	$0 < p \le 1$	$\mathbb{N}$	$(1-p)^x p$	$\frac{1-p}{p}$
negative binomial	#tails before nth head	$0 < p \le 1$ $n \ge 0$	$\mathbb{N}$	$\binom{n+x-1}{x}(1-p)^x p^n$	$\frac{n(1-p)}{p}$

Figure 1.1. Common discrete probability distributions.

- The ***trivial*** distribution describes the outcome of a "random" experiment that *always* has the same result. A trivially distributed random variable takes some fixed value with probability 1. Yes, this is still randomness.

- The ***uniform*** distribution assigns the same probability to every element of some finite non-empty set S. For example, if the random variable X is uniformly distributed over the integer range $[1\,..\,n]$, then $\Pr[X = x] = 1/n$ for each integer x between 1 and n, and $\mathrm{E}[X] = (n+1)/2$. This distribution models (idealized) fair coin flips, die rolls, and lotteries; consequently, this is what many people *incorrectly* think of as the definition of "random".

- The ***Bernoulli*** distribution models a random experiment (called a ***Bernoulli trial***) with two possible outcomes: ***success*** and ***failure***. The probability of success, usually denoted p, is a parameter of the distribution; the failure probability is often denoted $q = 1 - p$. Success and failure are usually represented by the *values* 1 and 0. Thus, every indicator random variable has a Bernoulli distribution, and its expected value is equal to its success probability:

$$\mathrm{E}[X] = \sum_x x \cdot \Pr[X = x] = 0 \cdot \Pr[X = 0] + 1 \cdot \Pr[X = 1] = \Pr[X = 1] = p.$$

  The special case $p = 1/2$ is a uniform distribution with two values: a fair coin flip. The special cases $p = 0$ and $p = 1$ are trivial distributions.

- The ***geometric*** distribution describes the number of independent Bernoulli trials (all with the same success probability p) before the first success. If X is a geometrically distributed random variable, then $X = x$ if and only if the first x Bernoulli trials fail and the $(x+1)$th trial succeeds:

$$\Pr[X = x] = \left(\prod_{i=1}^{x} \Pr[i\text{th trial fails}]\right) \cdot \Pr[(x+1)\text{th trial succeeds}] = (1-p)^x p.$$

- The ***binomial*** distribution is the sum of n independent Bernoulli distributions, all with the same probability p of success. If X is a binomially distributed random variable, then $X = x$ if and only if x of the n trials succeed and $n - x$ fail:

$$\Pr[X = x] = \binom{n}{x} p^x (1-p)^{n-x}.$$

  If $n = 1$, this is just the Bernoulli distribution.

- The ***negative binomial*** distribution describes the number of independent Bernoulli trials (all with the same success probability p) that fail before the nth successful trial. If X is a negative-binomially distributed random variable, then $X = x$ if and only if exactly x of the first $n + x - 1$ trials are failures, and the $(n+x)$th trial is a success:

$$\Pr[X = x] = \binom{n+x-1}{x} (1-p)^x p^n$$

  If $n = 1$, this is just the geometric distribution.

1.4 Coin Flips

Suppose you are given a coin and you are asked generate a uniform random bit. We distinguish between two types of coins: A coin is *fair* if the probability of heads (1) and the probability of tails (0) are both exactly $1/2$. A coin where one side is more likely than the other is said to be *biased*. Actual physical coins are reasonable approximations of abstract fair coins for most purposes, at least if they're flipped high into the air and allowed to bounce.[3] Physical coins can be biased by bending them.

1.4.1 Removing Unknown Bias

In 1951, John von Neumann discovered the following simple technique to simulate fair coin flips using an arbitrarily biased coin, *even without knowing the bias*. Flip the biased coin twice. If the two flips yield different results, return the first result; otherwise, repeat the experiment from scratch.

```
VonNeumannCoin():
  x ← BiasedCoin()
  y ← BiasedCoin()
  if x ≠ y
      return x
  else
      return VonNeumannCoin()
```

[3] Persi Diaconis, Susan Holmes, and Richard Montgomery published a thorough analysis of physical coin-flipping in 2007, which concluded (among other things) that coins that are flipped vigorously and then caught come up in the same state they started about 51% of the time. The small amount of bias arises because flipping coins tend to precess as they rotate. Letting the coin bounce instead of catching them appears to remove the bias from precession.

This is weird sort of algorithm, isn't it? There is *no* upper bound on the worst-case running time; in principle, the algorithm could run *forever*, because the biased coin always just happens to flip heads. Nevertheless, I claim that this is a useful algorithm for generating fair random bits. We need two technical assumptions:

(1) The biased coin always flips heads with the same fixed (but unknown!) probability p. To simplify notation, we let $q = 1 - p$ denote the probability of flipping tails. For example, a fair coin would have $p = q = 1/2$.

(2) All flips of the biased coin are mutually independent.

First, I claim that *if* the algorithm halts, then it returns a uniformly distributed random bit. Because the two biased coin flips are independent, we have

$$\Pr[x = 0 \wedge y = 1] = \Pr[x = 1 \wedge y = 0] = pq,$$

and therefore (assuming $pq > 0$)

$$\Pr[x = 0 \wedge y = 1 \mid x \neq y] = \Pr[x = 1 \wedge y = 0 \mid x \neq y] = \frac{pq}{2pq} = \frac{1}{2}.$$

Because the biased coin flips are mutually independent, the same analysis applies without modification to every recursive call to VonNeumannCoin. Thus, if *any* recursive call returns a bit, that bit is uniformly distributed.

Now let T denote the actual running time of this algorithm; T is a random variable that depends on the biased coin flips.[4] We can compute the expected running time $\mathrm{E}[T]$ by considering the conditional expectations in two cases: The first two flips are either different or equal.

$$\mathrm{E}[T] \;=\; \mathrm{E}[T \mid x \neq y] \cdot \Pr[x \neq y] \;+\; \mathrm{E}[T \mid x = y] \cdot \Pr[x = y]$$

Because the two biased coin flips are independent, we have

$$\Pr[x \neq y] = \Pr[x = 0 \wedge y = 1] + \Pr[x = 1 \wedge y = 0] = 2pq$$

and therefore $\Pr[x = y] = 1 - 2pq$. If the first two coin flips are different, the algorithm ends after two flips; thus, $\mathrm{E}[T \mid x \neq y] = 2$. Finally, if the first two coin flips are the same, the experiment starts over from scratch after the first two flips, so $\mathrm{E}[T \mid x = y] = 2 + \mathrm{E}[T]$. Putting all the pieces together, we have

$$\mathrm{E}[T] = 2 \cdot 2pq + (2 + \mathrm{E}[T]) \cdot (1 - 2pq).$$

Solving this equation for $\mathrm{E}[T]$ yields the solution $\mathbf{E[T] = 1/pq}$. For example, if $p = 1/3$, the expected number of coin flips is $9/2$.

Alternatively, we can think of VonNeumannCoin as performing an experiment with a different biased coin, which returns a result ("heads") with probability $2pq$. Thus, the expected number of *unsuccessful* iterations ("tails" before the first "head") is a geometric random variable with expectation $1/2pq - 1$, and thus the expected number of iterations is $1/2pq$. Because each iteration flips two coins, the expected number of coin flips is $1/pq$.

[4] Normally we use $T(n)$ to denote the worst-case running time of an algorithm as a function of some input parameter n, but this algorithm has no input parameters!

1.4.2 Removing Known Bias

But what if we *know* that $p = 1/3$? In that case, the following algorithm simulates a fair coin with fewer biased flips (on average):

```
FAIRCOIN():
  x ← BIASEDCOIN(1/3)
  y ← BIASEDCOIN(1/3)
  if x ≠ y                ⟨⟨probability 4/9⟩⟩
      return 0
  else if x = y = 1       ⟨⟨probability 4/9⟩⟩
      return 1
  else                    ⟨⟨probability 1/9⟩⟩
      return FAIRCOIN()
```

The algorithm returns a fair coin because

$$\Pr[x \neq y] = \frac{4}{9} = \Pr[x = y = 0].$$

The expected number of flips satisfies the equation

$$\mathrm{E}[T] = 2 + \frac{1}{9}\mathrm{E}[T],$$

which implies that $\mathrm{E}[T] = 9/4$, a factor of 2 better than von Neumann's algorithm.

1.5 Pokémon Collecting

A distressingly large fraction of my daughters' friends are obsessed with Pokémon—not the cartoon or the mobile game, but the collectible card game. The Pokémon Company sells small packets, each containing half a dozen cards, each describing a different Pokémon character. The cards can be used to play a complex turn-based combat game; the more cards a player owns, the more likely they are to win. So players are strongly motivated to collect as many cards, and in particular, as many *different* cards, as possible. Pokémon reinforces this message with their oh-so-subtle theme song "Gotta Catch 'Em All!" Unfortunately, the packets are opaque; the only way to find out which cards are in a pack is to buy the pack and tear it open.[5]

Let's consider the following oversimplified model of the Pokémon-collection process. In each trial, we purchase one Pokémon card, chosen independently and uniformly at random from the set of n possible card types. We repeat these trials until we have purchased at least one of each type of card. This problem was first considered by the French mathematician Abraham de Moivre in his seminal 1712 treatise *De Necessitate ut Caperent Omnium Eorum*.[6]

1.5.1 After n Trials

How many different types of Pokémon do we actually own after we buy n cards? Obviously in the worst case, we might just have n copies of one Pokémon,[7] but that's not particularly likely. To

[5] See also: cigarette cards, Dixie cups, baseball cards, Pez dispensers, Beanie Babies, Iwako puzzle erasers, Shopkins, and Guys Under Your Supervision.

[6] The actual title was *De Mensura Sortis seu; de Probabilitate Eventuum in Ludis a Casu Fortuito Pendentibus*, which means "On the measurement of chance, or on the probability of events in games depending on fortuitous chance".

[7] "Dave Guy!"

analyze the *expected* number of types that we own, we introduce an incredibly useful technique that lets us exploit linearity of expectation: decomposing more complex random variables into ***sums of indicator variables***.

For each index i, define an indicator variable $X_i = [\text{we own Pokémon } i]$ so that $X = \sum_i X_i$ is the number of cards we own. Linearity of expectation implies

$$\mathrm{E}[X] = \sum_{i=1}^{n} \mathrm{E}[X_i] = \sum_{i=1}^{n} \Pr[X_i = 1].$$

The probability that we *don't* own card i is $(1-1/n)^n \approx 1/e$, so

$$\mathrm{E}[X] = \sum_{i=1}^{n} \mathrm{E}[X_i] \approx \sum_{i=1}^{n} (1-1/e) = (1-1/e)n \approx 0.63212n.$$

In other words, after buying n cards, we expect to own a bit less than $2/3$ of the Pokémon.

Similar calculations implies that we expect to own about 86% of the Pokémon after buying $2n$ cards, about 95% after buying $3n$ cards, about 98% after buying $4n$ cards, and so on.

1.5.2 Gotta Catch 'em *All*

So how many Pokémon packs do we need to buy to catch 'em *all*? Obviously in the worst case, we might *never* have a complete collection[8], but assuming each type of card has some non-zero probability of being in each pack, the *expected* number of packs we need to buy is finite. Let $T(n)$ denote the number of packs (or "time") required to collect all n Pokémon. For purposes of analysis, we partition the random purchasing algorithm into n *phases*, where the ith phase ends just after we see the ith distinct card type. Let us write

$$T(n) = \sum_i T_i(n)$$

where $T_i(n)$ is the number of cards bought during the ith phase. Linearity of expectation implies

$$\mathrm{E}[T(n)] = \sum_i \mathrm{E}[T_i(n)].$$

We can think of each card purchase as a biased coin flip, where "heads" means "got a new Pokémon" and "tails" means "got a Pokémon we already own". For each index i, the probability of heads (that is, the probability of a single purchase being a new Pokémon) is exactly $p = (n-i+1)/n$: each of the n Pokémon is equally likely, and there are $n-i+1$ Pokémon that we don't already own. By our earlier analysis, the expected number of flips until the first head is $\mathrm{E}[T_i] = 1/p = n/(n-i+1)$. We conclude that

$$\mathrm{E}[T(n)] \;=\; \sum_i \mathrm{E}[T_i(n)] \;=\; \sum_{i=1}^{n} \frac{n}{n-i+1} \;=\; \sum_{j=1}^{n} \frac{n}{j} \;=\; nH_n.$$

Here $\boldsymbol{H_n}$ denotes the nth ***harmonic number***, defined recursively as

$$H_n = \begin{cases} 0 & \text{if } n = 0 \\ H_{n-1} + \dfrac{1}{n} & \text{otherwise} \end{cases}$$

[8] "Dave Guy!"

Approximating the summation $H_n = \sum_{i=1}^{n} \frac{1}{i}$ above and below by integrals implies the bounds

$$\ln(n+1) \le H_n \le (\ln n) + 1.$$

Thus, the expected number of cards we need to buy to get all n Pokémon is $\Theta(n \log n)$.

In particular, to catch all 150 of the original Pokémon, we should expect to buy $150 \cdot H_{150} \approx 838.67709$ cards, and to own at least one copy of each of the 9184 Pokémon card types available in 2013, we should expect to buy $9184 \cdot H_{9184} \approx 89107.65186$ cards. (In practice, of course, this estimate is far too low, because some cards are considerably more common than others.)

1.6 Random Permutations

Now suppose we are given a deck of n (Pokémon?) cards and are asked to shuffle them. Ideally, we would like an algorithm that produces each of the $n!$ possible permutations of the deck with the same probability $1/n!$.

There are many such algorithms, but the gold standard is ultimately based on the millennia-old tradition of drawing or casting lots. "Lots" are traditionally small pieces of wood, stone, or paper, which were blindly drawn from an opaque container. The following algorithm takes a set L of n distinct *L*ots (arbitrary objects) as input and returns an array $R[1..n]$ containing a *R*andom permutation of those n lots.

```
DRAWLOTS(L):
  n ← |L|
  for i ← 1 to n
      remove a random lot x from L
      R[i] ← x
  return R[1..n]
```

There are exactly $n!$ possible outcomes for this algorithm—exactly n choices for the first lot, then exactly $n-1$ choices for the second lot, and so on—each with exactly the same probability and each leading to a different permutation. Thus, every permutation of lots is equally likely to be output.

A modern formulation of lot-casting was described by (and is frequently misattributed to) statisticians Ronald Fisher and Frank Yates in 1938. Fisher and Yates formulated their algorithm as a method for randomly reordering a *L*ist of numbers. In their original formulation, the algorithm repeatedly chooses at random a number from the input list that has not been previously chosen, adds the chosen number to the output list, and then strikes the chosen number from the input list.

We can implement this formulation of the algorithm using a secondary boolean array $Chosen[1..n]$ indicating which items have already been chosen. The randomness is provided by a subroutine ***RANDOM(n)***, which returns an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, n\}$ in $O(1)$ time; in other words, RANDOM(n) simulates a fair n-sided die.

```
FisherYates(L[1..n]):
  for i ← 1 down to n
      Chosen[i] ← False
  for i ← n down to 1
      repeat
          r ← Random(n)
      until ¬Chosen[r]
      R[i] ← L[r]
      Chosen[r] ← True
  return R[1..n]
```

The repeat-until loop chooses an index r uniformly at random from the set of previously unchosen indices. Thus, this algorithm really is an implementation of DrawLots. But now choosing the next random ~~lot~~ element may require several iterations of the repeat-until loop. How slow is this algorithm?

In fact, FisherYates is equivalent to our earlier Pokémon-collecting algorithm! Each call to Random is a purchase, and $Chosen[i]$ indicates whether we've already purchased the ith Pokémon. By our earlier analysis, the expected number of calls to Random before the algorithm halts is *exactly* nH_n. We conclude that this algorithm runs in $\Theta(n \log n)$ ***expected time***.[9]

A most efficient implementation of lot-casting, which permutes the input array in place, was described by Richard Durstenfeld in 1961. In full accordance with Stigler's Law, this algorithm is almost universally called "the Fisher-Yates shuffle".[10]

```
SelectionShuffle(A[1..n]):
  for i ← n down to 1
      swap A[i] ↔ A[Random(i)]
```

The algorithm clearly runs in $O(n)$ time. Correctness follows from exactly the same argument as DrawLots: There are $n!$ equally likely possibilities from the n calls to Random—n for the first call, $n-1$ for the second, and so on—each leading to a different output permutation.

Although it may not appear so at first glance, SelectionShuffle is an implementation of lot-casting. After each iteration of the main loop, the suffix $A[i..n]$ (on the *R*ight) plays the role of R, storing the previously chosen input elements, and the prefix $A[1..i-1]$ (on the *L*eft) plays the role of L, containing all the unchosen input elements. Unlike the original FisherYates algorithm, SelectionShuffle changes the *order* of the unchosen elements as it runs. Fortunately, and obviously, that order is utterly irrelevant; only the *set* of unchosen elements matters.

We can also uniformly shuffle by reversing the order of the loop in the previous algorithm. Again, this algorithm is usually misattributed to Fisher and Yates.

```
InsertionShuffle(A[1..n]):
  for i ← 1 to n
      swap A[i] ↔ A[Random(i)]
```

Again, correctness follows from the observation that there are $n!$ equally likely output permutations. Alternatively, we can argue inductively that for every index i, the prefix $A[1..i]$ is

[9] In later editions of Fisher and Yates' monograph, they instead described a different algorithm due to C. Radhakrishna Rao, dismissing their earlier method as "tiresome, since each [item] must be deleted from a list as it is selected and a fresh count made for each further selection."

[10] However, some authors call this algorithm the "Knuth shuffle", because Donald Knuth described it in his landmark *Art of Computer Programming*, even though he attributed the algorithm to Durstenfeld in the first edition, and to Fisher and Yates in the second. It's actually rather shocking that Knuth did not attribute the algorithm to Aaron, citing the First Chronicles.

uniformly shuffled after the ith iteration of the loop. Alternatively, we can observe that running InsertionShuffle is the same as running SelectionShuffle *backward in time*—essentially putting the lots back into the bag—and that the inverse of a uniformly-distributed permutation is also uniformly distributed.

(The names SelectionShuffle and InsertionShuffle for these two variants are non-standard. SelectionShuffle randomly *selects* the next card and then adds to one end of the random permutation, just as selection sort repeatedly selects the largest element of the unsorted portion of the array. InsertionShuffle randomly *inserts* the first card in the untouched portion of the array into the random permutation, just as insertion sort repeatedly inserts the next item in the unsorted portion of the input into the sorted portion.)

1.7 Properties of Random Permutations

- For any subsequence of indices: the set of values and the permutation of those values are uniformly and independently distributed.
- For any subsequence of values: the set of indices and the permutation of those indices are uniformly and independently distributed.
- For example: In a randomly shuffled deck, the expected number of hearts among the first 5 face cards is 5/4.

Exercises

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). The 13 cards in each suit have distinct *ranks*: A (ace), 2 (deuce), 3 (trey), 4, 5, 6, 7, 8, 9, 10, J (jack), Q (queen), and K (king). Cards are normally named by writing their rank followed by their suit; for example, J♠ is the jack of spades, and 10♥ is the ten of hearts. For purposes of comparing ranks in the problems below, aces have rank 1, jacks have rank 11, queens have rank 12, and kings have rank 13; for example, J♠ has higher rank than 8♦, but lower rank than Q♣.

---

1. 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 T **H H H**

   (a) What is the expected number of flips in one of Rosencrantz's turns?

   (b) Suppose Rosencrantz happens to flip 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! Full credit requires *exact* answers, but a correct asymptotic bound (as a function of k) in part (b) is worth significant partial credit.

2. After sending his loyal friends Rosencrantz and Guildenstern off to Norway, Hamlet decides to amuse himself by repeatedly flipping a fair coin until the sequence of flips satisfies some condition. For each of the following conditions, compute the *exact* expected number of flips until that condition is met. Some conditions depend on a positive integer parameter n.

   (a) Hamlet flips n times.

   (b) Hamlet flips heads.

   (c) Hamlet flips both heads and tails (in different flips, of course).

   (d) Hamlet flips heads twice.

   (e) Hamlet flips heads twice in a row.

   (f) Hamlet flips heads followed immediately by tails.

   (g) Hamlet flips the sequence heads, tails, heads, tails.

   (h) Hamlet flips heads n times.

   (i) Hamlet flips heads n times in a row.

   (j) Hamlet flips more heads than tails.

   (k) Hamlet flips the same positive number of heads and tails.

   (l) Either Hamlet flips more heads than tails or he flips n times, whichever comes first.

   For each condition, prove that your answer is correct. If you have to appeal to "intuition" or "common sense", your answer is almost certainly wrong! Correct asymptotic bounds for the conditions that depend on n are worth significant partial credit.

3. 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. Consider the following randomized algorithm for generating biased random bits.

```
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 instead of FAIRCOIN you are given a subroutine BIASEDCOIN that returns an independent random bit equal to 1 with some fixed *but unknown* probability p, in $O(1)$ time. Describe an algorithm ONEINTHREE that returns either 0 or 1 with equal probability, using BIASEDCOIN as its only source of randomness.

(d) What is the *exact* expected number of times that your OneInThree algorithm calls BiasedCoin?

4. Describe an algorithm that takes a real number $0 \le p \le 1$ as input, and returns an independent random bit that is equal to 1 with the given probability p, using independent fair coin flips as the only source of randomness. What is the expected running time of your algorithm (as a function of p)?

5. (a) Describe an algorithm that simulates a fair three-sided die, using independent fair coin flips as the only source of randomness. Your algorithm should return 1, 2, or 3, each with probability 1/3. What is the expected number of coin flips used by your algorithm?

   (b) Describe an algorithm that simulates a fair coin flip, using independent rolls of a fair three-sided die as the only source of randomness. What is the expected number of die rolls used by your algorithm?

   *(c) Describe an algorithm that simulates a fair n-sided die, using independent rolls of a fair k-sided die as the only source of randomness. What is the expected number of die rolls used by your algorithm (as a function of n and k)? You may assume n and k are relatively prime.

6. Describe an algorithm FairDie that returns an integer chosen uniformly at random from the set $\{1, 2, 3, 4, 5, 6\}$, using an algorithm LoadedDie that returns an algorithm from the same set with some fixed *but unknown* non-trivial probability distribution. What is the expected number of times your FairDie algorithm calls LoadedDie? *[Hint:* $3! = 6$.*]*

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) that returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$, given a non-negative integer n as input, using FairCoin as its only source of randomness.

   (b) Suppose you have access to a function FairCoins(k) that returns an integer chosen uniformly and independently at random 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) that returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$, given *any* non-negative integer n as input, using FairCoins as its only source of randomness.

8. You are applying to participate in this year's Trial of the Pyx, the annual ceremony where samples of all British coinage are tested, to ensure that they conform as strictly as possible to legal standards. As a test of your qualifications, your interviewer at the Worshipful Company of Goldsmiths has given you a bag of n commemorative Alan Turing half-guinea coins, exactly two of which are counterfeit. One counterfeit coin is very slightly lighter than a genuine Turing; the other is very slightly heavier. Together, the two counterfeit coins have *exactly* the same weight as two genuine coins. Your task is to identify the two counterfeit coins.

The weight difference between the real and fake coins is too small to be detected by anything other than the Royal Pyx Coin Balance. You can place any two disjoint sets of coins in each of the Balance's two pans; the Balance will then indicate which of the two subsets has larger total weight, or that the two subsets have the same total weight. Unfortunately, each use of the Balance requires completing a complicated authorization form (in triplicate), submitting a blood sample, and scheduling the Royal Bugle Corps, so you *really* want to use the Balance as few times as possible.

(a) Suppose you *randomly* choose $n/2$ of your n coins to put on one pan of the Balance, and put the remaining $n/2$ coins on the other pan. What is the probability that the two subsets have equal weight?

(b) Describe and analyze a randomized algorithm to identify the two fake coins. What is the expected number of times your algorithm uses the Balance?

9. Consider the following algorithm for shuffling a deck of n cards, initially numbered in order from 1 on the top to n on the bottom. At each step, we remove the top card from the deck an insert it randomly back into in the deck, choosing one of the n possible positions uniformly at random. The algorithm ends immediately after we pick up card $n-1$ and insert it randomly into the deck.

(a) Prove that this algorithm uniformly shuffles the deck, so that each permutation of the deck has equal probability. *[Hint: Prove that at all times, the cards below card $n-1$ are uniformly shuffled.]*

(b) Prove that before the algorithm ends, the deck is *not* uniformly shuffled.

(c) What is the expected number of steps before this algorithm ends?

10. Suppose we want to write an efficient algorithm SHUFFLE($A[1..n]$) that randomly permutes the input array, so that each of the $n!$ permutations is equally likely.

(a) Prove that the following algorithm is ***not*** correct. *[Hint: Consider the case $n = 3$.]*

```
NAIVESHUFFLE(A[1..n]):
  for i ← 1 to n
      swap A[i] ↔ A[RANDOM(n)]
```

(The only difference from INSERTIONSHUFFLE is that the argument to RANDOM is n instead of i.)

(b) The algorithm BUFFERSHUFFLE, shown in Figure 1.2, takes an extra parameter b describing the size of the internal buffer array $B[1..b]$.

i. Prove that BUFFERSHUFFLE is correct for all $b \geq n$.

ii. What is the expected running time of BUFFERSHUFFLE when $b = n$?

iii. What is the expected running time of BUFFERSHUFFLE when $b = 2n$?

*(c) Prove that the algorithm RAOSHUFFLE, shown in Figure 1.2, is correct and analyze its expected running time. This is the algorithm of C. Radhakrishna Rao that Fisher and Yates found less "tiresome" than their own.

```
BUFFERSHUFFLE(n, b):
  ⟨⟨Initialize buffer⟩⟩
  for j ← 1 to 2n
      B[j] ← NULL
  ⟨⟨Copy items randomly into buffer⟩⟩
  for i ← 1 to n
      j ← RANDOM(b)
      while (B[j] == NULL)
          j ← RANDOM(n)
      B[j] ← A[i]
  ⟨⟨Compress buffer into input array⟩⟩
  i ← 1
  for j ← 1 to b
      if B[j] ≠ NULL
          A[i] ← B[j]
          i ← i + 1
```

```
RAOSHUFFLE(A[1..n]):
  ⟨⟨Initialize n lists⟩⟩
  for j ← 1 to n
      ℓ[j] ← 0
  ⟨⟨Add each A[i] to a random list⟩⟩
  for i ← 1 to n
      ℓ[j] ← ℓ[j] + 1
      L[j][ℓ[j]] ← A[i]
  ⟨⟨Recursively shuffle the lists⟩⟩
  for j ← 1 to n
      if ℓ[j] > 1
          SHUFFLE(L[j][1..ℓ[j]])
  ⟨⟨Concatenate the lists⟩⟩
  i ← 0
  for j ← 1 to n
      for k ← 1 to ℓ[j]
          A[i] ← L[j][k]
          i ← i + 1
```

Figure 1.2. Left: Shuffling using a buffer array. Right: Rao's less "tiresome" shuffling algorithm.

11. (a) Prove that the following algorithm, modeled after quicksort, uniformly permutes its input array, meaning each of the $n!$ possible output permutations is equally likely. *[Hint:* $\binom{n}{k} = \frac{n!}{k!(n-k)!}$*]*

```
QUICKSHUFFLE(A[1..n]):
  if n ≤ 1
      return
  j ← 1;  k ← n
  while j ≤ k
      with probability 1/2
          swap A[j] ↔ A[k]
          k ← k − 1
      else
          j ← j + 1
  QUICKSHUFFLE(A[1..k])
  QUICKSHUFFLE(A[j..n])
```

(b) Prove that QUICKSHUFFLE runs in $O(n \log n)$ expected time. *[Hint: This will be much easier after reading the next chapter.]*

(c) Prove that the algorithm MERGESHUFFLE shown on the next page, which is modeled after mergesort, does ***not*** uniformly permute its input array.

(d) Describe how to modify RANDOMMERGE so that MERGESHUFFLE ***does*** uniformly permute its input array, and prove that your modification is correct. *[Hint: Modify the line in red.]*

```
MERGESHUFFLE(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESHUFFLE(A[1..m])
      MERGESHUFFLE(A[m+1..n])
      RANDOMMERGE(A[1..n], m)
```

```
RANDOMMERGE(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 with probability 1/2
          B[k] ← A[i]; i ← i+1
      else
          B[k] ← A[j]; j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

12. 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?

13. 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 the good Professor, 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.

14. 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.[11] 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* $13 = n$*.]*

15. 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 $k/(n+1)$.

    (c) Prove that if we start at vertex 1, the expected number of steps before the random walk ends is exactly n.

    (d) What is the *exact* expected length of the random walk if we start at vertex k, as a function of n and k? Prove your result is correct. (For partial credit, give a tight Θ-bound for the case $k = (n+1)/2$, assuming n is odd.)

    [Hint: Trust the recursion fairy.]

16. Suppose n lights labeled $0, \ldots, n-1$ are placed clockwise around a circle. Initially, every light is off. Consider the following random process.

[11] Specifically, he hurls it directly into the back of Penn's right hand.

```
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 $p(i, n)$ denote the probability that the last light turned on by LightTheCircle$(n, 0)$ is light i. 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. *[Hint: You may find the previous exercise helpful.]*

17. Let T be an arbitrary, ***not*** necessarily balanced, binary tree T with n nodes.

(a) Consider a random walk downward from the root of T, as described by the following algorithm.

```
RandomTreeDescent(T):
  v ← T.root
  while v ≠ Null
      with probability 1/2
          v ← v.left
      else
          v ← v.right
```

Find a tight bound on the worst-case expected running time of this algorithm, as a function of n, and prove your answer is correct. What is the worst-case tree?

(b) Now consider a different random walk starting at the root of T, as described by the following algorithm.

```
RandomTreeWalk(T):
  v ← T.root
  while v ≠ Null
      with probability 1/3
          v ← v.left
      else with probability 1/3
          v ← v.right
      else
          v ← v.parent
```

Find a tight bound on the worst-case expected running time of this algorithm, as a function of n, and prove your answer is correct. What is the worst-case tree?

2 Randomized Algorithms

2.1 Nuts and Bolts

The following interesting problem was first proposed by Gregory Rawlins around 1992. 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, and we're playing with them in the dark, 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. How quickly can we match each but to the corresponding bolt?

Before we attack this problem, let's think about something simpler. 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 but except one; if we get to the last nut without finding a match, we know that the last nut is the one we're looking for.

Intuitively, in the "average" case, this algorithm will look at approximately $n/2$ nuts. But what exactly does "average case" mean?

2.2 Deterministic vs. Randomized Algorithms

Normally, when we talk about the running time of an algorithm, we mean its *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:

$$T_{\text{worst-case}}(n) = \max_{|X|=n} T(X).$$

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 input X:

$$T_{\text{average-case}}(n) = \mathop{\mathrm{E}}_{|X|=n}[T(X)] = \sum_{|X|=n} T(X) \cdot \Pr[X].$$

The problem with this definition is that we rarely, if ever, know the probability of getting any particular input X. 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, if it can be considered random at all, is random in some interesting way that can't be predicted in advance.

Instead of considering the 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 definition implies 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 consider the expected 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!

2.3 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. 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.[1] 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:

$$\mathrm{E}[T(n)] = \sum_{k=1}^{n-1} k \cdot \Pr[T(n) = k]$$

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:

$$\Pr[T(n) = k] = \begin{cases} 1/n & \text{if } k < n-1, \\ 2/n & \text{if } k = n-1. \end{cases}$$

[1]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 $T(X)$.

Plugging this into the definition of expectation gives us our answer.

$$\begin{aligned}\mathrm{E}[T(n)] &= \sum_{k=1}^{n-2} \frac{k}{n} + \frac{2(n-1)}{n} \\ &= \sum_{k=1}^{n} \frac{k}{n} - \frac{1}{n} \\ &= \frac{n+1}{2} - \frac{1}{n}\end{aligned}$$

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.

$$\begin{aligned} t(n) &= \sum_{k=2}^{n} k = \frac{n(n+1)}{2} - 1 \\ \Longrightarrow \mathrm{E}[T(n)] &= \frac{t(n)}{n} = \frac{n+1}{2} - \frac{1}{n}\end{aligned}$$

2.4 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 a more interesting strategy, which Rawlins proposed when he introduced the problem. 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

$$\begin{aligned} T(n) &= 2n - 1 + \max_{1 \le k \le n} \big\{ T(k-1) + T(n-k) \big\} \\ &= 2n - 1 + T(n-1)\end{aligned}$$

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.

2.5 Reductions to and from Sorting

Rawlins' algorithm for matching the nuts and bolts strongly resembles 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[2], but both the algorithms and their analysis are incredibly technical and the constant hidden in the $O(\cdot)$ notation is quite large.

2.6 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

[2] 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.

given by the following recurrence:

$$\begin{aligned}\overline{T}(n) &= 2n-1+\mathrm{E}_k\big[\overline{T}(k-1)+\overline{T}(n-k)\big] \\ &= 2n-1+\frac{1}{n}\sum_{k=1}^{n}\big(\overline{T}(k-1)+\overline{T}(n-k)\big)\end{aligned}$$

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 2.8 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 may be 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.

2.7 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.

Figure

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*. It follows (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; linearity of expectation allows us to write this expected value as follows:

$$\mathrm{E}[T(n)] = \mathrm{E}\left[\sum_{i=1}^{n}\sum_{j=1}^{n} X_{ij}\right] = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathrm{E}[X_{ij}].$$

By definition of expectation,

$$\mathrm{E}[X_{ij}] = 0 \cdot \Pr[X_{ij} = 0] + 1 \cdot \Pr[X_{ij} = 1] = \Pr[X_{ij} = 1].$$

Thus, to analyze our algorithm, we only 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.

$$\begin{aligned}
\mathrm{E}[T(n)] &= \sum_{i=1}^{n}\sum_{j=1}^{n}\mathrm{E}[X_{ij}] \\
&= \sum_{i=1}^{n}\mathrm{E}[X_{ii}] + 2\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}] \\
&= n + 4\sum_{i=1}^{n}\sum_{j=i+1}^{n}\frac{1}{j-i+1}
\end{aligned}$$

This is quite a bit simpler than our earlier recurrence. Just a few more lines of algebra gives us an exact, closed-form expression for the expected number of comparisons.

$$\begin{aligned}
\mathrm{E}[T(n)] &= n + 4\sum_{i=1}^{n}\sum_{k=2}^{n-i+1}\frac{1}{k} && \text{[substitute } k = j-i+1\text{]} \\
&= n + 4\sum_{k=2}^{n}\sum_{i=1}^{n-k+1}\frac{1}{k} && \text{[reorder summations]} \\
&= n + 4\sum_{k=2}^{n}\frac{n-k+1}{k} \\
&= n + 4\left((n+1)\sum_{k=2}^{n}\frac{1}{k} - \sum_{k=2}^{n}1\right) \\
&= n + 4((n+1)(H_n - 1) - (n-1)) \\
&= 4nH_n - 7n + 4H_n
\end{aligned}$$

Sure enough, it's $\Theta(n \log n)$.

*2.8 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.

$$\overline{T}(n) \;=\; \frac{1}{n}\sum_{k=1}^{n}\big(\overline{T}(k-1) + \overline{T}(n-k)\big) \;+\; 2n-1 \;=\; \frac{2}{n}\sum_{k=0}^{n-1}\overline{T}(k) \;+\; 2n-1$$

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.

$$n\overline{T}(n) = 2\sum_{k=0}^{n-1}\overline{T}(k) \;+\; 2n^2 - n$$

$$\begin{aligned}
(n-1)\overline{T}(n-1) &= 2\sum_{k=0}^{n-2}\overline{T}(k) \;+\; 2(n-1)^2-(n-1) \\
&= 2\sum_{k=0}^{n-2}\overline{T}(k) \;+\; 2n^2-5n+3 \\
n\overline{T}(n)-(n-1)\overline{T}(n-1) &= 2T(n-1)+4n-3 \\
\overline{T}(n) &= \frac{n+1}{n}\overline{T}(n-1)+4-\frac{3}{n}
\end{aligned}$$

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: $H_{n+1} = H_n + \frac{1}{n+1}$. Finally, substituting $\overline{T}(n) = (n+1)t(n)$ and simplifying gives us the exact solution to the original recurrence.

$$\overline{T}(n) \;=\; 4(n+1)H_n-7(n+1)+7 \;=\; 4nH_n-7n+4H_n$$

Surprise, surprise, we get exactly the same solution!

Exercises

Unless explicitly stated otherwise, you may assume there is a function RANDOM that, given any positive integer k as input, returns an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, k\}$ as output, in $O(1)$ time. For example, to perform a fair coin flip, one could call RANDOM(2).

1. The following algorithm finds the second smallest element in an unsorted array:

```
RANDOM2NDMIN(A[1..n]):
  min1 ← ∞
  min2 ← ∞
  for i ← 1 to n in random order
      if A[i] < min1
          min2 ← min1          (⋆)
          min1 ← A[i]
      else if A[i] < min2
          min2 ← A[i]          (⋆⋆)
  return min2
```

(a) In the worst case, how many times does RANDOM2NDMIN execute line (⋆)?

(b) What is the probability that line (⋆) is executed during the nth iteration of the loop?

(c) What is the *exact* expected number of executions of line (⋆)?

(d) In the worst case, how many times does RANDOM2NDMIN execute line (⋆⋆)?

(e) What is the probability that line (⋆⋆) is executed during the nth iteration of the loop?

(f) What is the *exact* expected number of executions of line (⋆⋆)?

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. 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?

4. 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⟩⟩
```

(a) 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). You may assume that the stream is non-empty.

(b) Describe and analyze an algorithm that chooses k elements 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 $O(k)$ space (not counting the stream itself). You may assume that the stream has at least k elements.

5. Consider the following randomized variant of one-armed quicksort, which selects the kth smallest element in an unsorted array $A[1..n]$. As usual, $\textsc{Partition}(A[1..n], p)$ 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?

6. Suppose we are given a two-dimensional array $M[1..n, 1..n]$ in which every row and every column is sorted in increasing order and no two elements 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.

7. Suppose we are given a *sorted* circular linked list of numbers, implemented as a pair of arrays, one storing the actual numbers and the other storing successor pointers. Specifically, we are given an array $X[1..n]$ of distinct real numbers and an array $next[1..n]$ be an array of indices with the following property:

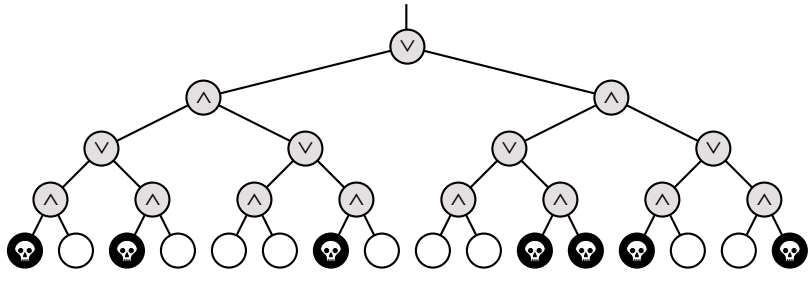
   - If $X[i]$ is the largest element of X, then $X[next[i]]$ is the smallest element of X
   - Otherwise, $X[next[i]]$ is the smallest element of X that is larger than $X[i]$.

   For example:

i	1	2	3	4	5	6	7	8	9
$X[i]$	83	54	16	31	45	99	78	62	27
$next[i]$	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 ***without modifying the input arrays***.

8. 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.



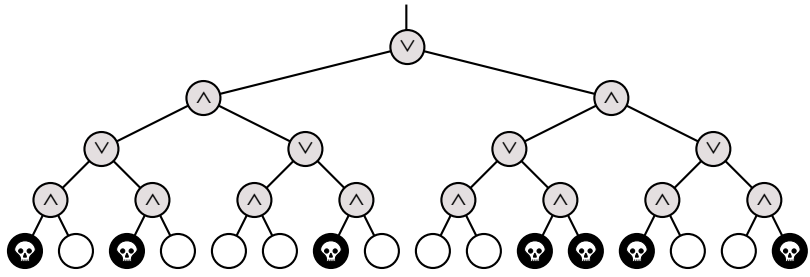
   You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand 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!]*

9. Death knocks on Dirk Gently's door one cold blustery morning and challenges him to a game. Emboldened by his experience with algorithms students, Death presents Dirk 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, Dirk 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, Dirk dies; if it's white, Dirk lives forever. Dirk moves first, so Death gets the last turn.[3]



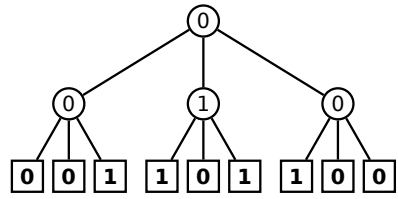
Unfortunately, Dirk slept through Death's explanation of the rules, so he decides to just play randomly. Whenever it's Dirk's turn, he flips a fair coin and moves left on heads, or right on tails, confident that the Fundamental Interconnectedness of All Things will keep him alive, unless it doesn't. Death plays much more purposefully, of course, always choosing the move that maximizes the probability that Dirk loses the game.

(a) Describe an algorithm that computes *the probability* that Dirk wins the game against Death.

(b) Realizing that Dirk is not taking the game seriously, Death gives up in desperation and decides to play randomly as well! Describe an algorithm that computes *the probability* that Dirk wins the game again Death, assuming *both* players flip fair coins to decide their moves.

10. 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 $1, 0, 0, 0, 1, 0, 1, 1, 1$, the root has value 0.

[3] Yes, this is precisely the same game from the previous exercise.



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.]*

(c) Suppose each node in the given tree has five children instead of three. Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 5$.

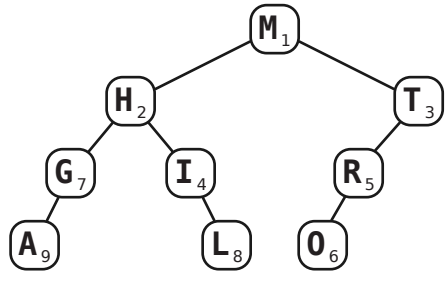
3 Randomized Binary Search Trees

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.

3.1 Treaps

3.1.1 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.

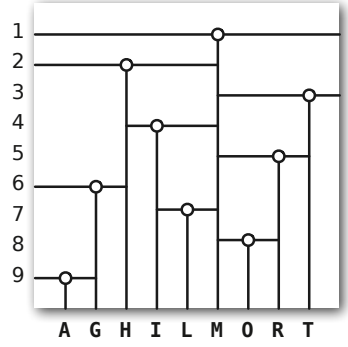
I'll assume from now on that all the keys and priorities are distinct. Under this assumption, we can easily prove by induction that the structure of a treap is completely determined by the

[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'.

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 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]

3.1.2 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

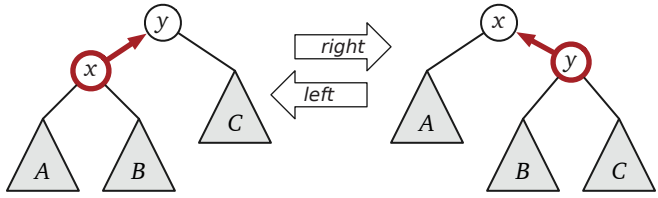
[2] J. Vuillemin, A unifying look at data structures. *Commun. ACM* 23:229–239, 1980.

[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.

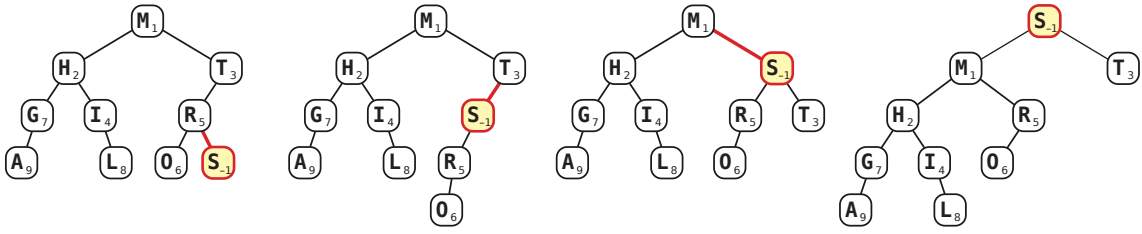
[5] C. Martínez and S. Roura. Randomized binary search trees. *J. ACM* 45(2):288-323, 1998. The results in this paper are virtually identical (including the constant factors!) to the corresponding results for treaps, although the analysis techniques are quite different.

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 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.

In the worst case, the depth of an n-node treap is $\Theta(n)$, so each of these operations has a worst-case running time of $\Theta(n)$.

3.1.3 Random Priorities

A *randomized treap* is a treap in which the priorities are *independently and uniformly distributed continuous random variables*. 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 $2^{31}-1$ and break ties arbitrarily; occasional ties have almost no practical effect on the performance of the data structure.) 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).

3.1.4 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.

```
RandomizedQuicksort:
  T ← 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.

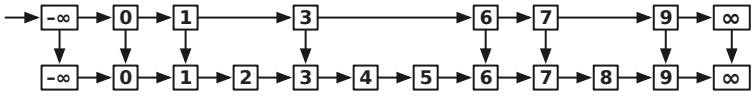
Why is this quicksort? Just like last time, all we've done is rearrange the order of the comparisons. Intuitively, the binary tree is just the recursion tree created by the normal version of quicksort. In the recursive formulation, we compare the initial pivot against everything else and then recurse. 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.

3.2 Skip Lists

Skip lists, 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 superficial 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.

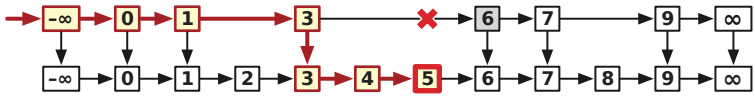


A linked list with some randomly-chosen shortcuts.

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.

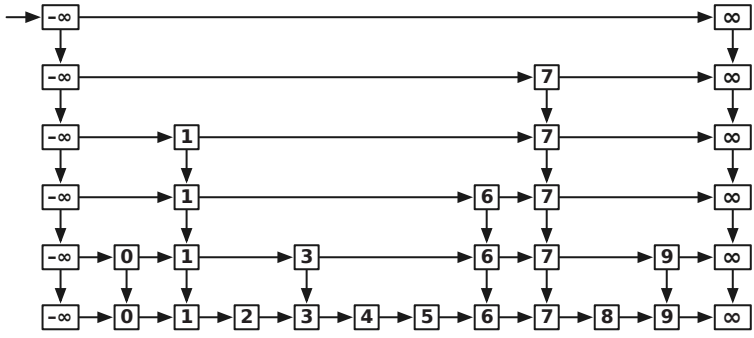


Searching for 5 in a list with shortcuts.

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.

3.2.1 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 repeatedly flip a coin until we get tails. Each time we get heads, we make a new copy 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 $down(v)$ to its next lower copy, and a pointer $right(v)$ to the next node in its level.

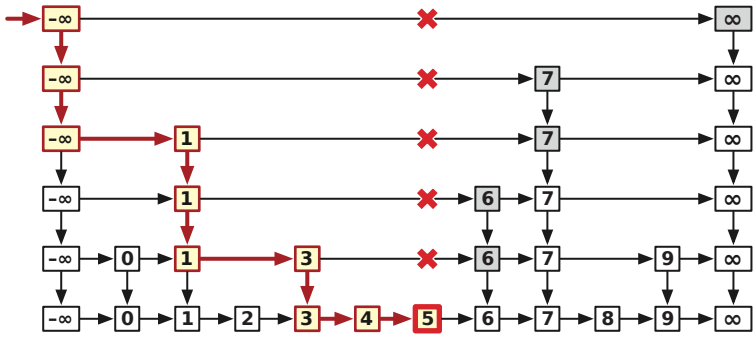


A skip list is a linked list with recursive random shortcuts.

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 $O(\log n)$ levels should give us an overall expected search time of $O(\log n)$. Let's formalize each of these two intuitive observations.



Searching for 5 in a skip list.

3.2.2 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 $\boldsymbol{\Pr[L(x) \geq \ell] = 2^{-\ell}}$. 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:

$$\begin{aligned}
\mathrm{E}[L(x)] = \sum_{\ell \geq 1} \Pr[L \geq \ell] &= \sum_{\ell=1}^{\lg n} \Pr[L \geq \ell] \; + \sum_{\ell \geq \lg n + 1} \Pr[L \geq \ell] \\
&\leq \sum_{\ell=1}^{\lg n} 1 \; + \sum_{\ell \geq \lg n + 1} \frac{n}{2^\ell} \\
&= \lg n + \sum_{i \geq 1} \frac{1}{2^i} \qquad\qquad [i = \ell - \lg n] \\
&= \mathbf{\lg n + 2}
\end{aligned}$$

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. Notice that this is an *additive* penalty over a perfectly balanced structure, as opposed to treaps, where the expected depth is a constant *multiple* of the ideal $\lg n$.

3.2.3 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 '$v \rightarrow$ down(v)' or '$v \rightarrow$ right(v)'.[8]

```
SkipListFind(v):
  while (level(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)
```

[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!

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. But we already know that the number of upward jumps is $O(\log n)$ with high probability. It follows the running time of FLIPWALK is $O(\log n)$ with high probability (and therefore in expectation).

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 above, 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) 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) The *left spine* of a binary tree is a path starting at the root and following only left-child pointers. What is the expected number of nodes in the left spine of T?

   (c) What is the expected number of leaves in T? *[Hint: What is the probability that node k is a leaf?]*

   (d) What is the expected number of nodes in T with two children?

   (e) What is the expected number of nodes in T with exactly one child?

   *(f) What is the expected number of nodes in T with exactly one *grandchild*?

   (g) Define the *priority rank* of a node in T to be one more than the number of nodes with smaller priority. For example, the root of T always has priority rank 1, and one of the children of the root has priority rank 2. What is the expected priority rank of node i?

   (h) What is the expected priority rank of the left child of the root (given that such a node exists)?

   *(i) What is the expected priority rank of the leftmost grandchild of the root (given that such a node exists)?

   *(j) What is the expected priority rank of a node with depth d?

   (k) What is the *exact* probability that node j is a common ancestor of node i and node k?

   (l) What is the *exact* expected length of the unique path in T from node i to node k?

   (m) What is the expected (key) rank of the leftmost leaf in T?

   (n) What is the expected (key) rank of the leftmost node in T with two children (given that such a node exists)?

   (o) What is the probability that T has no nodes with two children?

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 exactly $2/(i+1)$. *[Hint: Use part (a)!]*

(c) What is the exact 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.

(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. 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) The number of nodes is $O(n)$ with high probability.

(c) A new key can be inserted in $O(\log n)$ time with high probability.

(d) A key can be deleted in $O(\log n)$ time with high probability.

5. Suppose we are given two skip lists, one storing a set A of m keys, and 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 $O(n+m)$ expected time. Do *not* assume that every key in A is smaller than every key in B; the two sets could be arbitrarily intermixed. *[Hint: Do the obvious thing.]*

6. 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 the tree $T(\mathcal{L})$ should produce *exactly* the same tree as inserting x into the skip list $\mathcal{L}$ and then transforming $\mathcal{L}$ into a tree. *[Hint: You need to maintain some additional information in the tree nodes.]*

[9] If you choose not to decide, you still have made a choice.

(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.

7. Consider the following "loose" variant of treaps. Instead of generating priorities uniformly at random, for each node, we flip an independent fair coin until it comes up heads, and define the priority of the node to be the number of flips. Thus, for every positive integer k, a node has priority k with probability 2^{-k}. In addition, we invert the heap property, by requiring that the priority of any node is *not* larger than the priority of its parent.

   This method creates many nodes with the same priority; in particular, about half the nodes will have priority 1. Thus, a single set of search keys and priorities may be consistent with many different loose treaps.

   Prove that the expected depth of any node in an n-node "loose treap" is $O(\log n)$. To be pedantic, the expectation is over the random choice of priorities, but for each choice of priorities, we consider the worst possible loose treap.

   [Hint: This is almost exactly the same as the previous question.]

*8. 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

$$\textit{priority}(v) = \sum_{i=1}^{\infty} \pi_v[i] \cdot 2^{-i}.$$

   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)?]*

9. 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: What is the expected length of a random root-to-leaf path in an n-node binary tree, where 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.)

*10. Our probabilistic analysis of treaps and skip lists assumes that the sequence of operations used to query and update the data structures are independent of the random choices used to build the data structure. However, if a malicious adversary has additional information about the data structure, he can force significantly worse performance.

(a) Let $\pi[0..n^2]$ be an arbitrary permutation of the numbers $0, 1, 2, \ldots, n^2$. For each index i, define two integers:

- a_i is the length of the longest increasing subsequence of π that ends with $\pi[i]$.
- b_i be the length of the longest decreasing subsequence of π that starts with $\pi[i]$.

Argue that the $n^2 + 1$ ordered pairs (a_i, b_i) are all distinct.

(b) Prove the ***Erdős-Szekeres Theorem***: Any permutation $\pi[0..n^2]$ of the numbers $0, 1, 2, \ldots, n^2$ contains either an increasing subsequence of length $n+1$ or a decreasing subsequence of length $n + 1$.

(c) Suppose after constructing a skip list with n items, the adversary discovers the number of levels that contain each item. Prove that by deleting a subset of the nodes, the adversary can force the maximum search time to be $\Omega(\sqrt{n})$.

(d) Suppose after constructing a skip list with n items, the adversary discovers the number of levels that contain each item. Prove that by deleting a subset of the nodes, the adversary can force the maximum search time to be $\Omega(n)$ *with high probability*.

(e) Suppose after constructing a treap with n items, the adversary discovers the random priorities of each item. Prove that by deleting a subset of the nodes, the adversary can force the maximum depth of the treap to be $\Omega(\sqrt{n})$.

*4 Tail Inequalities

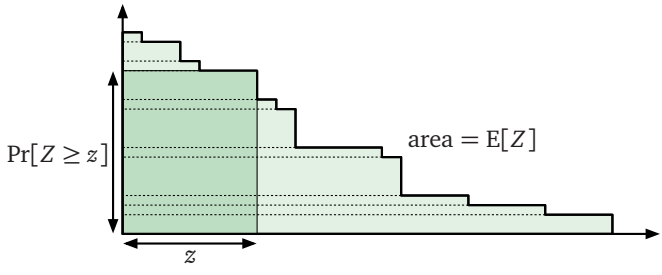
The simple recursive structure of skip lists made it relatively easy to derive an upper bound on the expected *worst-case* search time, by way of a stronger high-probability upper bound on the worst-case search time. We can prove similar results for treaps, but because of the more complex recursive structure, 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.

4.1 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 Z be a non-negative integer random variable. For any real number $z > 0$, we have* $\Pr[Z \geq z] \leq \mathrm{E}[Z]/z$.

Proof: Suppose we plot $\Pr[Z \geq z]$ as a function of z, as in the figure below. Because Z takes only integer values, the resulting curve consists of horizontal and vertical line segments. By splitting the region between this curve and the coordinate axes into horizontal rectangles, we see that the area of this region is exactly $\sum_z \Pr[Z \geq z] = \sum_z z \cdot \Pr[Z = z] = \mathrm{E}[Z]$. On the other hand, for any particular value of z, the rectangle with width z and height $\Pr[Z \geq z]$ (shaded darker in the figure) fits entirely under the curve. We conclude that $z \cdot \Pr[Z \geq z] \leq \mathrm{E}[Z]$. □



Proof of Markov's inequality.

[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. Just to be extra confusing, some sources refer to what we're calling Markov's inequality as "Chebyshev's inequality" or "Bienaymé's inequality".

In particular, Markov's inequality implies the following bound on the probability that any random variable X is significantly larger than its expectation:

$$\Pr[X \geq (1+\delta)\,\mathrm{E}[X]] \leq \frac{1}{1+\delta}.$$

Unfortunately, the bounds that Markov's inequality (directly) implies are generally too weak to be useful. (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.

4.2 Independence

Two random variables are ***independent*** if knowing the value of one variable gives us no additional knowledge about the distribution of the other. More formally, X and Y are and only if

$$\Pr[X = x \ \wedge\ Y = y] = \Pr[X = x] \cdot \Pr[Y = y],$$

or equivalently,

$$\Pr[X = x \mid Y = y] = \Pr[X = x]$$

for all possible values x and y. For example, two flips of the same (ideal) fair coin are independent, but the number of heads and the number of tails in a sequence of n coin flips are not independent (because they must add to n). Variables that are not independent are called ***dependent***. Independence of X and Y has two important consequences.

- The expectation of the product of two independent random variables is equal to the product of their expectations $\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y]$.
- If X and Y are independent, then for any function f, the random variables $f(X)$ and $f(Y)$ are also independent.

Neither of these properties hold for dependent random variables.

More generally, a collection of random variables $X_1, X_2, \ldots, X_n$ are said to be ***mutually independent*** (or ***fully 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 $x_1, x_2, \ldots, x_n$. 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:

$$\mathrm{E}\left[\prod_{i=1}^{n} X_i\right] = \prod_{i=1}^{n} \mathrm{E}[X_i].$$

Finally, if $X_1, X_2, \ldots, X_n$ are mutually independent, then for any function f, the random variables $f(X_1), f(X_2), \ldots, f(X_n)$ are also mutually independent.

In some contexts, especially in the analysis of hashing, we can only realistically assume a limited form of independence. A set of random variables is ***pairwise independent*** if every pair of variables in the set is independent. More generally, for any positive integer k, we say that a set of random variables is ***k-wise independent*** if every subset of size k is mutually independent. A set of variables is fully independent if and only if it is k-wise independent for *all* k.

Every k-wise independent set of random variables is also j-wise independent for all $j < k$, but the converse is not true. For example, let X, Y, and Z be independent random bits, and let $W = (X + Y + Z) \bmod 2$. The set of random variables $\{W, X, Y, Z\}$ is 3-wise independent but not 4-wise (or fully) independent.

4.3 Chebyshev's Inequality

Most of our analysis of randomized algorithms and data structures boils down to analyzing sums of indicator variables.[2] So far we have only been interested in the expected value of these sums, but we can prove stronger conditions under various assumptions about independence.

Consider a collection $X_1, X_2, \ldots, X_n$ of random bits, and for each index i, let $p_i = \mathrm{E}[X_i] = \Pr[X_i = 1]$. Let X denote the sum $\sum_i X_i$ and let $\mu = \mathrm{E}[X]$; linearity of expectation immediately implies that $\mu = \sum_i p_i$.

Chebyshev's Inequality. *If the indicator variables $X_1, X_2, \ldots, X_n$ are pairwise independent, then $\Pr[(X-\mu)^2 \geq z] < \mu/z$ for all $z > 0$.*

Proof: For each index i, let $Y_i := X_i - p_i$, and let $Y := \sum_i Y_i = X - \mu$.

$$\begin{aligned}
\mathrm{E}[Y^2] &= \mathrm{E}\left[\sum_{i,j} Y_i Y_j\right] && \text{[definition of } Y\text{]} \\
&= \sum_{i,j} \mathrm{E}[Y_i Y_j] && \text{[linearity]} \\
&= \sum_i \mathrm{E}[Y_i^2] + \sum_{i \neq j} \mathrm{E}[Y_i Y_j] \\
&= \sum_i \mathrm{E}[Y_i^2] + \sum_{i \neq j} \mathrm{E}[Y_i] \cdot \mathrm{E}[Y_j] && \text{[pairwise independence]} \\
&= \sum_i \mathrm{E}[Y_i^2] + 0 && \text{[}\mathrm{E}[Y_i] = 0\text{, by linearity]} \\
&= \sum_i \big((p_i(1-p_i)^2 + (1-p_i)(0-p_i)^2\big) && \text{[definition of E]} \\
&= \sum_i p_i(1-p_i) \;<\; \sum_i p_i \;=\; \mu
\end{aligned}$$

Because the random variable Y^2 is non-negative, Markov's inequality now directly implies the bound $\Pr[Y^2 \geq z] \leq \mathrm{E}[Y^2]/z < \mu/z$, which completes the proof. □

Chebyshev's inequality immediately gives us significantly tighter bounds than Markov's inequality when the component indicator variables X_i are pairwise independent. The following bounds hold for all positive real numbers Δ and δ:

$$\begin{array}{|ll|}
\hline
\Pr[X \geq \mu + \Delta] < \dfrac{\mu}{\Delta^2} & \Pr[X \geq (1+\delta)\mu] < \dfrac{1}{\delta^2 \mu} \\[2ex]
\Pr[X \leq \mu - \Delta] < \dfrac{\mu}{\Delta^2} & \Pr[X \leq (1-\delta)\mu] < \dfrac{1}{\delta^2 \mu} \\
\hline
\end{array}$$

[2] This focus on sums of indicator variables is a feature (or if you prefer, a handicap) of these lecture notes, not of randomized algorithm analysis more broadly!

The inequalities on the left are called *additive* tail bounds; the inequalities on the right are called *multiplicative* tail bounds. The inequalities in the top row bounds the *upper tail* of X's probability distribution; the inequalities in the bottom row bound the *lower tail*.

An important consequence of Chebyshev's inequality is the so-called ***Law of Large Numbers***, which states that if we repeat the same experiment many times, the statistical average of the outcomes tends toward the expected value of a single experiment with overwhelming probability. More formally, let $X_1, X_2, X_3, \ldots$ be independent random bits, with $\Pr[X_i = 1] = p$ for all i, and for any index n, let $\overline{X}_n = \sum_{i=1}^{n} X_i/n$ denote the mean of the first n bits. The ***weak*** law of large numbers states that

$$\lim_{n\to\infty} \Pr\left[|\overline{X}_n - p| > \varepsilon\right] = 0$$

for any real $\varepsilon > 0$; this law follows almost immediately from Chebyshev's inequality. The ***strong*** law of large numbers states that

$$\Pr\left[\lim_{n\to\infty} \overline{X}_n = p\right] = 1;$$

the proof of this law is considerably more involved.

4.4 Higher Moment Inequalities

Chebyshev's inequality and its corollaries are special cases of a more general family of so-called ***moment inequalities***, which consider the distribution of higher even powers of the random variable X under stronger assumptions about the independence of the components X_i. Here I'll state the most general result without proof; the derivation follows the same general outline as the proof of Chebyshev's inequality above, but with more, you know, math.

General Moment Inequality. *For any fixed integer $k > 0$, if the variables $X_1, X_2, \ldots, X_n$ are $2k$-wise independent, then $\Pr[(X-\mu)^k \geq z] = O(\mu^k/z)$ for all $z > 0$. The hidden constant in the $O(\cdot)$ notation depends on k.*

This inequality immediately implies the following asymptotic tail bounds for all positive real numbers Δ and δ; again, the hidden $O(\cdot)$ constants depend on the fixed independence parameter k.

$$\boxed{\begin{array}{ll} \Pr[X \geq \mu + \Delta] = O\left(\left(\frac{\mu}{\Delta^2}\right)^k\right) & \Pr[X \geq (1+\delta)\mu] = O\left(\left(\frac{1}{\delta^2\mu}\right)^k\right) \\ \Pr[X \leq \mu - \Delta] = O\left(\left(\frac{\mu}{\Delta^2}\right)^k\right) & \Pr[X \geq (1-\delta)\mu] = O\left(\left(\frac{1}{\delta^2\mu}\right)^k\right) \end{array}}$$

In short, the more independence we can assume about the indicator variables X_i, the higher the degree of the polynomial tails in the distribution of X, and the more likely that X stays close to its expected value.

4.5 Chernoff Bounds

For *fully* independent random variables, even tighter tail bounds were developed in the early 1950s by Herman Chernoff, who was then a mathematics professor at the University of Illinois. However, in strict accordance with Stigler's Law, the first published version of "Chernoff bounds",

which appeared in a 1952 paper by Chernoff that gave the bounds their name, were actually due to his colleague Herman Rubin.[3]

Exponential moment inequality. *If the indicator variables $X_1, X_2, \ldots, X_n$ are fully independent, then $E[\alpha^X] \le e^{(\alpha-1)\mu}$ for any $\alpha \ge 1$.*

Proof: The definition of expectation immediately implies that

$$E[\alpha^{X_i}] \;=\; p_i\alpha^1 + (1-p_i)\alpha^0 \;=\; (\alpha-1)p_i + 1.$$

Thus, by The World's Most Useful Inequality $1 + t \le e^t$, we have

$$E[\alpha^{X_i}] \;\le\; e^{(\alpha-1)p_i}.$$

Full independence of the X_i's now immediately implies

$$E[\alpha^X] \;=\; \prod_i E[\alpha^{X_i}] \;\le\; \prod_i e^{(\alpha-1)p_i} \;=\; e^{(\alpha-1)\mu}$$

and we're done. □

Chernoff bound (upper tail). *If the indicator variables $X_1, X_2, \ldots, X_n$ are fully independent, then $\Pr[X \ge x] \le e^{x-\mu}(\mu/x)^x$ for all $x \ge \mu$.*

Proof: Consider any fixed value $x \ge \mu$. The function $t \mapsto (x/\mu)^t$ is monotonically increasing, so

$$\Pr[X \ge x] = \Pr\left[\left(\frac{x}{\mu}\right)^X \ge \left(\frac{x}{\mu}\right)^x\right].$$

Markov's inequality implies that

$$\Pr\left[\left(\frac{x}{\mu}\right)^X \ge \left(\frac{x}{\mu}\right)^x\right] \;\le\; \frac{E[(x/\mu)^X]}{(x/\mu)^x}.$$

Finally, applying the exponential moment inequality at $\alpha = x/\mu$ gives us

$$E[(x/\mu)^X] \le e^{((x/\mu)-1)\mu} = e^{x-\mu},$$

which completes the proof. □

A nearly identical argument implies a similar bound on the lower tail:

Chernoff bound (lower tail). *If the indicator variables $X_1, X_2, \ldots, X_n$ are fully independent, then $\Pr[X \le x] \le e^{x-\mu}(\mu/x)^x$ for all $x \le \mu$.*

Proof: For any $x \le \mu$, we immediately have $\Pr[X \le x] = \Pr[(x/\mu)^X \ge (x/\mu)^x]$. (The direction of the inequality changes because $x/\mu \le 1$.) The remainder of the proof is unchanged. □

[3] "Since that seemed to be a minor lemma in the ensuing paper I published (Chernoff, 1952), I neglected to give [Rubin] credit. I now consider it a serious error in judgment, especially because his result is stronger, for the upper bound, than the asymptotic result I had derived."

The particular value x/μ in these results may seem arbitrary, but in fact it's chosen vary carefully. The same arguments imply that

$$\Pr[X \geq x] \leq \frac{e^{(\alpha-1)\mu}}{\alpha^x} \text{ for } \textit{all } \alpha > 1 \qquad \text{and} \qquad \Pr[X \leq x] \leq \frac{e^{(\alpha-1)\mu}}{\alpha^x} \text{ for } \textit{all } \alpha < 1.$$

A bit of calculus implies that the right sides of these inequalities are minimized when $\alpha = x/\mu$.

Direct substitution now implies the following more traditional forms of Chernoff bounds, for any positive reals Δ and δ. Unlike the polynomial moment bounds we derived earlier, Chernoff bounds for the upper and lower tails of X are asymmetric.

$$\boxed{\begin{aligned} \Pr[X \geq \mu + \Delta] &\leq e^{\Delta}\left(\frac{\mu}{\mu+\Delta}\right)^{\mu+\Delta} & \Pr[X \geq (1+\delta)\mu] &\leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \\ \Pr[X \leq \mu - \Delta] &\leq e^{-\Delta}\left(\frac{\mu}{\mu-\Delta}\right)^{\mu-\Delta} & \Pr[X \leq (1-\delta)\mu] &\leq \left(\frac{e^{-\delta}}{(1-\delta)^{1-\delta}}\right)^{\mu} \end{aligned}}$$

It is sometimes more convenient to work with the following looser forms of the multiplicative Chernoff bounds, which hold for any $0 < \delta < 1$. Here we use the notation $\exp(t) := e^t$.

$$\boxed{\begin{aligned} \Pr[X \geq (1+\delta)\mu] &\leq \exp\left(-\delta^2\mu/3\right) \\ \Pr[X \leq (1-\delta)\mu] &\leq \exp\left(-\delta^2\mu/2\right) \end{aligned}}$$

These looser bounds can be derived from the more precise forms above using the Taylor-series approximations $1/\ln(1+\delta) \leq 1/2 + 1/\delta$ and $\ln(1-\delta) \geq -\delta + \delta^2/2$. I'll omit the straightforward but grungy details. (You're welcome.)

4.6 Coin Flips

To give an elementary example of applying these tail inequalities, consider an experiment where we independently flip N perfectly fair coins, and we want to bound the probability of getting at least αN heads, for some constant $1/2 < \alpha < 1$.

For each index i, let $X_i = 1$ if the ith flip comes up heads and $X_i = 0$ otherwise, so that $X = \sum_{i=1}^{N} X_i$ is the number of heads. Because the coins are fair, $\Pr[X_i = 1] = 1/2$ for all i. Linearity of expectation immediately implies $\mu = \mathrm{E}[X] = N/2$.

- Markov's inequality implies the nearly trivial bound $\Pr[X \geq \alpha N] \leq 1/2\alpha$. In particular, we have $\Pr[X \geq 3N/4] \leq 2/3$ and $\Pr[X \geq 9N/10] \leq 5/9$.
- Chebyshev's inequality implies $\Pr[X \geq \alpha N] \leq 1/((2\alpha-1)N) = O(1/(\alpha N))$. In particular, we have $\Pr[X \geq 3N/4] \leq 2/N$ and $\Pr[X \geq 9N/10] \leq 5/(4N)$.
- Higher moment inequalities implies smaller inverse-polynomial upper bounds. For example, the fourth-moment inequality implies $\Pr[X \geq \alpha N] \leq O(1/(\alpha N)^2)$.
- The simple version of Chernoff's inequality implies the inverse-exponential upper bound

$$\Pr[X \geq \alpha N] \leq \exp\left(-\frac{(2\alpha-1)^2}{6} \cdot N\right).$$

  In particular, we have $\Pr[X \geq 3N/4] < e^{-N/24}$ and $\Pr[X \geq 9N/10] < e^{-8N/75} < e^{-N/10}$.

- Symmetrically, Chernoff's inequality for the *lower* tail symmetrically bounds the probability of getting *few* heads as follows:

$$\Pr[X \le \alpha N] \le \exp\left(-\frac{(1-2\alpha)^2}{4}N\right).$$

  In particular, we have $\Pr[X \le N/4] < e^{-N/16}$ and $\Pr[X \le N/10] < e^{-4N/25} \le e^{-N/7}$.

- A more specialized argument involving binomial coefficients, which relies on all coins being fair, implies

$$\Pr[X \le \alpha N] \le \left(2\alpha^{\alpha}(1-\alpha)^{1-\alpha}\right)^{-N} \quad \text{and} \quad \Pr[X \ge \alpha N] \le \left(2\alpha^{\alpha}(1-\alpha)^{1-\alpha}\right)^{-N}.$$

  In particular, we have

$$\Pr[X \le N/4] \;\le\; \left(2\left(\frac{1}{4}\right)^{1/4}\left(\frac{3}{4}\right)^{3/4}\right)^{-N} \;\approx\; e^{-0.13081N} \;<\; e^{-N/8}$$

  and

$$\Pr[X \le N/10] \;\le\; \left(2\left(\frac{1}{10}\right)^{1/10}\left(\frac{9}{10}\right)^{9/10}\right)^{-N} \;\approx\; e^{-0.36806N} \;<\; e^{-N/3}.$$

4.7 Back to Treaps, Take 1

Consider an arbitrary node k in an n-node treap. We can argue informally that $\mathit{depth}(k) = O(\log n)$ with high probability by following the "good/bad pivot" analysis of randomized quicksort as follows.

For each integer $0 \le i \le \mathit{depth}(k)$, let v_i denote the ancestor of node k at depth i in the treap, and let n_i denote the number of nodes in the subtree rooted at v_i. (Thinking in terms of randomized quicksort, n_i is the size of the subproblem containing k after i levels of recursion.) For any integer $i > 0$, we label level i ***good*** or ***bad*** as follows:

- If $i < \mathit{depth}(k)$, then level i is good if $n_{i-1}/4 \le n_i \le 3n_{i-1}/4$ and bad otherwise.
- If $i \ge \mathit{depth}(k)$, then level i is good or bad according to an independent fair coin flip.

Each level $i \ge 1$ is good with probability at least $1/2$, and the events $[i \text{ is good}]$ are mutually independent. There are at most $\log_{4/3} n$ good levels smaller than $\mathit{depth}(k)$, since otherwise $n_{\mathit{depth}(k)}$ would be less than 1.

Thus, the depth of node k is *at most* the number of fair coin flips required to get $\log_{4/3} n$ heads. Equivalently, for any constant c, the probability that $\mathit{depth}(k) \ge c \log_{4/3} n$ is at most the probability that we see at most $\log_{4/3} n$ heads in a sequence of $c \log_{4/3} n$ independent fair coin flips. Our earlier analysis of coin flips via Chernoff bounds, with $N = c \log_{4/3} n$ and $\alpha = 1/c$, implies

$$\Pr\left[\mathit{depth}(k) \ge c \log_{4/3} n\right] \le \exp\left(-\frac{(1-2/c)^2}{4} c \log_{4/3} n\right) = n^{-(c-2)^2/(4c\ln(4/3))}.$$

For any $c > 2$, we obtain an inverse-polynomial upper bound, which we can make arbitrarily small by increasing the constant c. For example, setting $c = 20\ln(4/3) \approx 5.75364$ gives us $\Pr[\mathit{depth}(k) > 20 \ln n] < 1/n^2$. We conclude that $\mathit{depth}(k) = O(\log n)$ with high probability.

Our more specialized argument implies that

$$\Pr\left[\mathit{depth}(k) \ge 10 \log_{4/3} n\right] < e^{-10 \log_{4/3} n/3} = n^{-10/(3\ln(4/3))} < \frac{1}{n^{11}}.$$

4.8 Back to Treaps: Take 2

In our analysis of randomized treaps, we wrote $\boldsymbol{i \uparrow k}$ 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

$$\mathrm{E}[\mathit{depth}(k)] = \sum_{i=1}^{n} \Pr[i \uparrow k] = H_k + H_{n-k} - 2 < 2\ln n.$$

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, and the $n-k$ random variables $[i \uparrow k]$ with $i > k$ are mutually independent.*

Proof: Fix an arbitrary index k. We explicitly consider only the indicators $[i \uparrow k]$ with $i > k$; the proof for $i < k$ is nearly identical.

To simplify notation, let X_i denote the indicator variable $[i \uparrow k]$. We will prove by induction on n that the $n-k$ indicator variables $X_{k+1}, X_{k+2}, \ldots, X_n$ are mutually independent, starting with the vacuous base case $n = k$. Fix $n-k$ arbitrary indicator values $x_{k+1}, x_{k+2}, \ldots, x_n$. The definition of conditional probability immediately implies that

$$\begin{aligned}
\Pr\left[\bigwedge_{i=k+1}^{n} (X_i = x_i)\right] &= \Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i) \wedge X_n = x_n\right] \\
&= \Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i) \,\middle|\, X_n = x_n\right] \cdot \Pr\left[X_n = x_n\right]
\end{aligned}$$

Now recall that $X_n = [n \uparrow k] = 1$ if and only if node n has the smallest priority of all nodes between k and n. The other $n-k-1$ indicator variables X_i depend only on the order of the priorities of nodes k through $n-1$. There are exactly $(n-k-1)!$ permutations of the $n-k$ priorities in which the nth priority is smallest, and each of these permutations is equally likely. Thus,

$$\Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i) \,\middle|\, X_n = x_n\right] = \Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i)\right]$$

The inductive hypothesis implies that the variables $X_{k+1}, \ldots, X_{n-1}$ are mutually independent, so

$$\Pr\left[\bigwedge_{i=k+1}^{n-1} (X_i = x_i)\right] = \prod_{i=k+1}^{n-1} \Pr[X_i = x_i].$$

We conclude that

$$\Pr\left[\bigwedge_{i=k+1}^{n} (X_i = x_i)\right] = \Pr\left[X_n = x_n\right] \cdot \prod_{i=k+1}^{n-1} \Pr[X_k = x_k] = \prod_{i=k+1}^{n} \Pr[X_i = x_i],$$

or in other words, that the variables $X_i = [i \uparrow k]$ are indeed mutually independent. □

For purposes of analysis, define a new sequence $Z_2, Z_3, \ldots, Z_n$ of independent indicator variables, where $\Pr[Z_j = 1] = 1/j$ for each j, and let $Z := \sum_{j=2}^{n} Z_j$.

Lemma 2. $\Pr[Z > 4\ln n] < 1/n^2$.

Proof: Let $\mu = \mathrm{E}[Z] = H_n - 1 < \ln n$ and $\delta = c - 1$. Since Z is a sum of independent indicator variables, Chernoff's inequality implies

$$\begin{aligned}
\Pr[Z > 4\ln n] &< \Pr[Z > 4\mu] \\
&= \Pr[Z \geq (1+\delta)\mu] \\
&\leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \\
&= \left(\frac{e^3}{4^4}\right)^\mu < \left(\frac{e^3}{4^4}\right)^{\ln n} = n^{3-4\ln 4} \approx n^{-2.54518} < \frac{1}{n^2}.
\end{aligned}$$

□

Theorem 3. *An n-node treap has depth $O(\log n)$ with high probability.*

Proof: Specifically, I will bound the probability that the depth is at most $8\ln n$. The union bound $\Pr[A \vee B] \leq \Pr[A] + \Pr[B]$ implies

$$\Pr\left[\max_k depth(k) > 8\ln n\right] = \Pr\left[\bigvee_{k=1}^{n} (depth(k) > 8\ln n)\right] \leq \sum_{k=1}^{n} \Pr[depth(k) > 8\ln n].$$

The depth of any node k is a sum of n indicator variables $[i \uparrow k]$, as i ranges from 1 to n. Lemma **??** partitions these variables into two mutually independent subsets. Define

$$depth_<(k) := \sum_{i=1}^{k-1} [i \uparrow k] \qquad \text{and} \qquad depth_>(k) := \sum_{i=k+1}^{n} [i \uparrow k],$$

so that $depth(k) = depth_<(k) + depth_>(k)$. The union bound implies

$$\Pr[depth(k) > 8\ln n] \leq \Pr[depth_<(k) > 4\ln n] + \Pr[depth_>(k) > 4\ln n].$$

For all indices $i \neq k$, the indicator variable $[i \uparrow k]$ has the same distribution as the variable $Z_{|i-k|+1}$ defined at the top of this page. Thus, for each k we have

$$\Pr[depth_<(k) > 4\ln n] = \Pr\left[\sum_{i=1}^{\ell-1} Z_\ell > 4\ln n\right] \leq \Pr[Z > 4\ln n] < 1/n^2.$$

Symmetrically, $\Pr[depth_<(k) > 4\ln n] \leq \Pr[Z > 4\ln n] < 1/n^2$. We immediately conclude that

$$\Pr\left[\max_k depth(k) > 8\ln n\right] \leq 2n \Pr[Z > 4\ln n] < 2/n,$$

which completes the proof. □

By varying the constants in the previous argument, we can prove 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 now finally conclude 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. (a) Prove that for any index k such that $1 < k < n$, the $n-1$ indicator variables $[i \uparrow k]$ with $i \neq k$ are *not* mutually independent. *[Hint: Consider the case $n = 3$ and $k = 2$.]*

   (b) Prove that for any index k, the $k-1$ random variables $[k \uparrow i]$ with $i < k$ are *not* mutually independent. *[Hint: Consider the case $k = 4$.]*

2. (a) Consider an arbitrary node k in an n-node treap. Prove that the expected number of descendants of node k is *precisely* equal to the expected number of ancestors of node k.

   (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 clearly bogus—Every treap has a node with n descendants!—but what's the hole in the argument?

3. The following algorithm finds 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 ← min1        (⋆)
      return min
```

   (a) Let X_i be the indicator variable that equals 1 if line $(\star)$ is executed in the ith iteration of the main loop. Prove that the variables X_i are mutually independent.

   (b) Prove that line $(\star)$ is executed $O(\log n)$ time with high probability.

4. 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. Suppose each point in S is chosen independently and uniformly at random from the unit square $[0,1] \times [0,1]$. Prove that the number of Pareto-optimal points in S is $O(\log n)$ with high probability.

5. 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.

6. Let $X_1, X_2, \ldots, X_n$ be independent indicator variables, each equal to 1 with probability $1/2$, and let $X = \sum_i X_i$. Clearly $\mathrm{E}[X] = n/2$.

   (a) Let $Y = (X - \mathrm{E}[X])^2$. What is $\mathrm{E}[Y]$?

   (b) Let $Z = (Y - \mathrm{E}[Y])^2$. What is $\mathrm{E}[Z]$?

   (c) Let $\Omega = (Z - \mathrm{E}[Z])^2$. What is $\mathrm{E}[\Omega]$?

5 Hash Tables

5.1 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 an array at index $h(x)$; the array itself 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 $u = m$, we can always just use the trivial hash function $h(x) = x$; in other words, we can use the item itself as the index into the table. The resulting data structure is called a *direct access table*, or more commonly, an *array*. In most applications, however, 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 need to store, not the number of items that we might possibly 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 results in as few collisions as possible? Second, when collisions do occur, how do we resolve them?

5.2 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 in advance 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 $\mathcal{U}$ *must* collide. In fact, for any fixed hash function, there is a subset of at least $|\mathcal{U}|/m$ 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 $\mathcal{MB}^+$ 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 $\mathcal{MB}^+$ according to some fixed distribution. Different sets $\mathcal{MB}^+$ 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? That's just crazy!

5.3 ...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:

$$\textbf{Uniform: } \Pr_{h\in\mathcal{H}}\big[h(x) = i\big] = \frac{1}{m} \quad \text{for all } x \text{ and all } i$$

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:

$$\textbf{Universal: } \Pr_{h\in\mathcal{H}}\big[h(x) = h(y)\big] \le \frac{1}{m} \quad \text{for all } x \neq y$$

(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:

$$\textbf{Near-universal: } \Pr_{h\in\mathcal{H}}\big[h(x) = h(y)\big] \le \frac{2}{m} \quad \text{for all } x \neq y$$

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 $1/m^k$:

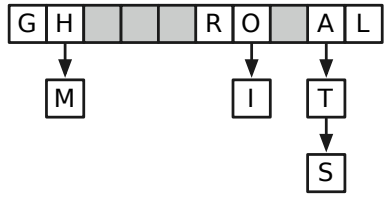
$$\textbf{k-uniform: } \Pr_{h\in\mathcal{H}}\left[\bigwedge_{j=1}^{k} h(x_j) = i_j\right] = \frac{1}{m^k} \quad \text{for all distinct } x_1, \ldots, x_k \text{ and all } i_1, \ldots, i_k$$

Ideal random hash functions are k-uniform for every positive integer k.

[1] Confusingly, universality is often called the ***uniform hashing assumption***, even though it is not an assumption that the hash function is uniform.

5.4 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 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 assuming near-universal hashing, 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 that the expected *worst-case* search time is constant! The expected worst-case search time is $O(1 + L)$, where $L = \max_x \ell(x)$. Even with *ideal* random hash functions, 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 overall distribution of those items is *almost never* 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.

5.5 Multiplicative Hashing

Arguably the simplest technique for near-universal hashing, first described by Lawrence Carter and Mark Wegman in the late 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 $[\![n]\!]$ denote the n-element set $\{0, 1, \ldots, n-1\}$, and let $[\![n]\!]^+$ denote the $(n-1)$-element set $\{1, 2, \ldots, n-1\}$.

5.5.1 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 $multp_a : \mathcal{U} \to [\![m]\!]$ by setting

$$\boxed{multp_a(x) = (ax \bmod p) \bmod m}$$

and let

$$\mathcal{MP} := \left\{ multp_a \;\middle|\; a \in [\![p]\!]^+ \right\}$$

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 near-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 $a \in [\![p]\!]^+$, there is a unique integer $z \in [\![p]\!]^+$ such that $az \bmod p = 1$.*

Proof: Fix an arbitrary integer $a \in [p]^+$.

Suppose $az \bmod p = az' \bmod p$ for some integers $z, z' \in [p]^+$. We immediately have $a(z-z') \bmod p = 0$, which implies that $a(z-z')$ is divisible by p. Because p is prime, the inequality $1 \le a \le p-1$ implies that $z-z'$ must be divisible by p. Similarly, because $1 \le z, z' \le p-1$, we have $2-p \le z-z' \le p-2$, which implies that $z = z'$. It follows that for each integer $h \in [p]^+$, there is *at most* one integer $z \in [p]^+$ such that $az \bmod p = h$.

Similarly, if $az \bmod p = 0$ for some integer $z \in [p]^+$, then because p is prime, either a or z is divisible by p, which is impossible.

We conclude that the set $\{az \bmod p \mid z \in [p]^+\}$ has exactly $p-1$ distinct elements, all non-zero, and therefore is equal to $[p]^+$. In other words, multiplication by a defines a permutation of $[p]^+$. The lemma follows immediately. □

Let $\boldsymbol{a^{-1}}$ denote the multiplicative inverse of a, as guaranteed by the previous lemma. We can now precisely characterize when the hash values of two items collide.

Lemma 2. *For any elements $a, x, y \in [p]^+$, we have a collision $multp_a(x) = multp_a(y)$ if and only if either $x = y$ or $multp_a((x-y) \bmod p) = 0$ or $multp_a((y-x) \bmod p) = 0$.*

Proof: Fix three arbitrary elements $a, x, y \in [p]^+$. There are three cases to consider, depending on whether $ax \bmod p$ is greater than, less than, or equal to $ay \bmod p$.

First, suppose $ax \bmod p = ay \bmod p$. Then $x = a^{-1}ax \bmod p = a^{-1}ay \bmod p = y$, which implies that $x = y$. (This is the only place we need primality.)

Next, suppose $ax \bmod p > ay \bmod p$. We immediately observe that

$$ax \bmod p - ay \bmod p \;=\; (ax - ay) \bmod p \;=\; a(x-y) \bmod p.$$

Straightforward algebraic manipulation now implies that $multp_a(x) = multp_a(y)$ if and only if $multp_a((x-y) \bmod p) = 0$.

$$\begin{aligned} multp_a(x) = multp_a(y) &\iff (ax \bmod p) \bmod m = (ay \bmod p) \bmod m \\ &\iff (ax \bmod p) - (ay \bmod p) \equiv 0 \pmod m \\ &\iff a(x-y) \bmod p \equiv 0 \pmod m \\ &\iff multp_a((x-y) \bmod p) = 0 \end{aligned}$$

Finally, if $ax \bmod p < ay \bmod p$, an argument similar to the previous case implies that $multp_a(x) = multp_a(y)$ if and only if $multp_a((y-x) \bmod p) = 0$. □

For any distinct integers $x, y \in \mathcal{U}$, Lemma **??** immediately implies that

$$\begin{aligned} &\Pr_a\big[multp_a(x) = multp_a(y)\big] \\ &\qquad \le \Pr_a\big[multp_a((x-y) \bmod p) = 0\big] + \Pr_a\big[multp_a((y-x) \bmod p) = 0\big]. \end{aligned}$$

Thus, to show that $\mathcal{MP}$ is near-universal, it suffices to prove the following lemma.

Lemma 3. *For any integer $z \in [p]^+$, we have $\Pr_a[multp_a(z) = 0] \le 1/m$.*

Proof: Fix an arbitrary integer $z \in [p]^+$. Lemma 1 implies that for any integer $h \in [p]^+$, there is a unique integer $a \in [p]^+$ such that $(az \bmod p) = h$; specifically, $a = h \cdot z^{-1} \bmod p$. 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 $multp_a(z) = 0$. □

Our analysis of collision probability can be improved, but only slightly. Carter and Wegman observed that if $p \bmod (m+1) = 1$, then $\Pr_a[multp_a(1) = multp_a(m+1)] = 2/(m+1)$. (For any positive integer m, there are infinitely many primes p such that $p \bmod (m+1) = 1$.) For example, by enumerating all possible values of $multp_a(x)$ when $p = 5$ and $m = 3$, we immediately observe that $\Pr_a[multp_a(1) = multp_a(4)] = 1/2 = 2/(m+1) > 1/3$.

	1	2	3	4
0	0	0	0	0
1	1	2	0	1
2	2	1	1	0
3	0	1	1	2
4	1	0	2	1

5.5.2 Actually universal hashing

Our first example of a truly universal family of hash functions uses a small modification of the multiplicative method we just considered. For any integers $a \in [p]^+$ and $b \in [p]$, let $h_{a,b} : \mathcal{U} \to [m]$ be the function

$$\boxed{h_{a,b}(x) = ((ax + b) \bmod p) \bmod m}$$

and let

$$\mathcal{MB}^+ := \{h_{a,b} \mid a \in [p]^+, b \in [p]\}$$

denote the set of all $p(p-1)$ such functions. A function in this family is specified by *two* salt parameters a and b.

Theorem 1. *$\mathcal{MB}^+$ is universal.*

Proof: Fix four integers $r, s, x, y \in [p]$ such that $x \neq y$ and $r \neq s$. The linear system

$$\begin{aligned} ax + b &\equiv r \pmod p \\ ay + b &\equiv s \pmod p \end{aligned}$$

has a unique solution $a, b \in [p]$ with $a \neq 0$, namely

$$\begin{aligned} a &= (r-s)(x-y)^{-1} \bmod p \\ b &= (sx - ry)(x-y)^{-1} \bmod p \end{aligned}$$

where z^{-1} denotes the mod-p multiplicative inverse of z, as guaranteed by Lemma 1. It follows that

$$\Pr_{a,b}\big[(ax+b) \bmod p = r \text{ and } (ay+b) \bmod p = s\big] = \frac{1}{p(p-1)},$$

and therefore

$$\Pr_{a,b}\big[h_{a,b}(x) = h_{a,b}(y)\big] = \frac{N}{p(p-1)},$$

where N is the number of ordered pairs $(r,s) \in [p]^2$ such that $r \neq s$ but $r \bmod m = s \bmod m$. For each fixed $r \in [p]$, there are at most $\lfloor p/m \rfloor$ integers $s \in [p]$ such that $r \neq s$ but $r \bmod m = s \bmod m$. Because p is prime, we have $\lfloor p/m \rfloor \leq (p-1)/m$. We conclude that $N \leq p(p-1)/m$, which completes the proof. □

More careful analysis implies that the collision probability for any pair of items is exactly

$$\frac{(p - p \bmod m)(p - (m - p \bmod m))}{mp(p-1)}.$$

Because p is prime, we must have $0 < p \bmod m < m$, so this probability is actually *strictly less than* $1/m$. For example, when $p = 5$ and $m = 3$, the collision probability is

$$\frac{(5 - 5 \bmod 3)(5 - (3 - 5 \bmod 3))}{3 \cdot 4 \cdot 5} = \frac{1}{5} < \frac{1}{3},$$

which we can confirm by enumerating all possible values:

$b = 0$

	1	2	3	4
0	0	0	0	0
1	**1**	2	0	**1**
2	2	1	1	0
3	0	1	1	2
4	**1**	0	2	**1**

$b = 1$

	1	2	3	4
1	1	1	1	1
1	2	0	1	0
2	0	0	2	1
3	1	2	0	0
4	0	1	0	2

$b = 2$

	1	2	3	4
0	2	2	2	2
1	0	1	0	1
2	1	1	0	0
3	0	0	1	1
4	1	0	1	0

$b = 3$

	1	2	3	4
0	0	0	0	0
1	1	0	1	2
2	0	2	1	1
3	1	1	2	0
4	2	1	0	1

$b = 4$

	1	2	3	4
0	1	1	1	1
1	**0**	1	2	**0**
2	1	0	0	2
3	2	0	0	1
4	**0**	2	1	**0**

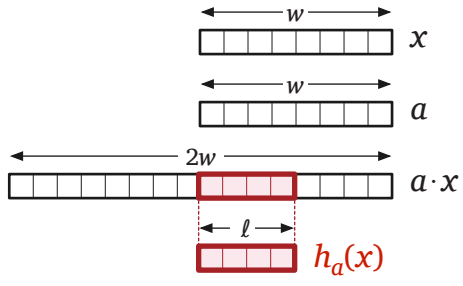
5.5.3 Binary multiplicative hashing

A slightly simpler variant of multiplicative hashing that avoids the need for large prime numbers was first formally analyzed by Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen in 1997, although it was proposed decades earlier. 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 $multb_a : \mathcal{U} \to [m]$ as follows:

$$multb_a(x) := \left\lfloor \frac{(a \cdot x) \bmod 2^w}{2^{w-\ell}} \right\rfloor$$

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 4. *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. □

Theorem 2. *$\mathcal{MB}$ 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 $x \neq y$, 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. □

*5.6 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, and the probability of any particular subset of k balls landing in bin j is $1/n^k$, so the union bound ($\Pr[A \vee B] \le \Pr[A] + \Pr[B]$ for any events A and B) implies

$$\Pr[X_j \ge k] \;\le\; \binom{n}{k}\left(\frac{1}{n}\right)^k \;\le\; \frac{n^k}{k!}\left(\frac{1}{n}\right)^k \;=\; \frac{1}{k!}.$$

Setting $k = 2c\lg n / \lg\lg n$, we have

$$k! \ge k^{k/2} = \left(\frac{2c\lg n}{\lg\lg n}\right)^{2c\lg n/\lg\lg n} \ge \left(\sqrt{\lg n}\right)^{2c\lg n/\lg\lg n} = 2^{c\lg n} = n^c,$$

which implies that

$$\Pr\left[X_j \ge \frac{2c\lg n}{\lg\lg n}\right] < \frac{1}{n^c}.$$

This probability bound holds for every bin j. Thus, by the union bound, we conclude that

$$\Pr\left[\max_j X_j > \frac{2c\lg n}{\lg\lg n}\right] \;=\; \Pr\left[X_j > \frac{2c\lg n}{\lg\lg n} \text{ for all } j\right] \;\le\; \sum_{j=1}^{n} \Pr\left[X_j > \frac{2c\lg n}{\lg\lg n}\right] \;<\; \frac{1}{n^{c-1}}. \qquad \square$$

A somewhat more complicated argument implies that if we throw n balls randomly into n bins, then with high probability, the fullest bin contains at least $\Omega(\log n / \log\log n)$ balls.

However, if we make the hash table sufficiently large, 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 $1/2$, and thus by Markov's inequality, the probability of getting *even one* collision is less than $1/2$.

We can give a slightly weaker version of this bound 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 collisions is

$$\sum_{x<y} \Pr[h(x) = h(y)] \le \binom{n}{2}\frac{2}{m} = \frac{n(n-1)}{m}.$$

In particular, if we set $m = 2n^2$, the expected number of collisions is less than $1/2$. Again, Markov's inequality implies that the probability of *even one* collision is less than $1/2$.

If we make the hash table slightly larger, we can even prove a high-probability bound.

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.*

Proof: 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 $1 - 1/n^{\varepsilon}$. □

5.7 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 $m = n$ and a near-universal hash function, 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* $\mathrm{E}\left[\sum_i n_i^2\right] < 3n$.

Proof: let $h(x)$ denote the position of x in the primary hash table. We can rewrite the sum $\sum_i n_i^2$ 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.

$$\begin{aligned}\sum_i n_i^2 &= \sum_i \left(\sum_x [h(x) = i]\right)^2 \\ &= \sum_i \left(\sum_x \sum_y [h(x) = i][h(y) = i]\right)\end{aligned}$$

$$
\begin{aligned}
&= \sum_i \left(\sum_x [h(x) = i]^2 \;+\; 2 \sum_{x<y} [h(x) = i][h(y) = i] \right) \\
&= \sum_x \sum_i [h(x) = i]^2 \;+\; 2 \sum_{x<y} \sum_i [h(x) = i][h(y) = i] \\
&= \sum_x \sum_i [h(x) = i] \;+\; 2 \sum_{x<y} [h(x) = h(y)]
\end{aligned}
$$

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 $O(1)$.

5.8 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 $\langle h_0, h_1, h_2, \ldots, h_{m-1} \rangle$, such that for any item x, the *probe sequence* $\langle h_0(x), h_1(x), \ldots, h_{m-1}(x) \rangle$ is a permutation of $\langle 0, 1, 2, \ldots, m-1 \rangle$. 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 each 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:** For each item x and index i, the hash value $h_i(x)$ is equally likely to be any integer in the set $\{0, 1, 2, \ldots, m-1\}$.
- **Indpendence:** For each item x, if we ignore the first probe $h_0(x)$, 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 $\{0, 1, 2, \ldots, m-1\} \setminus \{h_0(x)\}$.

Uniformity implies that the probability that $T[h_0(x)]$ is occupied is exactly n/m. Independence 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 $\mathbf{E[}\boldsymbol{T(m,n)}\mathbf{] \le m/(m-n)}$:

$$\begin{aligned}
\mathrm{E}[T(m,n)] &= 1 + \frac{n}{m}\,\mathrm{E}[T(m-1,n-1)] \\
&\le 1 + \frac{n}{m}\cdot\frac{m-1}{m-n} && \text{[induction hypothesis]} \\
&< 1 + \frac{n}{m}\cdot\frac{m}{m-n} && [m-1 < m] \\
&= \frac{m}{m-n}\ \checkmark && \text{[algebra]}
\end{aligned}$$

Rewriting this in terms of the load factor $\alpha = n/m$, we get $\mathbf{E[}\boldsymbol{T(m,n)}\mathbf{] \le 1/(1-\alpha)}$. In other words, the expected time for an unsuccessful search is $O(1)$, unless the hash table is almost completely full.

5.9 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 $[r2^k .. (r+1)2^k - 1]$; 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.

*5.10 Analysis of Binary Probing

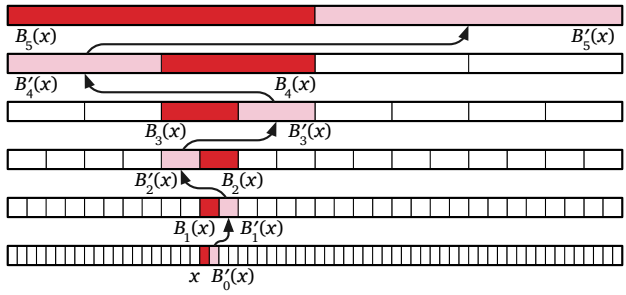
Lemma 8. *In a hash table of size $m = 2^\ell$ containing $n \le m/4$ keys, built using binary probing, the expected time for any search is $O(1)$, **assuming ideal random hashing**.*

Proof: The hash table is an array $H[0..m-1]$. 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 $H[c2^k .. (c+1)2^k - 1]$ 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 $H[h(x)]$:

$$B_k(x) = \left[2^k \lfloor h(x)/2^k \rfloor .. 2^k \lfloor h(x)/2^k \rfloor + 2^k - 1\right]$$

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.



A conservative view of binary probing.

The binary probing algorithm can be recast conservatively as follows. First the algorithm probes $H[h(x)]$; if that cell contains x or is empty, the algorithm halts. Then for each k from 0 to $\ell - 1$, the algorithm probes *every* cell in the uncle block $B'_k(x)$, and then halts if that block contained either x or an empty cell. The actual binary probing algorithm probes the cells in $B'_k(x)$ in a particular order and stops immediately when it finds either x or an empty cell, but for purposes of proving an upper bound, let's assume that the algorithm probes the entire block in some arbitrary order.

```
LooseBinaryProbe(x):
  if H[h(x)] = x
      return True
  if H[h(x)] is empty
      return False
  first ← Dunno

  for k ← 0 to ℓ − 1
      for each index j ∈ B'_k(x) in arbitrary order
          if first ≠ Dunno
              if H[j] = x
                  first ← True
              if H[j] is empty
                  first ← False

      if first ≠ Dunno
          return first

  return Full
```

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.) The expected running time of LooseBinaryProbe(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 $B'_k(x)$ (or any 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 $B'_k(x)$ (or any 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 $\sum_{k\ge 0} O(2^k) \cdot O(4^{-k}) = \sum_{k\ge 0} O(2^{-k}) = O(1)$. □

In fact, we can prove the same expected time bound with a much weaker randomness requirement.

Lemma 9. *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 5-uniform hashing**.*

Proof: Most of the previous proof carries through without modification; the only change is that we need a different argument to bound the probability that $B'_k(x)$ is popular.

For each element $y \ne x$, we define an indicator variable $P_y := [h(y) \in B'_k(x)]$. The uniformity of h implies that $\mathrm{E}[P_y] = \Pr[h(y) \in B'_k(x)] = 2^k/m$, to simplify notation, let $p = 2^k/m$. Now we define a second indicator variable

$$Q_y = P_y - p = \begin{cases} 1-p & \text{if } h(y) \in B'_k(x) \\ -p & \text{otherwise} \end{cases}$$

Linearity of expectation implies that $\mathrm{E}[Q_y] = 0$. Finally, define $P = \sum_{y\ne x} P_y$ and $Q = \sum_{y\ne x} Q_y = P - \mathrm{E}[P]$; again, linearity of expectation gives us $\mathrm{E}[P] = p(n-1) = 2^k(n-1)/m$. We can bound the probability that $B'_k(x)$ is popular in terms of these variables as follows:

$$\begin{aligned}
\Pr[B'_k(x) \text{ is popular}] &= \Pr[P \ge 2^k - 1] && \text{by definition of ``popular''} \\
&= \Pr[Q \ge 2^k - 1 - 2^k(n-1)/m] \\
&= \Pr[Q \ge 2^k(1 - n/m - 1/m) - 1] \\
&\le \Pr[Q \ge 2^k(3/4 - 1/m) - 1] && \text{because } n \le m/4 \\
&\le \Pr[Q \ge 2^{k-1}] && \text{because } m \ge 4n \ge 4.
\end{aligned}$$

Now we do something that looks a little weird; instead of considering the variable Q directly, we consider its fourth power. Because Q^4 is non-negative, Markov's inequality gives us

$$\Pr[Q \ge 2^{k-1}] \;=\; \Pr[Q^4 \ge 2^{4(k-1)}] \;\le\; \frac{E[Q^4]}{2^{4(k-1)}}$$

Linearity of expectation implies

$$\mathrm{E}[Q^4] = \sum_{y\ne x} \sum_{z\ne x} \sum_{y'\ne x} \sum_{z'\ne x} \mathrm{E}[Q_y Q_z Q_{y'} Q_{z'}].$$

Because h is 5-uniform, the random variables Q_y are 4-independent. (We lose one level of independence because Q_y depends on both y and the fixed element x.) It follows that if y, z, y', z' are all distinct, then $\mathrm{E}[Q_y Q_z Q_{y'} Q_{z'}] = \mathrm{E}[Q_y]\mathrm{E}[Q_z]\mathrm{E}[Q_{y'}]\mathrm{E}[Q_{z'}] = 0$. More generally, if any one of y, z, y', z' is different from the other three, then $\mathrm{E}[Q_y Q_z Q_{y'} Q_{z'}] = 0$. The expectation $\mathrm{E}[Q_y Q_z Q_{y'} Q_{z'}]$ is only non-zero when $y = z = y' = z'$, or when the values y, z, y', z' consist of two identical pairs.

$$\mathrm{E}[Q^4] = \sum_y \mathrm{E}[Q_y^4] + 6 \sum_{y<z} \mathrm{E}[Q_y^2]\mathrm{E}[Q_z^2]$$

The definition of expectation implies

$$\mathrm{E}[Q_y^2] \;=\; p(1-p)^2 + (1-p)(-p)^2 \;=\; p(1-p) \;<\; p$$

and similarly

$$\mathrm{E}[Q_y^4] \;=\; p(1-p)^4 + (1-p)(-p)^4 \;=\; p(1-p)((1-p)^3 + p^3) \;<\; p.$$

It follows that

$$\begin{aligned}\mathrm{E}[Q^4] &< (n-1)p + 6\binom{n-1}{2}p^2 \\ &< \frac{mp}{4} + 3\left(\frac{mp}{4}\right)^2 \\ &< 2^{k-2} + 3\cdot 2^{2(k-2)} \;<\; 2^{2(k-1)}\end{aligned}$$

Putting all the pieces together, we conclude that $\Pr[B'_k(x) \text{ is popular}] \le 2^{-2(k-1)}$. The rest of the proof is unchanged. □

5.11 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 $m \ge n$. (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,

$$\Pr_{h\in\mathcal{MB}^+}[h(x)=i \mid h(y)=j] = \Pr_{h\in\mathcal{MB}^+}[h(x)=i]$$

or equivalently,

$$\Pr_{h\in\mathcal{MB}^+}[(h(x)=i)\wedge(h(y)=j)] = \Pr_{h\in\mathcal{MB}^+}[h(x)=i]\cdot\Pr_{h\in\mathcal{MB}^+}[h(y)=j]$$

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 **??** is not uniform. *[Hint: What is $multb_a(0)$?]*

(b) Prove that $\mathcal{MB}$ is not pairwise independent. *[Hint: Compare $multb_a(0)$ and $multb_a(2^{w-1})$.]*

(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 $\mathrm{E}[X] = O(\log n)$.

6 Filtering and Streaming

The randomized algorithms and data structures we have seen so far *always* produce the correct answer but have a small probability of being slow. In this lecture, we will consider randomized algorithms that are always fast, but return the wrong answer with some small probability.[1] More generally, we are interested in tradeoffs between the (likely) efficiency of the algorithm and the (likely) quality of its output.

Specifically, we introduce an ***error rate*** δ and analyze the running time required to guarantee the output is correct with probability $1-\delta$. For "high probability" correctness, we need $\delta < 1/n^c$ for some constant c. In practice, it may be sufficient (or even necessary) to set δ to a small constant; for example, setting $\delta = 1/1000$ means the algorithm produces correct results at least 99.9% of the time.

6.1 Bloom Filters

Bloom filters are a natural variant of hashing proposed by Burton Bloom in 1970 as a mechanism for supporting membership queries in sets. In strict accordance with Stigler's Law of Autonomy, Bloom filters are identical to ***Zatocoding***, a coding system for library cards developed by Calvin Mooers in 1947. (Mooers was the person who coined the phrase "information retrieval".) A probabilistic analysis of Zatocoding appears in the personal notes of cybernetics pioneer W. Ross Ashby from 1960.

A Bloom filter (or Zatocode) for a set X of n items from some universe $\mathcal{U}$ allows one to test whether a given item $x \in \mathcal{U}$ is an element of X. Of course we can already do this with hash tables in $O(1)$ expected time, using $O(n)$ space. Bloom (and Mooers) observed that by allowing false positives—occasionally reporting $x \in X$ when in fact $x \notin X$—we can still answer queries in $O(1)$ expected time using considerably less space. False positives make Bloom filters unsuitable as an exact membership data structure, but because of their speed and low false positive rate, they are commonly used as filters or sanity checks for more complex data structures.

A Bloom filter consists of an array $B[0..m-1]$ of bits, together with k hash functions $h_1, h_2, \ldots, h_k : \mathcal{U} \to \{0, 1, \ldots, m-1\}$. For purposes of theoretical analysis, we assume the hash functions h_i are mutually independent, ideal random functions. This assumption is of course unsupportable in practice, but may be necessary to guarantee theoretical performance. Unlike many other types of hashing, nobody knows whether the same theoretical guarantees can be achieved using practical hash functions with more limited independence.[2] Fortunately, the actual

[1] Some textbooks (and Wikipedia) use the terms "Las Vegas" and "Monte Carlo" algorithms to respectively describe these two types of randomized algorithms. Most algorithms researchers don't.

[2] PhD thesis, anyone?

real-world behavior of Bloom filters appears to be consistent with this unrealistic theoretical analysis.

A Bloom filter for a set $X = \{x_1, x_2, \ldots, x_n\}$ is initialized by setting the bit $B[h_j(x_i)]$ to 1 for all indices i and j. Because of collisions, some bits may be set more than once, but that's fine.

```
MakeBloomFilter(X):
  for h ← 0 to m−1
      B[h] ← 0
  for i ← 1 to n
      for j ← 1 to k
          B[h_j(x_i)] ← 1
  return B
```

Given a new item y, the Bloom filter determines whether $y \in X$ by checking each bit $B[h_j(y)]$. If any of those bits is 0, the Bloom filter correctly reports that $y \notin X$ However, if all bits are 1, the Bloom filter reports that $y \in X$, although this is not necessarily correct.

```
BloomMembership(B, y):
  for j ← 1 to k
      if B[h_j(y)] = 0
          return False
  return Maybe
```

One nice feature of Bloom filters is that the various hash functions h_i can be evaluated in parallel on a multicore machine.

6.2 False Positive Rate

Let's estimate the probability of a false positive, as a function of the various parameters n, m, and k. For all indices h, i, and j, we have $\Pr[h_j(x_i) = h] = 1/m$, so ideal randomness gives us

$$Pr[B[h] = 0] \;=\; \left(1 - \frac{1}{m}\right)^{kn} \;\approx\; e^{-kn/m}$$

for every index h. Using this exact probability is rather unwieldy; to keep things sane, we will use the close approximation $\boldsymbol{p := e^{-kn/m}}$ instead.

The expected number of 0-bits in the Bloom filter is approximately mp; moreover, Chernoff bounds imply that the number of 0-bits is close to mp with very high probability. Thus, the probabililty of a false positive is very close[3] to

$$(1-p)^k = (1 - e^{-kn/m})^k.$$

If all other parameters are held constant, then the false positive rate increases with n (the number of items) and decreases with m (the number of bits). The dependence on k (the number of hash functions) is a bit more complicated, but we can derive the best value of k for given n and m as follows. Consider the logarithm of the false-positive rate:

$$\ln\left((1-p)^k\right) = k\ln(1-p) = -\frac{m}{n}\ln p \ln(1-p).$$

By symmetry, this expression is minimized when $\boldsymbol{p = 1/2}$. We conclude that, the optimal number of hash functions is $\boldsymbol{k = \ln 2 \cdot (m/n)}$, which would give us the false positive rate

[3] This analysis, originally due to Bloom, assumes that certain events are independent even though they are not; as a result, the estimate given here is slightly below the true false positive rate.

$(1/2)^{\ln 2(m/n)} \approx (0.61850)^{m/n}$. Of course, in practice, k must be an integer, so we cannot achieve precisely this rate, but we can get reasonably close (at least if $m \gg n$).

Finally, the previous analysis implies that we can achieve any desired false positive rate $\delta > 0$ using a Bloom filter of size

$$m = \left\lceil \frac{\lg(1/\delta)}{\ln 2} n \right\rceil = \Theta(n \log(1/\delta))$$

that uses with $k = \lceil \lg(1/\delta) \rceil$ hash functions. For example, we can achieve a 1% false-positive rate using a Bloom filter of size $10n$ bits with 7 hash functions; in practice, this is *considerably* fewer bits than we would need to store all the elements of S explicitly. With a $32n$-bit table (equivalent to one integer per item) and 22 hash functions, we get a false positive rate of just over $2 \cdot 10^{-7}$.

Deletions via counting filters? Key-value pairs via Bloomier filters? Other extensions?

6.3 Streaming Algorithms

A ***data stream*** is an extremely long sequence S of items from some universe $\mathcal{U}$ that can be read only once, in order. Good examples of data streams include the sequence of packets that pass through a network router, the sequence of searches at google.com, the sequence of all bids on the New York Stock Exchange, and the sequence of humans passing through the Shinjuku Railway Station in Tokyo. Standard algorithms are not appropriate for data streams; there is simply too much data to store, and it arrives too quickly for any complex computations.

A ***streaming algorithm*** processes each item in a data stream stream as it arrives, maintaining some summary information in a local data structure. The basic structure of every streaming algorithm is the following:

```
DoSomething(S):
  ⟨⟨initialize⟩⟩
  while S is not done
      x ← next item in S
      ⟨⟨do something fast with x⟩⟩
  return ⟨⟨something⟩⟩
```

Ideally, neither the running time per item nor the space used by the data structure depends on the overall length of the stream; viewed as algorithms in the traditional sense, all streaming algorithms run in constant time! Somewhat surprisingly, even within this very restricted model, it is possible to compute interesting properties of the stream using randomization, provided we are willing to tolerate some errors in the output.

6.4 The Count-Min Sketch

As an example, consider the following problem: At any point during the stream, we want to estimate the number of times that an arbitrary item $x \in \mathcal{U}$ has appeared in the stream so far. This problem can be solved with a variant of Bloom filters called the ***count-min sketch***, first published by Graham Cormode and S. Muthu Muthukrishnan in 2005.

The count-min sketch consists of a $w \times d$ array of counters (all initially zero) and d hash functions $h_1, h_2, \ldots, h_d : \mathcal{U} \to [m]$, drawn independently and uniformly at random from a 2-uniform family of hash functions. Each time an item x arrives in the stream, we call CMIncrement(x). Whenever we want to estimate the number of occurrences of an item x so far, we call CMEstimate(x).

```
CMIncrement(x):
  for i ← 1 to d
      j ← h_i(x)
      Count[i, j] ← Count[i, j] + 1
```

```
CMEstimate(x):
  est ← ∞
  for i ← 1 to d
      j ← h_i(x)
      est ← min{est, Count[i, j]}
  return est
```

If we set $w := \lceil e/\varepsilon \rceil$ and $d := \lceil \ln(1/\delta) \rceil$, then the data structure uses $O(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ space and processes updates and queries in $O(\log \frac{1}{\delta})$ time.

Let f_x be the true frequency (number of occurrences) of x, and let $\hat{f}_x$ be the value returned by CMEstimate. It is easy to see that $f_x \le \hat{f}_x$; we can never return a value smaller than the actual number of occurrences of x. We claim that $\mathbf{Pr}[\hat{f}_x > f_x + \varepsilon N] < \delta$, where N is the total length of the stream. In other words, our estimate is never too small, and with high probability, it isn't a significant overestimate either. (Notice that the error here is additive; the estimates or truly infrequent items may be much larger than their true frequencies.)

For any items $x \neq y$ and any index j, we define an indicator variable $\boldsymbol{X_{i,x,y} = [h_i(x) = h_i(y)]}$; because the hash functions h_i are universal, we have

$$\mathrm{E}[X_{i,x,y}] = \Pr[h_i(x) = h_i(y)] = \frac{1}{w}.$$

Let $\boldsymbol{X_{i,x} := \sum_{y \neq x} X_{i,x,y} \cdot f_y}$ denote the total number of collisions with x in row i of the table. Then we immediately have

$$Count[i, h_i(x)] \;=\; f_x + X_{i,x} \;\ge\; f_x.$$

On the other hand, linearity of expectation implies

$$\mathrm{E}[X_{i,x}] \;=\; \sum_{y \neq x} \mathrm{E}[X_{i,x,y}] \cdot f_y \;=\; \frac{1}{w} \sum_{y \neq x} f_y \;\le\; \frac{N}{w}.$$

Now Markov's inequality implies

$$\begin{aligned}
\Pr[\hat{f}_x > f_x + \varepsilon N] &= \Pr[X_{i,x} > \varepsilon N \text{ for all i}] && \text{[definition]} \\
&= \Pr[X_{1,x} > \varepsilon N]^d && \text{[independence of } h_i\text{'s]} \\
&\le \left(\frac{\mathrm{E}[X_{1,x}]}{\varepsilon N} \right)^d && \text{[Markov's inequality]} \\
&\le \left(\frac{N/w}{\varepsilon N} \right)^d = \left(\frac{1}{w\varepsilon} \right)^d && \text{[derived earlier]}
\end{aligned}$$

Now setting $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$ gives us $\Pr[\hat{a}_x > a_x + \varepsilon N] \le (1/e)^{\ln(1/\delta)} = \delta$, as claimed.

6.5 Estimating Distinct Items

Write this, possibly as a simpler replacement for the count-min sketch. AMS estimator: $2^{z+1/2}$ where $z = \max\{zeros(x) \mid x \in S\}$? Or stick with the Flajolet-Martin/Bar-Yossef-et-al estimator in the exercises? Median amplification.

Estimating larger moments?

Exercises

1. ***Reservoir sampling*** is a method for choosing an item uniformly at random from an arbitrarily long stream of data; for example, the sequence of packets that pass through a router, or the sequence of IP addresses that access a given web page. Like all data stream algorithms, this algorithm must process each item in the stream quickly, using very little memory.

```
GetOneSample(stream S):
  ℓ ← 0
  while S is not done
      x ← next item in S
      ℓ ← ℓ + 1
      if Random(ℓ) = 1
          sample ← x        (⋆)
  return sample
```

   At the end of the algorithm, the variable ℓ stores the length of the input stream S; this number is *not* known to the algorithm in advance. If S is empty, the output of the algorithm is (correctly!) undefined. In the following, consider an arbitrary non-empty input stream S, and let n denote the (unknown) length of S.

   (a) Prove that the item returned by GetOneSample(S) is chosen uniformly at random from S.

   (b) What is the *exact* expected number of times that GetOneSample(S) executes line ($\star$)?

   (c) What is the *exact* expected value of ℓ when GetOneSample(S) executes line ($\star$) for the *last* time?

   (d) What is the *exact* expected value of ℓ when either GetOneSample(S) executes line ($\star$) for the *second* time (or the algorithm ends, whichever happens first)?

   (e) Describe and analyze an algorithm that returns a subset of k distinct items chosen uniformly at random from a data stream of length at least k. The integer k is given as part of the input to your algorithm. Prove that your algorithm is correct.

   For example, if $k = 2$ and the stream contains the sequence $\langle ♠, ♥, ♦, ♣ \rangle$, the algorithm should return the subset $\{♦, ♠\}$ with probability $1/6$.

2. In this problem, we will derive a simple streaming algorithm (first published by Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan in 2002) to estimate the number of distinct items in a data stream S.

   Suppose S contains n unique items (but possibly several copies of each item); as usual, the algorithm does *not* know n in advance. Given an accuracy parameter $0 < \varepsilon < 1$ and a confidence parameter $0 < \delta < 1$ as part of the input, our final algorithm will compute an estimate $\widetilde{N}$ such that $\Pr[|\widetilde{N} - n| > \varepsilon n] < \delta$.

   As a first step, fix a positive integer m that is large enough that we don't have to worry about round-off errors in the analysis. Our first algorithm chooses a hash function $h : \mathcal{U} \to [m]$ at random from a ***2-uniform*** family, computes the minimum hash value $\hbar = \min\{h(x) \mid x \in S\}$, and finally returns the estimate $\widetilde{n} = m/\hbar$.

(a) Prove that $\Pr[\widetilde{n} > (1+\varepsilon)n] \le 1/(1+\varepsilon)$. *[Hint: Markov's inequality]*

(b) Prove that $\Pr[\widetilde{n} < (1-\varepsilon)n] \le 1-\varepsilon$. *[Hint: Chebyshev's inequality]*

(c) We can improve this estimator by maintaining the k smallest hash values, for some integer $k > 1$. Let $\widetilde{n}_k = k \cdot m/\hbar_k$, where $\hbar_k$ is the kth smallest element of $\{h(x) \mid x \in S\}$.

Estimate the smallest value of k (as a function of the accuracy parameter ε) such that $\Pr[|\widetilde{n}_k - n| > \varepsilon n] \le 1/4$.

(d) Now suppose we run d copies of the previous estimator in parallel to generate d independent estimates $\widetilde{n}_{k,1}, \widetilde{n}_{k,2}, \ldots, \widetilde{n}_{k,d}$, for some integer $d > 1$. Each copy uses its own independently chosen hash function, but they all use the same value of k that you derived in part (c). Let $\widetilde{N}$ be the *median* of these d estimates.

Estimate the smallest value of d (as a function of the confidence parameter δ) such that $\Pr[|\widetilde{N} - n| > \varepsilon n] \le \delta$.

7 String Matching

7.1 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.

[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!

In most cases the pattern is much smaller than the text; to make this concrete, I'll assume that $m < n/2$.

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.

7.2 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 $T =$ `31415926535897932384626433832795028841971` 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!

7.3 Karp-Rabin Fingerprinting

To make this algorithm efficient, we will make one simple change, proposed by Richard Karp and Michael Rabin in 1981:

> 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:

$$p \bmod q = P[m] + \Big(\cdots + \big(10 \cdot \big(P[2] + \big(10 \cdot P[1] \bmod q\big) \bmod q\big) \bmod q\big) \cdots\Big) \bmod q.$$

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
  t̃₁ ← 0
  for i ← 1 to m
      p̃ ← (10 · p̃ mod q) + P[i] mod q
      t̃₁ ← (10 · t̃₁ 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.

```
CarterWegmanKarpRabin(T[1..n], P[1..m]):
  q ← an arbitrary prime number larger than m²
  b ← Random(q) − 1        ⟨⟨uniform between 0 and q − 1⟩⟩
  σ ← b^(m−1) mod q
  p̃ ← 0
  t̃_1 ← 0
  for i ← 1 to m
      p̃ ← (b · p̃ mod q) + P[i] mod q
      t̃_1 ← (b · 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} ← (b · (t̃_s − (σ · T[s] mod q) mod q) mod q) + T[s + m] mod 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***.

7.4 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
          ✗|BRACADABRA
           ✗|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 $T[15] = P[4] = \texttt{A}$. 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.**

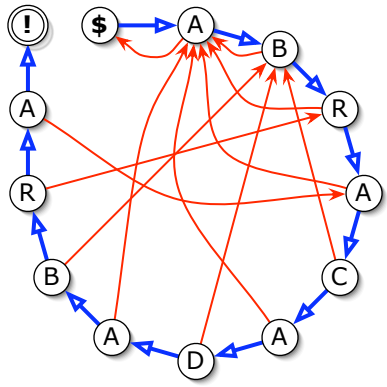
in 1977, Donald Knuth, James Morris, and Vaughn Pratt published a string-matching algorithm that implements both of these ideas.

7.5 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 (Ⓢ).)



A finite state machine for the string ABRACADABRA.
Thick arrows are the success edges; thin arrows are the failure edges.

- 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.

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 $O(n)$.

7.6 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:

> $P[1..fail[j]-1]$ is the longest proper prefix of $P[1..j-1]$ that is also a suffix of $P[1..j-1]$.

This is the definition of the Knuth-Morris-Pratt failure function $fail[j]$ for all $j > 1$. By convention we set $fail[1] = 0$; 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
$fail[i]$	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 $fail[1] = 0$ is obvious. Assuming inductively that we correctly computed $fail[1]$ through $fail[i-1]$ in line $(*)$, we need to show that $fail[i]$ is also correct. Just after the ith iteration of line $(*)$, we have $j = fail[i]$, 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 $fail^c[j]$ inductively as follows: $fail^0[j] = j$, and

$$fail^c[j] = fail[fail^{c-1}[j]] = \overbrace{fail[fail[\cdots[fail}^{c}[j]]\cdots]].$$

$j \leftarrow 0, i \leftarrow 1$	$\$^j$	$\mathbf{A}^i$	B	R	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		**0**											...
$j \leftarrow j+1, i \leftarrow i+1$	$	$\mathbf{A}^j$	$\mathbf{B}^i$	R	A	C	A	D	A	B	R	X	...
$fail[i] \leftarrow j$		0	**1**										...
$j \leftarrow fail[j]$	$\$^j$	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	...
$fail[i] \leftarrow j$		0	1	**1**									...
$j \leftarrow fail[j]$	$\$^j$	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	...
$fail[i] \leftarrow j$		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	...
$fail[i] \leftarrow j$		0	1	1	1	**2**							...
$j \leftarrow fail[j]$	$	$\mathbf{A}^j$	B	R	A	$\mathbf{C}^i$	A	D	A	B	R	X	...
$j \leftarrow fail[j]$	$\$^j$	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	...
$fail[i] \leftarrow j$		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	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	**2**					...
$j \leftarrow fail[j]$	$	$\mathbf{A}^j$	B	R	A	C	A	$\mathbf{D}^i$	A	B	R	X	...
$j \leftarrow fail[j]$	$\$^j$	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	...
$fail[i] \leftarrow j$		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	...
$fail[i] \leftarrow j$		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	...
$fail[i] \leftarrow j$		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	$\mathbf{X}^i$	...
$fail[i] \leftarrow j$		0	1	1	1	2	1	2	1	2	3	**4**	...
$j \leftarrow fail[j]$	$	$\mathbf{A}^j$	B	R	A	C	A	D	A	B	R	$\mathbf{X}^i$	...
$j \leftarrow fail[j]$	$\$^j$	A	B	R	A	C	A	D	A	B	R	$\mathbf{X}^i$	...

ComputeFailure in action. Do this yourself by hand!

In particular, if $fail^{c-1}[j] = 0$, then $fail^c[j]$ is undefined. We can easily show by induction that every string of the form $P[1..fail^c[j]-1]$ 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 $P[1..fail^c[j]]$—the one with smallest c—such that $P[fail^c[j]] = P[i]$. This is exactly what the while loop in ComputeFailure computes; 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 $fail[i]$:

$$fail[i] = \begin{cases} 0 & \text{if } i = 0, \\ \max_{c \geq 1} \left\{ fail^c[i-1] + 1 \;\middle|\; P[i-1] = P[fail^c[i-1]] \right\} & \text{otherwise.} \end{cases}$$

7.7 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]]
```

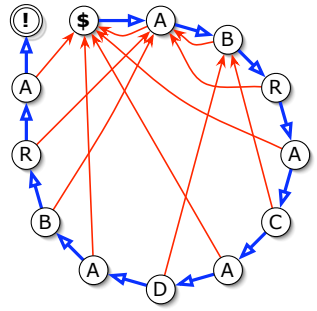
We can also compute the optimized failure function directly by adding three new lines (in bold) to the ComputeFailure function.

```
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.

Feb 2017: Manacher's palindrome-substring algorithm and the Aho-Corasick dictionary-matching algorithm use similar ideas. Knuth-Morris-Pratt was published in 1977, two years after Manacher and Aho-Corasick, but was available as a Stanford tech report in 1974. See also: Gusfield's algorithm Z (which is essentially just KMP preprocessing on $P \bullet T$).



$P[i]$	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.

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 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 `SCHABRAINCADBRANCH`; 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 `SCHABRUCADIBRANCH`. 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* $j \to fail[j] \to fail[fail[j]] \to fail[fail[fail[j]]] \to \cdots \to 0$, 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.

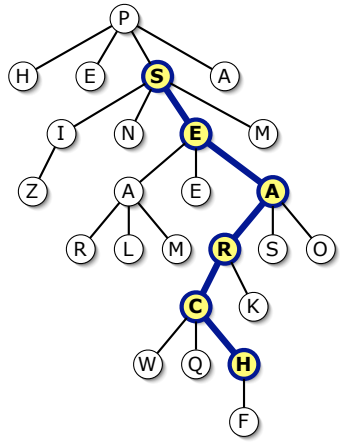
(a) Describe a pattern $A[1..m]$ whose longest *unoptimized* failure chain has length m.

(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.

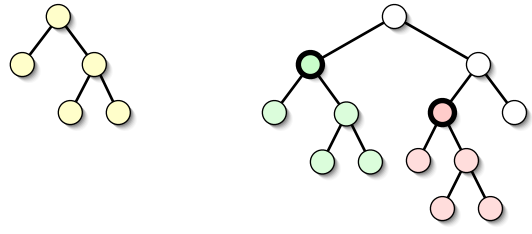


The string SEARCH appears on a downward path in the tree.

(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.

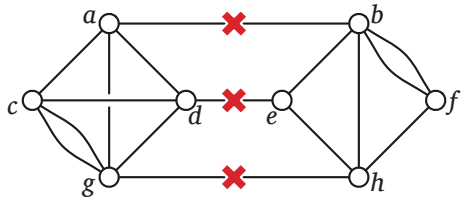
(b) Describe and analyze a variant of KnuthMorrisPratt that solves this problem in $O(m+n)$ expected time.

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.



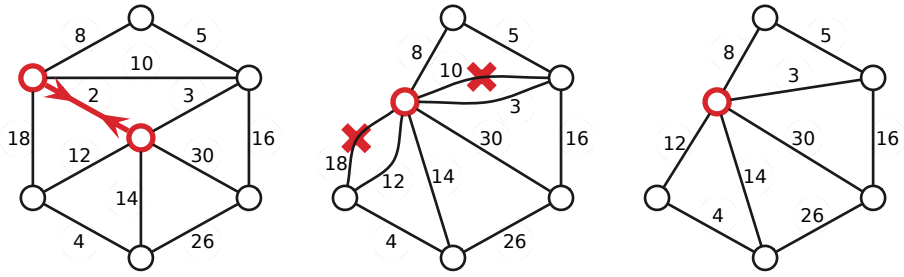
A multigraph whose minimum cut has three edges.

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 ***edge contraction***. Suppose u and v are vertices that are connected by an edge in some multigraph G. To contract 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, contracting the edge shrinks the edge down to nothing, pulling the two endpoints together. The new contracted graph is denoted $G/\{u, v\}$. We don't allow self-loops in our multigraphs; if there are multiple edges between u and v, contracting any one of them deletes them all.



A graph G and two contracted graphs $G/\{b, e\}$ and $G/\{c, d\}$.

Any edge in an n-vertex graph can be contracted 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 contracted graph $G/\{c, d\}$ does not.

This simple observation has two immediate but important consequences. First, contracting an edge cannot decrease the minimum cut size. More importantly, contracting 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 contracting edges until only two vertices remain.

```
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 contraction requires $O(n)$ time, this algorithm runs in $O(n^2)$ time. Our earlier observations imply that as long as we never contract 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 contracted 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 with high probability.

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 exploiting the observation we made earlier:

> *As the graph shrinks, the probability of contracting 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 was to group the first several random contractions a "safe" phase, so that the cumulative probability of screwing up is relatively small and a "dangerous" phase, which is much more likely to screw up.

The safe phase shrinks the graph from n nodes to about $n/2$ nodes, using a sequence of about $n/2$ random contractions.[4] Following our earlier analysis, the probability that *none* of these safe contractions touches the minimum cut is at least

$$\prod_{i=n/2+1}^{n} \frac{i-2}{i} = \frac{(n/2)(n/2-1)}{n(n-1)} = \frac{n-2}{4(n-1)} \approx \frac{1}{4}.$$

To get around the danger of the dangerous phase, we use amplification: we run the dangerous phase *four times* and keep the best of the four answers. Naturally, we treat the dangerous phase recursively, so we actually obtain a recursion tree with degree 4, 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 n < 1000
      use brute force
  else
      H ← CONTRACT(G, n/α)
      X1 ← BETTERGUESS(H, α)
      X2 ← BETTERGUESS(H, α)
      X3 ← BETTERGUESS(H, α)
      X4 ← BETTERGUESS(H, α)
      return min{X1, X2, X3, X4}
```

At first glance, it may look like we are performing exactly the same BETTERGUESS computation four times in a row, but remember that CONTRACT (and therefore BETTERGUESS) is randomized. Each recursive call to BETTERGUESS contracts a different, independently chosen random set of edges. Thus, X_1, X_2, X_3, and X_4 are almost always four completely different cuts!

Why four recursive calls, and not two or six? We're facing a tradeoff between the speed of the algorithm and its probability of success. More recursive calls makes the algorithm more likely to succeed but slower; fewer recursive calls makes the algorithm faster but *considerably* more likely to fail. Four recursive calls turns out to be the right balance, both guaranteeing a fast algorithm and a reasonably large success probability.

[3] David R. Karger* and Cliff Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. Proc. 25th STOC, 757–765, 1993.

[4] I'm deliberately simplifying the analysis here to expose more intuition. More formally, if we contract from n to $\lceil n/2 \rceil + 1$ nodes, the probability that no minimum cut edge is contracted is strictly greater than $1/2$.

BETTERGUESS correctly returns the minimum cut unless (1) none of the edges of the minimum cut are CONTRACTed and (2) *all four* recursive calls return the incorrect cut for H. Let $P(n)$ denote the probability that BETTERGUESS returns a minimum cut of an n-node graph. Then for each index i, the probability that X_i is the minimum cut of H is $P(n/2)$. Because the four recursive calls are independent, we obtain the following recurrence for $P(n)$, with base case $P(n) = 1$ for all $n \le 8$.

$$\begin{aligned}
P(n) &= \Pr[\text{mincut in } H \text{ is mincut in } G] \cdot \Pr[\text{some } X_i \text{ is mincut in } H] \\
&\ge \frac{1}{4}\left(1 - \Pr[\text{no } X_i \text{ is mincut in } H]\right) \\
&\ge \frac{1}{4}\left(1 - \Pr[X_1 \text{ is not mincut in } H]^4\right) \\
&\ge \frac{1}{4}\left(1 - \left(1 - \Pr[X_1 \text{ is mincut in } H]\right)^4\right) \\
&\ge \frac{1}{4}\left(1 - \left(1 - P\left(\frac{n}{2}\right)\right)^4\right) \\
&= \frac{1}{4} - \frac{1}{4}\left(1 - P\left(\frac{n}{2}\right)\right)^4
\end{aligned}$$

Using a series of transformations, Karger and Stein prove that $P(n) = \Omega(1/\log n)$. I've included a proof at the end of this note.

For the running time, we get a simple recurrence that is easily solved using recursion trees:

$$T(n) = O(n^2) + 4T\left(\frac{n}{2}\right) = O(n^2 \log n)$$

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 $P(n) \ge \alpha/\ln n$ 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

$$1 - \left(1 - \frac{\alpha}{\ln n}\right)^{c\ln^2 n} \ge 1 - e^{-(c/\alpha)\ln n} = 1 - \frac{1}{n^{c/\alpha}}.$$

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 recursive inequality for the probability that BETTERGUESS successfully finds a minimum cut of an n-node graph:

$$P(n) \ge \frac{1}{4} - \frac{1}{4}\left(1 - P\left(\frac{n}{2}\right)\right)^4$$

Let $\overline{P}(n)$ be the function that satisfies this recurrence with equality; clearly, $P(n) \ge \overline{P}(n)$.

We can solve this rather ugly recurrence for $\overline{P}(n)$ through a series of functional transformations. First, consider the function $p(k) = \overline{P}(2^k)$; this function satisfies the recurrence

$$p(k) = \frac{1}{4} - \frac{1}{4}(1 - p(k-1))^4,$$

Next, define $d(k) = 1/p(k)$; this new function obeys the reciprocal recurrence

$$d(k) = \frac{4}{1-\left(1-\frac{1}{d(k-1)}\right)^4} = \frac{4\,d(k-1)^4}{d(k-1)^4-(d(k-1)-1)^4}$$

Straightforward algebraic manipulation[5] implies the inequalities

$$x < \frac{4x^4}{x^4-(x-1)^4} \le x+3$$

for all $x \ge 1$. Thus, as long as $d(k-1) \ge 1$, we have the simple recursive inequalities

$$d(k-1) < d(k) \le d(k-1)+3.$$

Our original base case $P(2) = 1$ implies the base case $d(1) = 1$. It immediately follows by induction that $d(k) \le 3k-2$, and therefore

$$P(n) \ge \overline{P}(n) = p(\lg n) = \frac{1}{d(\lg n)} \ge \frac{1}{3\lg n - 2} = \Omega\left(\frac{1}{\log n}\right),$$

as promised. Whew!

Galton-Watson process?

Exercises

1. (a) 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.[6] Use this algorithm as a subroutine to improve the running time of GuessMinCut from $O(n^2)$ to $O(m)$.

   (b) Describe and analyze an algorithm to compute the *maximum-weight edge* in the minimum spanning tree of a graph with m edges in $O(m)$ time. *[Hint: We can compute the median edge weight in $O(m)$ time. How do you tell quickly if that's too big or too small?]*

   (c) Use your algorithm from part (b) to improve the running time of GuessMinCut from $O(n^2)$ to $O(m)$.

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 and analyze 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 $\Omega(1/n^2)$.

[5] otherwise known as Wolfram Alpha

[6] 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 2015 runs in $O(m\alpha(m))$ time, where $\alpha(\cdot)$ is the inverse Ackermann function.

(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):
  if n < 1000
      use brute force
  else
      H ← CONTRACT(G, n/α)
      X1 ← BETTERGUESS(H, α)
      X2 ← BETTERGUESS(H, α)
      X3 ← BETTERGUESS(H, α)
      X4 ← BETTERGUESS(H, α)
      return min{X1, X2, X3, X4}
```

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 $\alpha < 2$, $\alpha = 2$, and $\alpha > 2$ 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 $\alpha = 2$, the probability is approximately $1/4$.]*

(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 $\alpha < 2$, $\alpha = 2$, and $\alpha > 2$ 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 make k recursive calls instead of four. What is the best choice of α, as a function of k? What is the resulting running time? What is the best choice for k and α?

9 Amortized Analysis

9.1 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.

9.2 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.

9.2.1 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 $T(n)/n$.*

9.2.2 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.

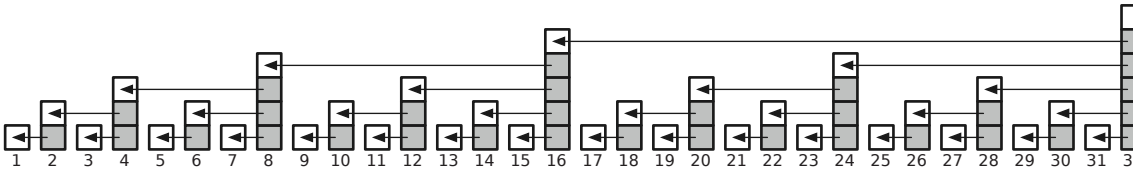
9.2.3 Charging

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.

9.2.4 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.

$$c_i = \#\text{bits changed from 0 to 1} + \#\text{bits changed from 1 to 0}$$
$$\Phi_i - \Phi_{i-1} = \#\text{bits changed from 0 to 1} - \#\text{bits changed from 1 to 0}$$

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).

9.3 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
```

$$\begin{array}{ccccccccccc}
P = 10001 & & P = 10010 & & P = 10011 & & P = 10000 & & P = 10000 & & P = 10000 & & P = 10001 \\
N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 01000 & \xrightarrow{--} & N = 01001 & \xrightarrow{--} & N = 01010 & \xrightarrow{++} & N = 01010 \\
P - N = 5 & & P - N = 6 & & P - N = 7 & & P - N = 8 & & P - N = 7 & & P - N = 6 & & P - N = 7
\end{array}$$

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 $a_i \leq 2$.

*9.4 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, 1
F : 3210, 3211, 3220, 3212, 3310, 3311, 3230, 3213, 4210, 4211, 4220, 4212, 3410, 3411, 3240, 3

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?

9.5 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. $f(k) = size(k) \lg size(k)$, where $size(k)$ 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 $O(1)$.

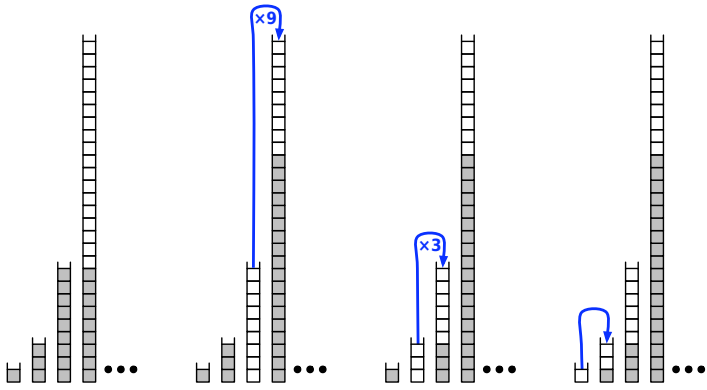
5. A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. The user always pushes and pops elements from the smallest stack S_0. However, before any element can be pushed onto any full stack S_i, we first pop all the elements 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 ± 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±01 (because $2\cdot 2^3 - 2^2 + 2^0 = 13$) or 10±1± (because $2^4 - 2^2 + 2^1 - 2^0 = 13$) or ±1200010±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 ± (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.

$$
\begin{aligned}
&0 \xrightarrow{++} 1 \xrightarrow{++} 10 \xrightarrow{++} 11 \xrightarrow{++} 20 \xrightarrow{++} 101 \xrightarrow{++} 110 \xrightarrow{++} 111 \xrightarrow{++} 120 \xrightarrow{++} 201 \xrightarrow{++} 210 \\
&\xrightarrow{++} 1011 \xrightarrow{++} 1020 \xrightarrow{++} 1101 \xrightarrow{++} 1110 \xrightarrow{++} 1111 \xrightarrow{++} 1120 \xrightarrow{++} 1201 \xrightarrow{++} 1210 \xrightarrow{++} 2011 \xrightarrow{++} 2020 \\
&\xrightarrow{--} 2011 \xrightarrow{--} 2010 \xrightarrow{--} 2001 \xrightarrow{--} 2000 \xrightarrow{--} 20±1 \xrightarrow{--} 2±10 \xrightarrow{--} 2±01 \xrightarrow{--} 1100 \xrightarrow{--} 11±1 \xrightarrow{--} 1010 \\
&\xrightarrow{--} 1001 \xrightarrow{--} 1000 \xrightarrow{--} 10±1 \xrightarrow{--} 1±10 \xrightarrow{--} 1±01 \xrightarrow{--} 100 \xrightarrow{--} 1±1 \xrightarrow{--} 10 \xrightarrow{--} 1 \xrightarrow{--} 0
\end{aligned}
$$

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.

10 Scapegoat and Splay Trees

10.1 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].

10.2 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 $\boldsymbol{O(\log n)}$.

10.3 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

$$size(right(v)) < \big(1 - 1/2^{1/\alpha}\big) size(v) - 1.$$

Finally, we combine these two inequalities using the recursive definition of imbalance.

$$Imbal(v) \geq size(left(v)) - size(right(v)) - 1 > \big(2/2^{1/\alpha} - 1\big) size(v)$$

Since α is a constant bigger than 1, the factor in parentheses is a positive constant. □

10.4 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.

10.5 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.

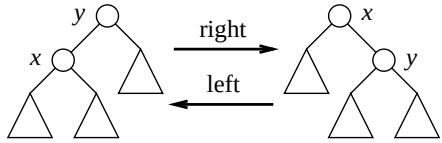


Figure 1. A right rotation at x and a left rotation at y are inverses.

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!)

10.6 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:

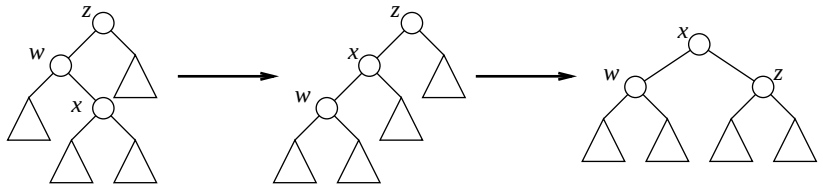


Figure 2. A zig-zag at x. The symmetric case is not shown.

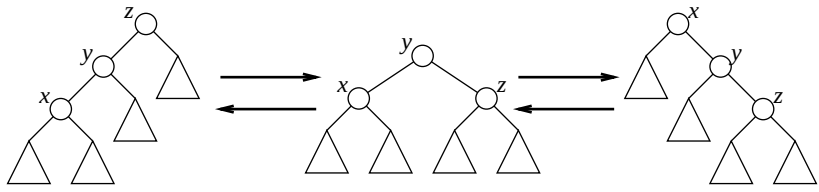


Figure 3. A right roller-coaster at x and a left roller-coaster at z.

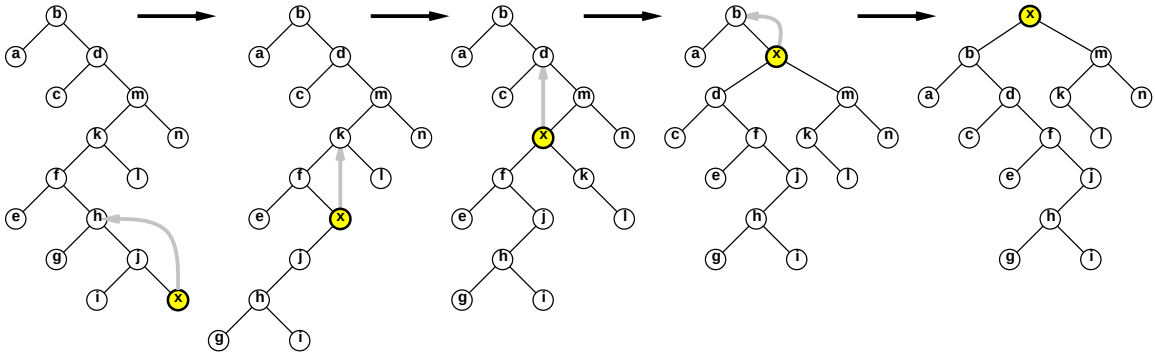


Figure 4. Splaying a node. Irrelevant subtrees are omitted for clarity.

- **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.

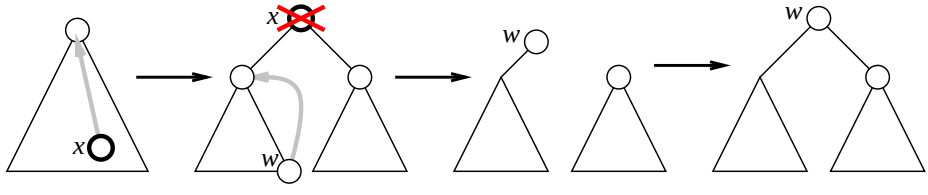


Figure 5. Deleting a node in a splay tree.

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.

*10.7 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.

$$\begin{aligned}
&2 + \Phi' - \Phi \\
&\quad = 2 + r'(w) + r'(x) + r'(z) - r(w) - r(x) - r(z) && \text{[only } w, x, z \text{ change rank} \\
&\quad \le 2 + r'(w) + r'(x) + r'(z) - 2r(x) && [r(x) \le r(w) \text{ and } r'(x) = r(z) \\
&\quad = 2 + (r'(w) - r'(x)) + (r'(z) - r'(x)) + 2(r'(x) - r(x)) \\
&\quad = 2 + \lg \frac{s'(w)}{s'(x)} + \lg \frac{s'(z)}{s'(x)} + 2(r'(x) - r(x)) \\
&\quad \le 2 + 2\lg \frac{s'(x)/2}{s'(x)} + 2(r'(x) - r(x)) && [s'(w) + s'(z) \le s'(x), \lg \text{ is concave} \\
&\quad = 2(r'(x) - r(x)) \\
&\quad \le 3(r'(x) - r(x)) && [r'(x) \ge r(x)
\end{aligned}$$

Finally, consider a roller-coaster, as shown in Figure 3. Only x, y, and z change rank.

$$\begin{aligned}
&2 + \Phi' - \Phi \\
&\quad = 2 + r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z) && \text{[only } x, y, z \text{ change rank]} \\
&\quad \le 2 + r'(x) + r'(z) - 2r(x) && [r'(y) \le r(z) \text{ and } r(x) \ge r(y)] \\
&\quad = 2 + (r(x) - r'(x)) + (r'(z) - r'(x)) + 3(r'(x) - r(x)) \\
&\quad = 2 + \lg \frac{s(x)}{s'(x)} + \lg \frac{s'(z)}{s'(x)} + 3(r'(x) - r(x)) \\
&\quad \le 2 + 2\lg \frac{s'(x)/2}{s'(x)} + 3(r'(x) - r(x)) && [s(x) + s'(z) \le s'(x), \lg \text{ is concave}] \\
&\quad = 3(r'(x) - r(x))
\end{aligned}$$

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 $key(x) \le key(y) \le key(z)$ or $key(x) \ge key(y) \ge key(z)$. 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 $s(root) \le \sum_{i=1}^{\infty} 2/i^2 = \pi^2/3 = O(1)$, and $r(x) \ge \lg w(x) = -2\lg dist(f,x)$. □

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 $O(\lg dist(x,y))$.*

*10.8 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 Bădiou* 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 Pătraşcu. 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 Pătraşcu**. 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 $n \leq 1$, or its two subtrees are perfectly balanced binary trees, each with at most $\lfloor n/2 \rfloor$ nodes. Prove that $I(v) \leq 1$ 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.

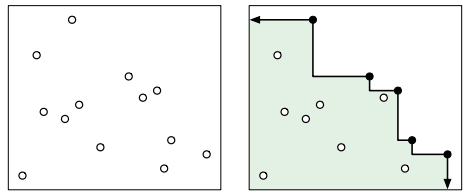
   ★(c) Modify your algorithm so that is uses only $O(1)$ additional space. In particular, your algorithm cannot call itself recursively at all.

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.

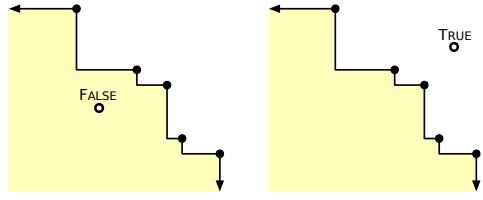
4. Let P be a set of n points in the plane. The *staircase* of P is the set of all points in the plane that have at least one point in P both above and to the right.



A set of points in the plane and its staircase (shaded).

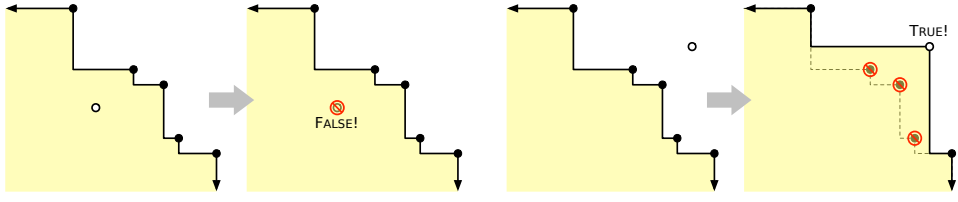
(a) Describe an algorithm to compute the staircase of a set of n points in $O(n \log n)$ time.

(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:

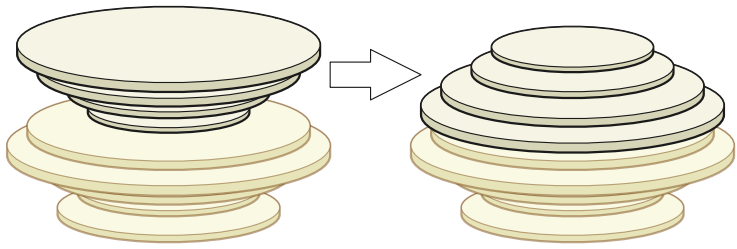
   - NEWSTRING(a) creates a new string of length 1 containing only the character a and returns a pointer to that string.
   - CONCAT(S, T) removes the strings S and T (given by pointers) from the data structure, adds the concatenated string ST to the data structure, and returns a pointer to the new string.
   - 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, $\underline{1}\,\overline{4}\,3\,\underline{\overline{2}}$ 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:

- POSITION(x): Return the position of integer x in the current sequence, or 0 if x is not in the sequence.
- VALUE(k): Return the kth integer in the current sequence, or 0 if the sequence has no kth element. VALUE is essentially the inverse of POSITION.
- TOPBURNED(k): Return TRUE if and only if the top side of the kth pancake in the current sequence is burned.
- FLIP(k): Reverse the order and the burn marks of the first k elements of the sequence.
- 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.

$$\underline{1}\,\overline{4}\,3\,\underline{\overline{2}} \xrightarrow{\text{FLIP}(4)} \underline{\overline{2}}\,3\,\underline{4}\,\overline{1} \xrightarrow{\text{DISCARD}(1)} 3\,\underline{4}\,\overline{1} \xrightarrow{\text{FLIP}(2)} \overline{4}\,3\,\overline{1} \xrightarrow{\text{FLIP}(3)} \underline{1}\,3\,\underline{4}$$

(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.

11 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.

11.1 Reversed Trees

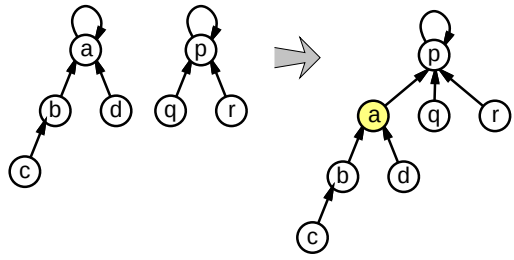
One of the easiest ways to store sets is using trees, in which each node represents a single element of the set. Each node points to another node, called its *parent*, except for the leader of each set, which points to itself and thus is the root of the tree. MAKESET is trivial. FIND traverses

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
```

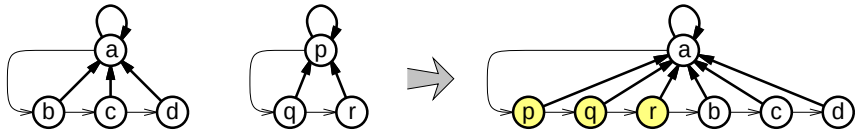
With this new rule in place, it's not hard to prove by induction that for any set leader $\overline{x}$, the size of $\overline{x}$'s set is at least $2^{depth(\overline{x})}$, as follows. If $depth(\overline{x}) = 0$, then $\overline{x}$ is the leader of a singleton set. For any $d > 0$, when $depth(\overline{x})$ 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.

11.2 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.

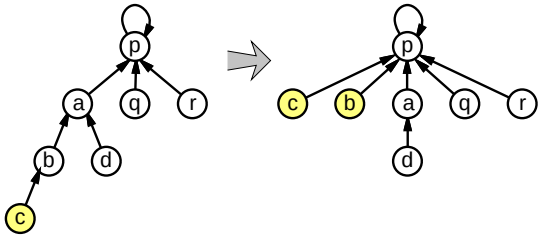
Since each WEIGHTEDUNION reduces the number of sets by one, there are $m - n$ sets at the end of the sequence, and at most n objects are *not* in singleton sets. Since each of the non-singleton objects had $O(\log n)$ leader changes, the total amount of work done in updating the leader pointers is $O(n \log n)$. □

The aggregate method now implies that each WEIGHTEDUNION has ***amortized cost $O(\log n)$***.

11.3 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: $size(\overline{x}) \ge 2^{rank(\overline{x})}$. (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 $\overline{x}$ changes from $r-1$ to r, mark all the objects in $\overline{x}$'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.

*11.4 $O(\log^* n)$ 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 arguments; Seidel and Sharir's analysis relies on a comparatively simple recursive decomposition. (Of course, simple is in the eye of the beholder.)

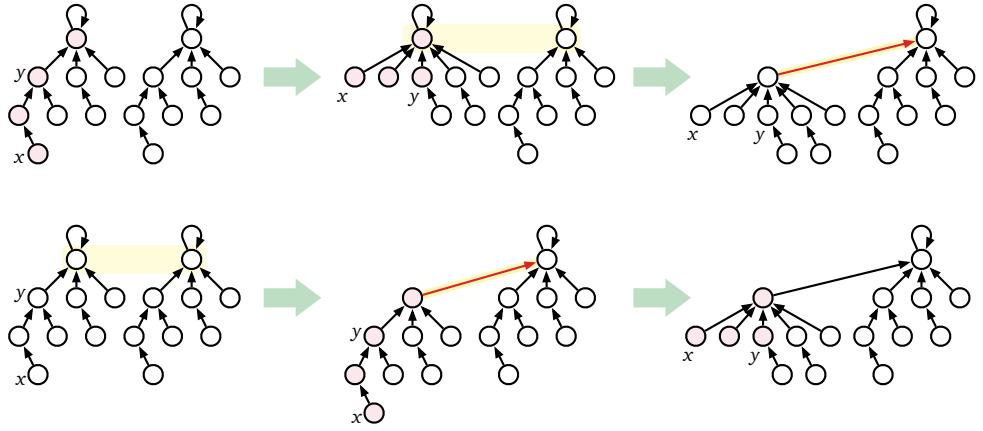
Seidel and Sharir phrase their analysis in terms of two more general operations on set forests. Their more general COMPRESS operation compresses *any* directed path, not just paths that lead to the root. The new SHATTER operation makes every node on a root-to-leaf path into its own parent.

```
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.

[1] Raimund Seidel and Micha Sharir. Top-down analysis of path compression. *SIAM J. Computing* 34(3):515–525, 2005.

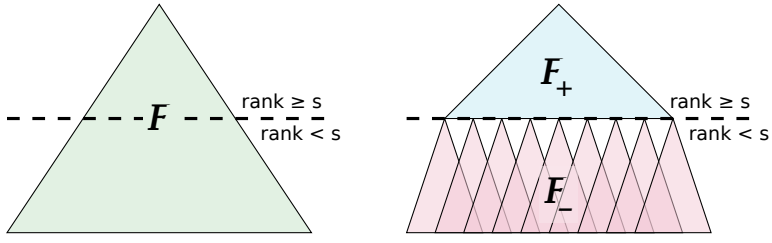
[2] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. Assoc. Comput. Mach.* 22:215–225, 1975.

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. □

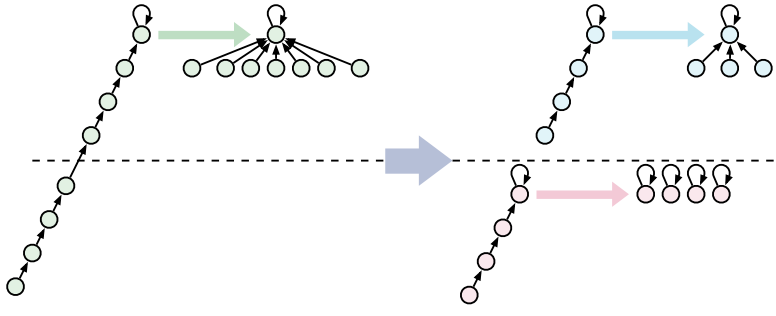
Fix a forest F of n nodes with maximum rank r, and a sequence C of m COMPRESS operations on F, and let $\boldsymbol{T(F,C)}$ denote the total number of pointer assignments executed by this sequence.

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. $T(m,n,r) \le m + 2n \lg^* r$

*11.5 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.

11.6 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. $\mathbf{2}\uparrow\uparrow\boldsymbol{n}$ is the *tower* function:

$$2\uparrow\uparrow n = \underbrace{2\uparrow 2\uparrow\ldots\uparrow 2}_{n} = 2^{2^{\cdot^{\cdot^{2}}}}\Big\}n$$

John Conway named $2\uparrow\uparrow\uparrow n$ the *wower* function:

$$2\uparrow\uparrow\uparrow n = \underbrace{2\uparrow\uparrow 2\uparrow\uparrow\cdots\uparrow\uparrow 2}_{n}.$$

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
$2\uparrow\uparrow n$	2	4	16	65536	2^{65536}
$2\uparrow\uparrow\uparrow n$	2	4	65536	$2^{2^{\cdot^{\cdot^{2}}}}\Big\}65536$	$2^{2^{\cdot^{\cdot^{2}}}}\Big\}2^{2^{\cdot^{\cdot^{2}}}}\Big\}65536$
$2\uparrow\uparrow\uparrow\uparrow n$	2	4	$2^{2^{\cdot^{\cdot^{2}}}}\Big\}65536$	$2^{2^{\cdot^{\cdot^{2}}}}\Big\}2^{2^{\cdot^{\cdot^{2}}}}\Big\}65536\Big\}2^{2^{\cdot^{\cdot^{2}}}}\Big\}65536$	⟨⟨*Yeah, right.*⟩⟩
$2\uparrow\uparrow\uparrow\uparrow\uparrow n$	2	4	$2^{2^{\cdot^{\cdot^{2}}}}\Big\}2^{2^{\cdot^{\cdot^{2}}}}\Big\}65536\Big\}2^{2^{\cdot^{\cdot^{2}}}}\Big\}65536$	⟨⟨*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 $\tilde{\alpha}(n) = \min\{c \ge 1 \mid \lg^{*^{c}} n \le c\} = \min\{c \ge 1 \mid 2\uparrow^{c+2} c \ge n\}$. 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!

11.7 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 $\overline{x}$ stores the number of elements of its set in the field $\textit{weight}(\overline{x})$. 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 $\textsc{Find}(x)$ 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:

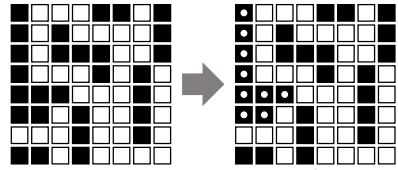
   - NEWSTRING(a) creates a new string of length 1 containing only the character a and returns a pointer to that string.

- 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?]*

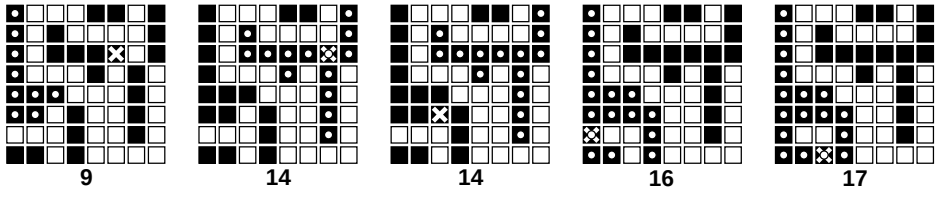
7. (a) Describe and analyze an algorithm to compute the size of the largest connected component of black pixels in an $n \times n$ bitmap $B[1..n, 1..n]$.

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!

12 Lower Bounds

12.1 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!"

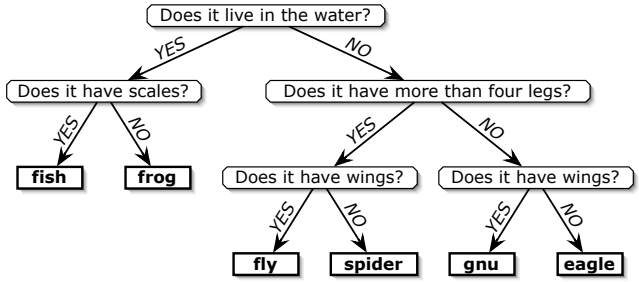
an upper bound, because it's no longer enough to examine a single algorithm. To prove an inequality of the form $T_\Pi(n) = \Omega(f(n))$, we must prove that ***every*** algorithm that solves Π has a worst-case running time $\Omega(f(n))$, or equivalently, that ***no*** algorithm runs in $o(f(n))$ time.

12.2 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.

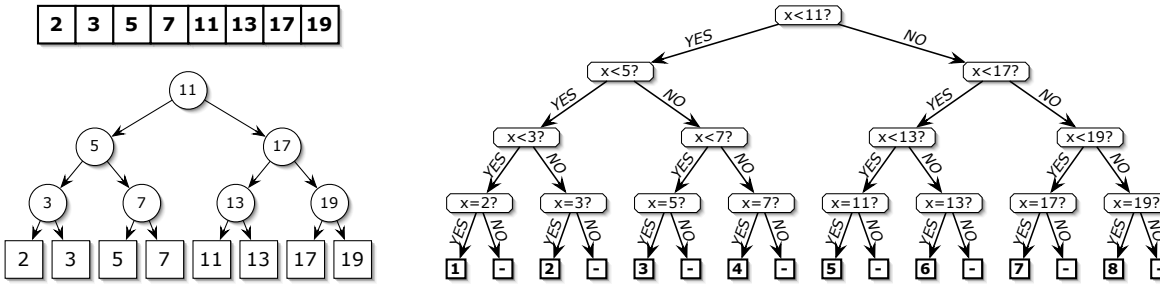


A decision tree to choose one of six animals.

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.**

12.3 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.

12.4 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.)

12.5 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.

12.6 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] 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.]*

13 Adversary Arguments

13.1 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]

13.2 *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.

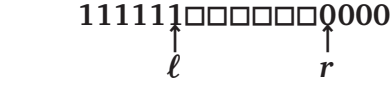
One absolutely crucial feature of this argument is that *the adversary makes absolutely* ***no*** *assumptions about the algorithm*. The adversary strategy can't depend on some predetermined order of examining bits, and it doesn't care about anything the algorithm might or might not do when it's not looking at bits. *Any* algorithm that doesn't examine every bit falls victim to the adversary.

13.3 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 $r = n+1$.



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.

13.4 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.

13.5 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!*

13.6 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 $\mathrm{Aut}(\Delta)$ is vertex-transitive, and let Γ be a vertex-transitive subgroup of $\mathrm{Aut}(\Delta)$. 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.

13.7 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 $X = \langle x_1, x_2, \ldots, x_n \rangle$ 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.

13.8 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 $\langle x_1, x_2, \ldots, x_n \rangle$ 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 $X = \langle x_1, x_2, \ldots, x_n \rangle$ 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.

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. □

> **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. □

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_J such that $1 < d_J < d'$.
>
> Let d_J be an arbitrary divisor of d' such that $1 < d_J < d'$.
>
> Because d'/d_J is an integer, $n/d_J = (n/d') \cdot (d'/d_J)$ is also an integer.
>
> Thus, d_J is also a divisor of n.
>
> Since n has no prime divisors, d_J cannot be prime.
>
> *... blah HELP! blah I'M STUCK IN AN INFINITE LOOP! blah ...*
>
> But that's just silly. Our assumption must be incorrect. □

We seem to be stuck in an infinite loop, looking at smaller and smaller divisors $d > d' > d_J > \cdots$, 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.

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 $p \to q$.

> *If the implication $(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1)) \to P(n)$ 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 "$(P(1) \wedge P(2) \wedge \cdots \wedge P(n-1)) \to P(n)$ 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 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. □

Or more generally:

Bellman's Theorem: Every snark is a boojum.

Proof by induction: Let X be an arbitrary snark.
Assume that for every snark younger than X is a boojum.
There are three cases to consider:

- Suppose X is the youngest snark.
  Then *... blah blah blah ...*
- Suppose X is the second-youngest snark.
  Then *... blah blah blah ...*
- Suppose X is older than the second-youngest snark.
  Then the inductive hypothesis implies *... blah blah blah ...* and therefore *... blah blah blah ...*

An all cases, we conclude that X is a boojum. □

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 $n \leq 28$ (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 $n \leq 28$.

- 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 $n \leq 28$.

. . . 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. $\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$ *for every non-negative integer* n.

First let's write down the induction boilerplate, which empty space for details we'll fill in later.

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 *some number of* cases to consider:

. . . blah blah blah . . .

We conclude that $\sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$. □

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 $\overset{IH}{=\!=}$ 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

$$\sum_{i=0}^{n} 3^i = 3^0 + \sum_{i=1}^{n} 3^i = 3^0 + 3 \cdot \sum_{i=0}^{n-1} 3^i \overset{IH}{=\!=} 3^0 + 3 \cdot \frac{3^n-1}{2} = \frac{3^{n+1}-1}{2}.$$

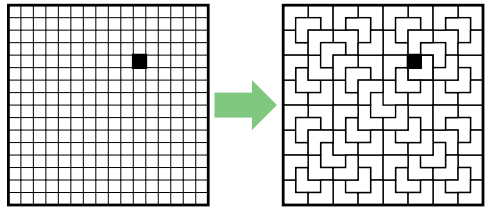
□

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.

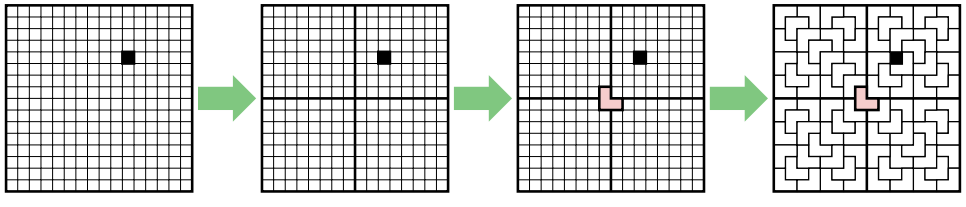
Theorem 5. *For any non-negative integer n, the $2^n \times 2^n$ checkerboard with any square removed can be tiled using L-shaped triominos.*

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. □

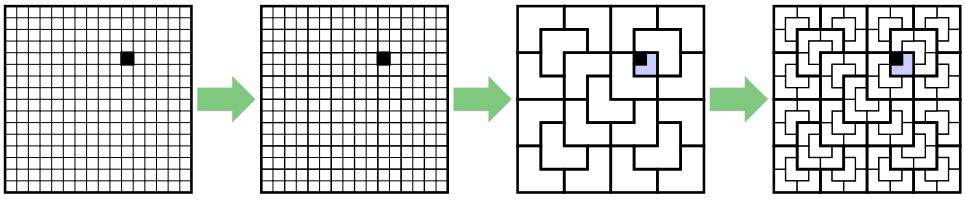


Top-down inductive proof of Theorem 4.

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 both cases, 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*** $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$ 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:

$$\begin{aligned} \langle 4,2,1\rangle &\Longrightarrow \langle 4,1,1,1\rangle \Longrightarrow \langle 3,2,1,1\rangle \Longrightarrow \langle 2,2,2,1\rangle \Longrightarrow \langle 2,2,1,1,1\rangle \\ &\Longrightarrow \langle 2,1,1,1,1,1\rangle \Longrightarrow \langle 2,1,1,1,1\rangle \Longrightarrow \langle 2,1,1,1\rangle \Longrightarrow \langle 2,1,1\rangle \\ &\Longrightarrow \langle 2,1\rangle \Longrightarrow \langle 2\rangle \Longrightarrow \langle 1,1\rangle \Longrightarrow \langle 1\rangle \Longrightarrow \langle\rangle \end{aligned}$$

   (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:

$$\begin{aligned} \langle 4,2,1\rangle &\Longrightarrow \langle 3,3,2\rangle \Longrightarrow \langle 3,2,2,1,1\rangle \Longrightarrow \langle 3,2,2,1\rangle \Longrightarrow \langle 3,2,2\rangle \Longrightarrow \langle 3,2,1,1,1\rangle \\ &\Longrightarrow \langle 2,2,2,2,1\rangle \Longrightarrow \langle 2,2,2,2\rangle \Longrightarrow \langle 2,2,2,1,1,1\rangle \Longrightarrow \langle 2,2,2,1,1\rangle \\ &\Longrightarrow \langle 2,2,2,1\rangle \Longrightarrow \langle 2,2,2\rangle \Longrightarrow \langle 2,2,1,1,1\rangle \Longrightarrow \langle 2,2,1,1\rangle \Longrightarrow \langle 2,2,1\rangle \\ &\Longrightarrow \langle 2,2\rangle \Longrightarrow \langle 2,1,1,1\rangle \Longrightarrow \langle 1,1,1,1,1,1\rangle \Longrightarrow \langle 1,1,1,1,1\rangle \Longrightarrow \langle 1,1,1, \\ &\Longrightarrow \langle 1,1,1\rangle \Longrightarrow \langle 1,1\rangle \Longrightarrow \langle 1\rangle \Longrightarrow \langle\rangle. \end{aligned}$$

      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. A *binomial tree of order k* is defined recursively as follows:

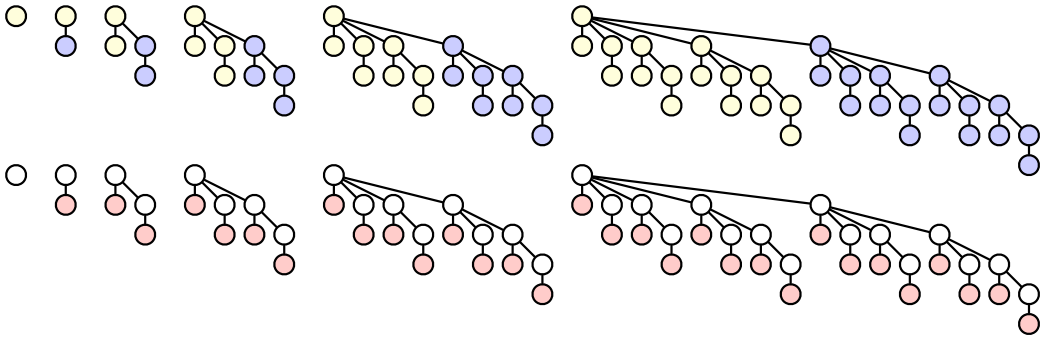
    - A binomial tree of order 0 is a single node.
    - For all $k > 0$, a binomial tree of order k consists of two binomial trees of order $k-1$, with the root of one tree connected as a new child of the root of the other. (See the figure below.)

    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 new 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. (Hence the name!)



Binomial trees of order 0 through 5.
Top row: The recursive definition. Bottom row: The property claimed in part (b).

12. The nth *Fibonacci binary tree* $\mathcal{F}_n$ is defined recursively as follows:

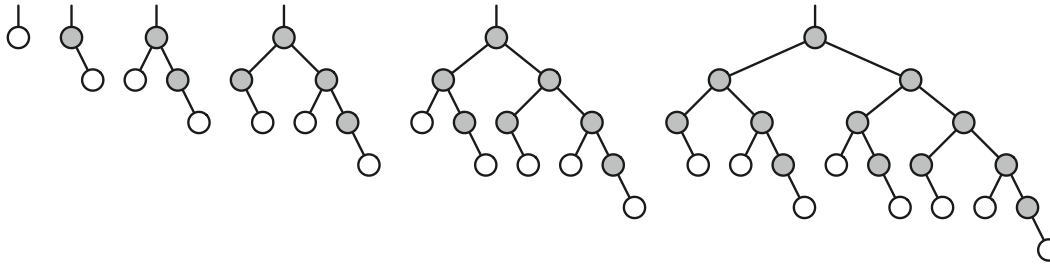
    - $\mathcal{F}_1$ is a single root node with no children.
    - For all $n \geq 2$, $\mathcal{F}_n$ is obtained from $\mathcal{F}_{n-1}$ by adding a right child to every leaf and adding a left child to every node that has only one child.

    (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.

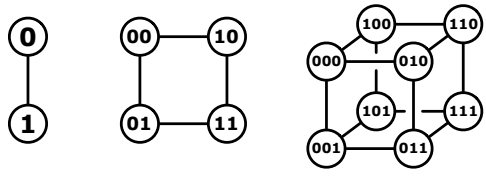
(c) Prove that for all $n \geq 2$, the right subtree of $\mathcal{F}_n$ is a copy of $\mathcal{F}_{n-1}$.

(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 $\mathcal{F}_{n-1}$.

13. 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.



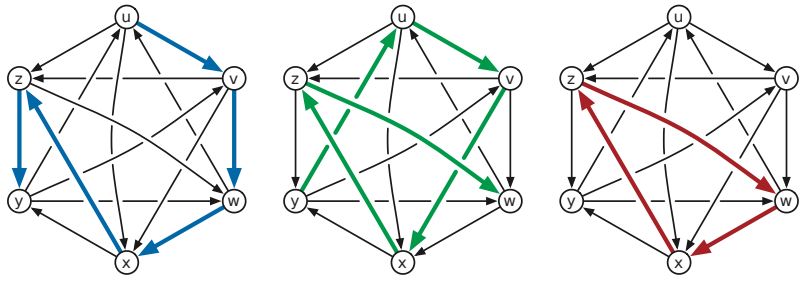
The 1-dimensional, 2-dimensional, and 3-dimensional hypercubes.

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.

14. 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 $w \to x \to z \to w$.

15. 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.

16. ***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:

    $$(((\neg p \vee \neg q) \wedge r)) \wedge ((\neg p \wedge r) \wedge q)$$

17. 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.

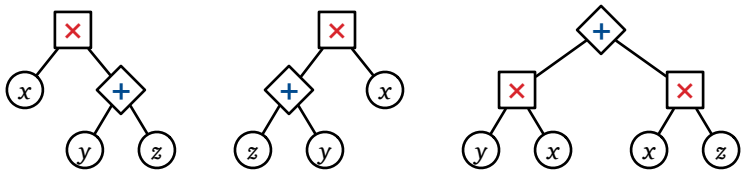
(d) Prove that the derivative f' of a polynomial f is a polynomial, using ***only*** the following facts:

- Constant rule: If f is constant, then f' is identically zero.
- Sum rule: $(f+g)' = f' + g'$.
- Product rule: $(f \cdot g)' = f' \cdot g + f \cdot g'$.

*18. 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. *[Hint: This is harder than it looks.]*

*19. 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:

$$
\begin{aligned}
4 &= 16 + (-8 - 8i) + 8i + (-4) &&= \alpha^8 + \alpha^7 + \alpha^6 + \alpha^4 \\
-8 &= (-8 - 8i) + 8i &&= \alpha^7 + \alpha^6 \\
15i &= (-16 + 16i) + 16 + (-2i) + (-1 + i) + 1 &&= \alpha^9 + \alpha^8 + \alpha^2 + \alpha^1 + \alpha^0 \\
1 + 6i &= (8i) + (-2i) + 1 &&= \alpha^6 + \alpha^2 + \alpha^0 \\
2 - 3i &= (4 - 4i) + (-4) + (2 + 2i) + (-2i) + (-1 + i) + 1 &&= \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha^1 + \alpha \\
-4 + 2i &= (-16 + 16i) + 16 + (-8 - 8i) + (4 - 4i) + (-2i) &&= \alpha^9 + \alpha^8 + \alpha^7 + \alpha^5 + \alpha^2
\end{aligned}
$$

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 $-2 - i$?]

*20. *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 $\lfloor \lg(N+1) \rfloor$.

★21. 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.

22. 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.

*23. 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 any real numbers a and z define four different intervals:

- The ***open*** interval $\mathbf{(a, z)} := \{t \in \mathbb{R} \mid a \le t < z\}$,
- The ***half-open*** interval $\mathbf{[a, z)} := \{t \in \mathbb{R} \mid a \le t < z\}$,
- The ***half-open*** interval $\mathbf{(a, z]} := \{t \in \mathbb{R} \mid a < t \le z\}$,
- The ***closed*** interval $\mathbf{[a, z]} := \{t \in \mathbb{R} \mid a \le t \le z\}$.

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 $a \le s < z$ and $s \in S$, then $[s, u] \subseteq S$ for some $u > s$.

(c) If $a \le s \le z$ and $[a, s) \subseteq S$, then $s \in S$.

Then $S = [a, z]$.

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.

[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.

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!

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 $f(n) = n$:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1)+1 & \text{otherwise} \end{cases} \qquad \begin{array}{l} f(0) = 0 \\ f(n) = f(n-1)+1 \text{ for all } n > 0 \end{array}$$

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.

$$\begin{aligned} T(0) &= 0 = 2^0 - 1 \quad \checkmark \\ T(n) &= 2T(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 && \text{[induction hypothesis]} \\ &= 2^n - 1 \quad \checkmark && \text{[algebra]} \end{aligned}$$

[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 $T(n/b)$.

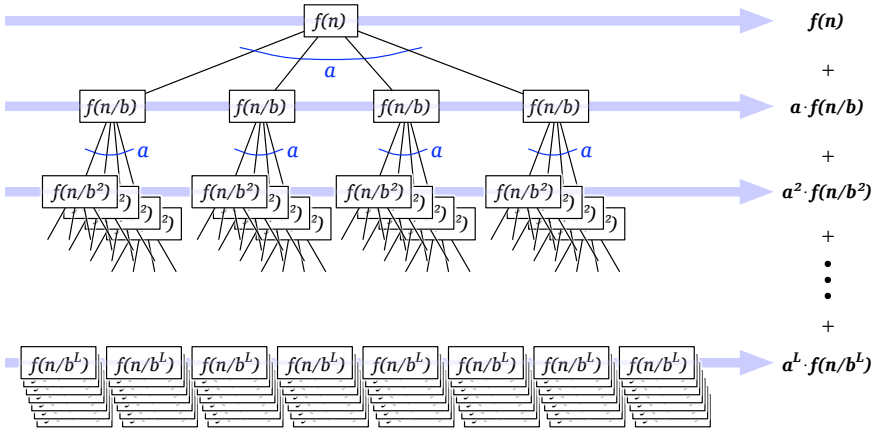
Equivalently, a recursion tree is a complete a-ary tree where each node at depth i contains the value $f(n/b^i)$. The recursion stops when we get to the base case(s) of the recurrence. Because we're only looking for asymptotic bounds, the exact base case doesn't matter; we can safely assume that $T(1) = \Theta(1)$, 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 $\ln(n+1) \le H_n \le \ln n + 1$.

$$T(n) = \sum_{i=0}^{\lg n - 1} \frac{n}{\lg n - i} = \sum_{j=1}^{\lg n} \frac{n}{j} = n H_{\lg n} = \boldsymbol{\Theta(n \lg \lg n)}$$

- **$T(n) = 4T(n/2) + n \lg n$**

  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

  $$\int_1^n \frac{f(u)}{u^{\rho+1}}\,du = \int_1^n u^{-\rho}\,du = \frac{u^{1-\rho}}{1-\rho}\Bigg|_{u=1}^{n} = \frac{n^{1-\rho} - 1}{1-\rho} = \Theta(n^{1-\rho}),$$

  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

  $$\int_1^n \frac{1}{u^{\rho+1}}\,du = \frac{u^{-\rho}}{-\rho}\Bigg|_{u=1}^{n} = \frac{1 - n^{-\rho}}{\rho} = \Theta(1)$$

  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:

$$\begin{aligned}(\boldsymbol{E}-2)(2^n+3^n) &= \boldsymbol{E}(2^n+3^n) - 2(2^n+3^n) \\ &= (2^{n+1}+3^{n+1}) - (2^{n+1}+2\cdot 3^n) = 3^n\end{aligned}$$

$$\implies (\boldsymbol{E}-3)(\boldsymbol{E}-2)(2^n+3^n) = (\boldsymbol{E}-3)3^n = 0$$

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 $(\alpha n + \beta)a^n$:

$$\begin{aligned}(\boldsymbol{E}-a)((\alpha n+\beta)a^n) &= (\alpha(n+1)+\beta)a^{n+1} - a(\alpha n+\beta)a^n \\ &= \alpha a^{n+1}\end{aligned}$$

$$\implies (\boldsymbol{E}-a)^2((\alpha n+\beta)a^n) = (\boldsymbol{E}-a)(\alpha a^{n+1}) = 0$$

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:

- $\boldsymbol{r(n) = 5r(n-1)}$**, where** $\boldsymbol{r(0) = 3}$**.**

  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:

  $$\begin{aligned} r(0) &= 3 \cdot 5^0 = 3 \quad \checkmark && \text{[definition]} \\ r(n) &= 5r(n-1) && \text{[definition]} \\ &= 5 \cdot (3 \cdot 5^{n-1}) && \text{[induction hypothesis]} \\ &= 5^n \cdot 3 \quad \checkmark && \text{[algebra]} \end{aligned}$$

- **Fibonacci numbers:** $\boldsymbol{F(n) = F(n-1) + F(n-2)}$**, where** $\boldsymbol{F(0) = 0}$ **and** $\boldsymbol{F(1) = 1}$**.**

  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 $(E-2)T(n) = 1$.
2. The operator $(E-2)$ doesn't quite annihilate the function; it leaves a *residue* of 1. But we can annihilate the residue by applying the operator $(E-1)$. Thus, the compound operator $(E-1)(E-2)$ 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 $(E^2 - E - 1)H = 1$. The residue 1 is annihilated by $(E-1)$, so the compound operator $(E-1)(E^2-E-1)$ annihilates the recurrence. This operator factors into $(E-1)(E-\phi)(E-\hat{\phi})$, 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:

$$\begin{aligned} H(-1) = 0 &= \alpha\phi^{-1} + \beta + \gamma\hat{\phi}^{-1} &&= \alpha/\phi + \beta - \gamma/\hat{\phi} \\ H(0) = 1 &= \alpha\phi^0 + \beta + \gamma\hat{\phi}^0 &&= \alpha + \beta + \gamma \\ H(1) = 2 &= \alpha\phi^1 + \beta + \gamma\hat{\phi}^1 &&= \alpha\phi + \beta + \gamma\hat{\phi} \end{aligned}$$

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 $(E-2)^2(E-1)$, 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 $(E^2 - E - 2)T(n) = E^2(2^n - n^2)$. Notice that we had to shift up the non-recursive parts of the recurrence when we expressed it in this form. The operator $(E-2)(E-1)^3$ annihilates the residue $2^n - n^2$, and therefore also annihilates the shifted residue $E^2(2^n + n^2)$. Thus, the operator $(E-2)(E-1)^3(E^2 - E - 2)$ annihilates the entire recurrence. We can factor the quadratic factor into $(E-2)(E+1)$, so the annihilator factors into $(E-2)^2(E-1)^3(E+1)$. 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.

$$n \cdot T(n) = 2\sum_{k=0}^{n-1} T(j) + n^2 \qquad \text{[multiply both sides by } n\text{]}$$

$$(n-1) \cdot T(n-1) = 2\sum_{k=0}^{n-2} T(j) + (n-1)^2 \qquad \text{[shift]}$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1 \qquad \text{[subtract]}$$

$$T(n) = \frac{n+1}{n}T(n-1) + 2 - \frac{1}{n} \qquad \text{[simplify]}$$

[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. . .