Randomization
The first nuts and bolts appeared in the middle 1400’s. The bolts were just screws with straight sides and a blunt end. The nuts were hand-made, and very crude. When a match was found between a nut and a bolt, they were kept together until they were finally assembled.

In the Industrial Revolution, it soon became obvious that threaded fasteners made it easier to assemble products, and they also meant more reliable products. But the next big step came in 1801, with Eli Whitney, the inventor of the cotton gin. The lathe had been recently improved. Batches of bolts could now be cut on different lathes, and they would all fit the same nut.

Whitney set up a demonstration for President Adams, and Vice-President Jefferson. He had piles of musket parts on a table. There were 10 similar parts in each pile. He went from pile to pile, picking up a part at random. Using these completely random parts, he quickly put together a working musket.

— Karl S. Kruszelnicki (“Dr. Karl”), Karl Trek, December 1997

Dr [John von] Neumann in his Theory of Games and Economic Behavior introduces the cut-up method of random action into game and military strategy: Assume that the worst has happened and act accordingly. If your strategy is at some point determined... by random factor your opponent will gain no advantage from knowing your strategy since he cannot predict the move. The cut-up method could be used to advantage in processing scientific data. How many discoveries have been made by accident? We cannot produce accidents to order.


9 Randomized Algorithms

9.1 Nuts and Bolts

Suppose we are given $n$ nuts and $n$ bolts of different sizes. Each nut matches exactly one bolt and vice versa. The nuts and bolts are all almost exactly the same size, so we can’t tell if one bolt is bigger than the other, or if one nut is bigger than the other. If we try to match a nut with a bolt, however, the nut will be either too big, too small, or just right for the bolt.

Our task is to match each nut to its corresponding bolt. But before we do this, let’s try to solve some simpler problems, just to get a feel for what we can and can’t do.

Suppose we want to find the nut that matches a particular bolt. The obvious algorithm — test every nut until we find a match — requires exactly $n - 1$ tests in the worst case. We might have to check every bolt except one; if we get down the the last bolt without finding a match, we know that the last nut is the one we’re looking for.\(^1\)

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

9.2 Deterministic vs. Randomized Algorithms

Normally, when we talk about the running time of an algorithm, we mean the worst-case running time. This is the maximum, over all problems of a certain size, of the running time of that algorithm on that input:

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

On extremely rare occasions, we will also be interested in the best-case running time:

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

\(^1\)“Whenever you lose something, it’s always in the last place you look. So why not just look there first?”
The average-case running time is best defined by the expected value, over all inputs \( X \) of a certain size, of the algorithm’s running time for \( X \):\(^2\)

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

The problem with this definition is that we rarely, if ever, know what the probability of getting any particular input \( X \) is. We could compute average-case running times by assuming a particular probability distribution—for example, every possible input is equally likely—but this assumption doesn’t describe reality very well. Most real-life data is decidedly non-random (or at least random in some unpredictable way).

Instead of considering this rather questionable notion of average case running time, we will make a distinction between two kinds of algorithms: deterministic and randomized. A deterministic algorithm is one that always behaves the same way given the same input; the input completely determines the sequence of computations performed by the algorithm. Randomized algorithms, on the other hand, base their behavior not only on the input but also on several random choices. The same randomized algorithm, given the same input multiple times, may perform different computations in each invocation.

This means, among other things, that the running time of a randomized algorithm on a given input is no longer fixed, but is itself a random variable. When we analyze randomized algorithms, we are typically interested in the worst-case expected running time. That is, we look at the average running time for each input, and then choose the maximum over all inputs of a certain size:

\[
T_{\text{worst-case expected}}(n) = \max_{|X|=n} \mathbb{E}[T(X)].
\]

It’s important to note here that we are making no assumptions about the probability distribution of possible inputs. All the randomness is inside the algorithm, where we can control it!

### 9.3 Back to Nuts and Bolts

Let’s go back to the problem of finding the nut that matches a given bolt. Suppose we use the same algorithm as before, but at each step we choose a nut uniformly at random from the untested nuts. ‘Uniformly’ is a technical term meaning that each nut has exactly the same probability of being chosen.\(^3\) So if there are \( k \) nuts left to test, each one will be chosen with probability \( 1/k \). Now what’s the expected number of comparisons we have to perform? Intuitively, it should be about \( n/2 \), but let’s formalize our intuition.

Let \( T(n) \) denote the number of comparisons our algorithm uses to find a match for a single bolt out of \( n \) nuts.\(^4\) We still have some simple base cases \( T(1) = 0 \) and \( T(2) = 1 \), but when \( n > 2 \), \( T(n) \) is a random variable. \( T(n) \) is always between 1 and \( n-1 \); it’s actual value depends on our algorithm’s random choices. We are interested in the expected value or expectation of \( T(n) \), which is defined as follows:

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

---

\(^2\)The notation \( \mathbb{E}[\cdot] \) for expectation has nothing to do with the shift operator \( \mathbb{E} \) used in the annihilator method for solving recurrences!

\(^3\)This is what most people think ‘random’ means, but they’re wrong.

\(^4\)Note that for this algorithm, the input is completely specified by the number \( n \). Since we’re choosing the nuts to test at random, even the order in which the nuts and bolts are presented doesn’t matter. That’s why I’m using the simpler notation \( T(n) \) instead of \( T(X) \).
If the target nut is the $k$th 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 $k$th bolt tested, for any $k$. Thus:

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

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

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

$$= \sum_{k=1}^{n-1} \frac{k}{n} + \frac{n-1}{n}$$

$$= \frac{n(n-1)}{2n} + 1 - \frac{1}{n}$$

$$= \frac{n+1}{2} - \frac{1}{n}$$

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:

$$T(1) = 0, \quad E[T(n)] = 1 + \frac{n-1}{n} E[T(n-1)]$$

To get the solution, we define a new function $t(n) = n E[T(n)]$ and rewrite:

$$t(1) = 0, \quad t(n) = n + t(n-1)$$

This recurrence translates into a simple summation, which we can easily solve.

$$t(n) = \sum_{k=2}^{n} k = \frac{n(n+1)}{2} - 1$$

$$\Rightarrow E[T(n)] = \frac{t(n)}{n} = \frac{n+1}{2} - \frac{1}{n}$$

### 9.4 Finding All Matches

Let’s go back to the problem introduced at the beginning of the lecture: finding the matching nut for every bolt. The simplest algorithm simply compares every nut with every bolt, for a total of $n^2$ comparisons. The next thing we might try is repeatedly finding an arbitrary matched pair, using our very first nuts and bolts algorithm. This requires

$$\sum_{i=1}^{n} (i - 1) = \frac{n^2 - n}{2}$$

comparisons in the worst case. So we save roughly a factor of two over the really stupid algorithm. Not very exciting.
Here’s another possibility. Choose a pivot bolt, and test it against every nut. Then test the matching pivot nut against every other bolt. After these $2n - 1$ tests, we have one matched pair, and the remaining nuts and bolts are partitioned into two subsets: those smaller than the pivot pair and those larger than the pivot pair. Finally, recursively match up the two subsets. The worst-case number of tests made by this algorithm is given by the recurrence

$$T(n) = 2n - 1 + \max_{1 \leq k \leq n} \{T(k - 1) + T(n - k)\}$$

Along with the trivial base case $T(0) = 0$, this recurrence solves to

$$T(n) = \sum_{i=1}^{n} (2n - 1) = n^2.$$

In the worst case, this algorithm tests every nut-bolt pair! We could have been a little more clever—for example, if the pivot bolt is the smallest bolt, we only need $n - 1$ tests to partition everything, not $2n - 1$—but cleverness doesn’t actually help that much. We still end up with about $n^2/2$ tests in the worst case.

However, since this recursive algorithm looks almost exactly like quicksort, and everybody ‘knows’ that the ‘average-case’ running time of quicksort is $\Theta(n \log n)$, it seems reasonable to guess that the average number of nut-bolt comparisons is also $\Theta(n \log n)$. As we shall see shortly, if the pivot bolt is always chosen uniformly at random, this intuition is exactly right.

### 9.5 Reductions to and from Sorting

The second algorithm for matching up the nuts and bolts looks exactly like quicksort. The algorithm not only matches up the nuts and bolts, but also sorts them by size.

In fact, the problems of sorting and matching nuts and bolts are equivalent, in the following sense. If the bolts were sorted, we could match the nuts and bolts in $O(n \log n)$ time by performing a binary search with each nut. Thus, if we had an algorithm to sort the bolts in $O(n \log n)$ time, we would immediately have an algorithm to match the nuts and bolts, starting from scratch, in $O(n \log n)$ time. This process of assuming a solution to one problem and using it to solve another is called reduction—we can reduce the matching problem to the sorting problem in $O(n \log n)$ time.

There is a reduction in the other direction, too. If the nuts and bolts were matched, we could sort them in $O(n \log n)$ time using, for example, merge sort. Thus, if we have an $O(n \log n)$ time algorithm for either sorting or matching nuts and bolts, we automatically have an $O(n \log n)$ time algorithm for the other problem.

Unfortunately, since we aren’t allowed to directly compare two bolts or two nuts, we can’t use heapsort or mergesort to sort the nuts and bolts in $O(n \log n)$ worst case time. In fact, the problem of sorting nuts and bolts deterministically in $O(n \log n)$ time was only ‘solved’ in 1995, but both the algorithms and their analysis are incredibly technical and the constant hidden in the $O(\cdot)$ notation is quite large.

Reductions will come up again later in the course when we start talking about lower bounds and NP-completeness.  

---

9.6 Recursive Analysis

Intuitively, we can argue that our quicksort-like algorithm will usually choose a bolt of approximately median size, and so the average numbers of tests should be $O(n \log n)$. We can now finally formalize this intuition. To simplify the notation slightly, I’ll write $\bar{T}(n)$ in place of $E[T(n)]$ 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 $k$th smallest for any $k$, the expected number of tests performed by our algorithm is given by the following recurrence:

\[
\bar{T}(n) = 2n - 1 + E_k \left[ \bar{T}(k - 1) + \bar{T}(n - k) \right]
\]

The base case is $T(0) = 0$. (We can save a few tests by setting $T(1) = 0$ instead of 1, but the analysis will be easier if we’re a little stupid.)

Yuck. At this point, we could simply guess the solution, based on the incessant rumors that quicksort runs in $O(n \log n)$ time in the average case, and prove our guess correct by induction. (See Section 9.8 below for details.)

However, if we’re only interested in asymptotic bounds, we can afford to be a little conservative. What we’d really like is for the pivot bolt to be the median bolt, so that half the bolts are bigger and half the bolts are smaller. This isn’t very likely, but there is a good chance that the pivot bolt is close to the median bolt. Let’s say that a pivot bolt is good if it’s in the middle half of the final sorted set of bolts, that is, bigger than at least $\frac{n}{4}$ bolts and smaller than at least $\frac{n}{4}$ 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:

\[
\bar{T}(n) \leq 2n - 1 + \frac{1}{2} \left( \bar{T}(\frac{3n}{4}) + \bar{T}(\frac{n}{4}) \right) + \frac{1}{2} \cdot \bar{T}(n)
\]

A little algebra simplifies this even further:

\[
\bar{T}(n) \leq 4n - 2 + \bar{T}(\frac{3n}{4}) + \bar{T}(\frac{n}{4})
\]

We can solve this recurrence using the recursion tree method, giving us the unsurprising upper bound $\bar{T}(n) = O(n \log n)$. A similar argument gives us the matching lower bound $\bar{T}(n) = \Omega(n \log n)$.

Unfortunately, while this argument is convincing, it is not a formal proof, because it relies on the unproven assumption that $\bar{T}(n)$ is a convex function, which means that $\bar{T}(n+1) + \bar{T}(n-1) \geq 2\bar{T}(n)$ for all $n$. $\bar{T}(n)$ is actually convex, but we never proved it. Convexity follows form the closed-form solution of the recurrence, but using that fact would be circular logic. Sadly, formally proving convexity seems to be almost as hard as solving the recurrence. If we want a proof of the expected cost of our algorithm, we need another way to proceed.
9.7 Iterative Analysis

By making a simple change to our algorithm, which has no effect on the number of tests, we can analyze it much more directly and exactly, without solving a recurrence or relying on hand-wavy intuition.

The recursive subproblems solved by quicksort can be laid out in a binary tree, where each node corresponds to a subset of the nuts and bolts. In the usual recursive formulation, the algorithm partitions the nuts and bolts at the root, then the left child of the root, then the leftmost grandchild, and so forth, recursively sorting everything on the left before starting on the right subproblem.

But we don’t have to solve the subproblems in this order. In fact, we can visit the nodes in the recursion tree in any order we like, as long as the root is visited first, and any other node is visited after its parent. Thus, we can recast quicksort in the following iterative form. Choose a pivot bolt, find its match, and partition the remaining nuts and bolts into two subsets. Then pick a second pivot bolt and partition whichever of the two subsets contains it. At this point, we have two matched pairs and three subsets of nuts and bolts. Continue choosing new pivot bolts and partitioning subsets, each time finding one match and increasing the number of subsets by one, until every bolt has been chosen as the pivot. At the end, every bolt has been matched, and the nuts and bolts are sorted.

Suppose we always choose the next pivot bolt uniformly at random from the bolts that haven’t been pivots yet. Then no matter which subset contains this bolt, the pivot bolt is equally likely to be any bolt in that subset. That implies (by induction) that our randomized iterative algorithm performs exactly the same set of tests as our randomized recursive algorithm, but possibly in a different order.

Now let $B_i$ denote the $i$th smallest bolt, and $N_j$ denote the $j$th 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:

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

This equation uses a crucial property of random variables called linearity of expectation: for any random variables $X$ and $Y$, the sum of their expectations is equal to the expectation of their sum: $E[X + Y] = E[X] + E[Y]$.

To analyze our algorithm, we only need to compute the expected value of each $X_{ij}$. By definition of expectation,

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

so we just need to calculate $Pr[X_{ij} = 1]$ for all $i$ and $j$.

First let’s assume that $i < j$. The only comparisons our algorithm performs are between some pivot bolt (or its partner) and a nut (or bolt) in the same subset. The only event that can prevent a comparison between $B_i$ and $N_j$ is choosing some intermediate pivot bolt $B_k$, with $i < k < j$, before $B_i$ or $B_j$. In other words:
Our algorithm compares \( B_i \) and \( N_j \) if and only if the first pivot chosen from the set \( \{B_i, B_{i+1}, \ldots, B_j\} \) is either \( B_i \) or \( B_j \).

Since the set \( \{B_i, B_{i+1}, \ldots, B_j\} \) contains \( j - i + 1 \) bolts, each of which is equally likely to be chosen first, we immediately have

\[
E[X_{ij}] = \frac{2}{j - i + 1} \quad \text{for all } i < j.
\]

Symmetric arguments give us \( E[X_{ij}] = \frac{2}{i - j + 1} \) 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.

\[
E[T(n)] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}]
\]

\[
= \sum_{i=1}^{n} E[X_{ii}] + 2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}]
\]

\[
= n + 4 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j - i + 1}
\]

This is quite a bit simpler than the recurrence we got before. With just a few more lines of algebra, we can turn it into an exact, closed-form expression for the expected number of comparisons.

\[
E[T(n)] = n + 4 \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{1}{k} \quad \text{[substitute } k = j - i + 1]\]

\[
= n + 4 \sum_{k=2}^{n} \sum_{i=1}^{n-k+1} \frac{1}{k} \quad \text{[reorder summations]}\]

\[
= n + 4 \sum_{k=2}^{n} \frac{n - k + 1}{k}
\]

\[
= n + 4 \left( (n + 1) \sum_{k=2}^{n} \frac{1}{k} - \sum_{k=2}^{n} \frac{1}{k} \right)
\]

\[
= n + 4((n + 1)(H_n - 1) - (n - 1))
\]

\[
= n + 4(nH_n - 2n + H_n)
\]

\[
= 4nH_n - 7n + 4H_n
\]

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

### 9.8 Masochistic Analysis

If we’re feeling particularly masochistic, we can actually solve the recurrence directly, all the way to an exact closed-form solution. I’m including this only to show you it can be done; this won’t be on the test.
First we simplify the recurrence slightly by combining symmetric terms.

\[
\overline{T}(n) = \frac{1}{n} \sum_{k=1}^{n} (\overline{T}(k-1) + \overline{T}(n-k)) + 2n - 1 = \frac{2}{n} \sum_{k=0}^{n-1} \overline{T}(k) + 2n - 1
\]

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.

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

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

\[
= 2 \sum_{k=0}^{n-2} \overline{T}(k) + 2n^2 - 5n + 3
\]

\[
n\overline{T}(n) - (n-1)\overline{T}(n-1) = 2T(n-1) + 4n - 3
\]

\[
\overline{T}(n) = \frac{n+1}{n}\overline{T}(n-1) + 4 - \frac{3}{n}
\]

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 \(t(n-1)\):

\[
t(n) = \frac{\overline{T}(n)}{n+1}
\]

\[
= \frac{1}{n+1} \left( (n+1) \frac{T(n-1)}{n} + 4 - \frac{3}{n} \right)
\]

\[
= t(n-1) + \frac{4}{n+1} - \frac{3}{n(n+1)}
\]

\[
= t(n-1) + \frac{7}{n+1} - \frac{3}{n}
\]

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.

\[
t(n) = \sum_{i=1}^{n} \left( \frac{7}{i+1} - \frac{3}{i} \right)
\]

\[
= 7 \sum_{i=1}^{n} \frac{1}{i+1} - 3 \sum_{i=1}^{n} \frac{1}{i}
\]

\[
= 7(H_{n+1} - 1) - 3H_n
\]

\[
= 4H_n - 7 + \frac{7}{n+1}
\]

The last step uses the recursive definition of the harmonic numbers: \(H_n = H_n + \frac{1}{n+1}\). Finally, substituting \(\overline{T}(n) = (n+1)t(n)\) and simplifying gives us the exact solution to the original recurrence.

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

Surprise, surprise, we get exactly the same solution!
Exercises

Probability

Several of these problems refer to decks of playing cards. A standard (Anglo-American) deck of 52 playing cards contains 13 cards in each of four suits: spades ($\spadesuit$), hearts (♥), diamonds (♦), and clubs (♣). Within each suit, the 13 cards have distinct ranks: 2, 3, 4, 5, 6, 7, 8, 9, 10, jack (J), queen (Q), king (K), and ace (A). For purposes of these problems, the ranks are ordered $A < 2 < 3 < \cdots < 9 < 10 < J < Q < K$; thus, for example, the jack of spades has higher rank than the eight of diamonds.

1. Clock Solitaire is played with a standard deck of playing cards. To set up the game, deal the cards face down into 13 piles of four cards each, one in each of the ‘hour’ positions of a clock and one in the center. Each pile corresponds to a particular rank—A through Q in clockwise order for the hour positions, and K for the center. To start the game, turn over a card in the center pile. Then repeatedly turn over a card in the pile corresponding to the value of the previous card. The game ends when you try to turn over a card from a pile whose four cards are already face up. (This is always the center pile—why?) You win if and only if every card is face up when the game ends. What is the exact probability that you win a game of Clock Solitaire, assuming that the cards are permuted uniformly at random before they are dealt into their piles?

2. Professor Jay is about to perform a public demonstration with two decks of cards, one with red backs (‘the red deck’) and one with blue backs (‘the blue deck’). Both decks lie face-down on a table in front of Professor Jay, shuffled so that every permutation of each deck is equally likely.

To begin the demonstration, Professor Jay turns over the top card from each deck. If one of these two cards is the three of clubs ($\clubsuit$), 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 $\clubsuit$ is turned over. Thus, if $\clubsuit$ is the last card in both decks, the demonstration ends with 102 cards embedded in the watermelon, that most prodigious of household fruits.

(a) What is the exact expected number of cards that Professor Jay hurls into the watermelon?

(b) For each of the statements below, give the exact probability that the statement is true of the first pair of cards Professor Jay turns over.
   i. Both cards are threes.
   ii. One card is a three, and the other card is a club.
   iii. If (at least) one card is a heart, then (at least) one card is a diamond.
   iv. The card from the red deck has higher rank than the card from the blue deck.

(c) For each of the statements below, give the exact probability that the statement is true of the last pair of cards Professor Jay turns over.
   i. Both cards are threes.
   ii. One card is a three, and the other card is a club.
iii. If (at least) one card is a heart, then (at least) one card is a diamond.
iv. The card from the red deck has higher rank than the card from the blue deck.

3. Penn and Teller agree to play the following game. Penn shuffles a standard deck of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3♣), at which point the remaining undrawn cards instantly burst into flames.

The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn. To make the rules unambiguous, they agree beforehand that A = 1, J = 11, Q = 12, and K = 13.

(a) What is the expected number of cards that Teller draws?
(b) What is the expected maximum value among the cards Teller gives to Penn?
(c) What is the expected minimum value among the cards Teller gives to Penn?
(d) What is the expected number of cards that Teller gives to Penn? [Hint: Let 13 = n.]

4. Suppose n lights labeled 0, . . . , n − 1 are placed clockwise around a circle. Initially, every light is off. Consider the following random process.

\[
\text{LIGHTTHECIRCLE}(n): \\
k \leftarrow 0 \\
\text{turn on light } 0 \\
\text{while at least one light is off} \\
\quad \text{with probability } 1/2 \\
\quad k \leftarrow (k + 1) \mod n \\
\text{else} \\
\quad k \leftarrow (k - 1) \mod n \\
\text{if light } k \text{ is off, turn it on}
\]

(a) Let \( p(i, n) \) be the probability that light \( i \) is the last to be turned on by LIGHTTHECIRCLE\( (n, 0) \). For example, \( p(0, 2) = 0 \) and \( p(1, 2) = 1 \). Find an exact closed-form expression for \( p(i, n) \) in terms of \( n \) and \( i \). Prove your answer is correct.

(b) Give the tightest upper bound you can on the expected running time of this algorithm.

5. Consider a random walk on a path with vertices numbered 1, 2, . . . , n from left to right. At each step, we flip a coin to decide which direction to walk, moving one step left or one step right with equal probability. The random walk ends when we fall off one end of the path, either by moving left from vertex 1 or by moving right from vertex \( n \).

(a) Prove that the probability that the walk ends by falling off the right end of the path is exactly \( 1/(n + 1) \).

(b) Prove that if we start at vertex \( k \), the probability that we fall off the right end of the path is exactly \( k/(n + 1) \).

\(^6\text{Specifically, hehurls it directly into the back of Penn's right hand.}\)
(c) Prove that if we start at vertex 1, the expected number of steps before the random walk ends is exactly $n$.
(d) Suppose we start at vertex $n/2$ instead. State and prove a tight $\Theta$-bound on the expected length of the random walk in this case.

Randomized Algorithms

6. Consider the following randomized algorithm for generating biased random bits. The subroutine $\text{FAIRCOIN}$ returns either 0 or 1 with equal probability; the random bits returned by $\text{FAIRCOIN}$ are mutually independent.

```
\text{ONEINTHREE:}
  \text{if FAIRCOIN} = 0
  \quad \text{return 0}
  \text{else}
  \quad \text{return } 1 - \text{ONEINTHREE}
```

(a) Prove that $\text{ONEINTHREE}$ returns 1 with probability $1/3$.
(b) What is the exact expected number of times that this algorithm calls $\text{FAIRCOIN}$?
(c) Now suppose you are given a subroutine $\text{ONEINTHREE}$ that generates a random bit that is equal to 1 with probability $1/3$. Describe a $\text{FAIRCOIN}$ algorithm that returns either 0 or 1 with equal probability, using $\text{ONEINTHREE}$ as your only source of randomness.
(d) What is the exact expected number of times that your $\text{FAIRCOIN}$ algorithm calls $\text{ONEINTHREE}$?

7. (a) Suppose you have access to a function $\text{FAIRCOIN}$ that returns a single random bit, chosen uniformly and independently from the set $\{0, 1\}$, in $O(1)$ time. Describe and analyze an algorithm $\text{RANDOM(n)}$, which returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$.
(b) Suppose you have access to a function $\text{FAIRCOINS}(k)$ that returns $k$ random bits (or equivalently, a random integer chosen uniformly and independently from the set $\{0, 1, \ldots, 2^k - 1\}$) in $O(1)$ time, given any non-negative integer $k$ as input. Describe and analyze an algorithm $\text{RANDOM(n)}$, which returns an integer chosen uniformly and independently at random from the set $\{1, 2, \ldots, n\}$.

For each of the remaining problems, you may assume a function $\text{RANDOM}(k)$ that returns, given any positive integer $k$, an integer chosen independently and uniformly at random from the set $\{1, 2, \ldots, k\}$, in $O(1)$ time. For example, to perform a fair coin flip, one could call $\text{RANDOM}(2)$.

8. Consider the following algorithm for finding the smallest element in an unsorted array:

```
\text{RANDOMMIN}(A[1..n]):
  \quad min \leftarrow \infty
  \text{for } i \leftarrow 1 \text{ to } n \text{ in random order}
  \quad \text{if } A[i] < min
  \quad \quad min \leftarrow A[i] \quad (*)
  \text{return } min
```
(a) In the worst case, how many times does RANDOMMIN execute line (⋆)?

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

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

9. Consider the following randomized algorithm for choosing the largest bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

What is the exact expected number of nut-bolt tests performed by this algorithm? Prove your answer is correct. [Hint: What is the expected number of unchosen nuts and bolts when the algorithm terminates?]

10. Let S be a set of n points in the plane. A point p in S is called Pareto-optimal if no other point in S is both above and to the right of p.

(a) Describe and analyze a deterministic algorithm that computes the Pareto-optimal points in S in O(n log n) time.

(b) Suppose each point in S is chosen independently and uniformly at random from the unit square [0, 1] × [0, 1]. What is the exact expected number of Pareto-optimal points in S?

11. Suppose we want to write an efficient function RANDOMPERMUTATION(n) that returns a permutation of the integers {1, ..., n} chosen uniformly at random.

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

RandomPermutation(n):
for i ← 1 to n
π[i] ← i
for i ← 1 to n
swap π[i] ← π[Random(n)]

(b) Consider the following implementation of RANDOMPERMUTATION.

RandomPermutation(n):
for i ← 1 to n
π[i] ← NULL
for i ← 1 to n
j ← Random(n)
while (π[j] != NULL)
j ← Random(n)
π[j] ← i
return π

Prove that this algorithm is correct. Analyze its expected runtime.

(c) Consider the following partial implementation of RANDOMPERMUTATION.
Prove that if the subroutine SomeFunction is deterministic, then this algorithm cannot be correct. [Hint: There is a one-line proof.]

(d) Describe and analyze an implementation of RandomPermutation that runs in expected worst-case time \( O(n) \).

(e) Describe and analyze an implementation of RandomPermutation that runs in expected worst-case time \( O(n \log n) \), using fair coin flips (instead of Random) as the only source of randomness.

⋆(f) Consider a correct implementation of RandomPermutation\((n)\) with the following property: whenever it calls Random\((k)\), the argument \( k \) is at most \( m \). Prove that this algorithm always calls Random at least \( \Omega(\frac{n \log n}{\log m}) \) times.

12. A data stream is an extremely long sequence of items that you can only read only once, in order. A good example of a data stream is the sequence of packets that pass through a router. Data stream algorithms must process each item in the stream quickly, using very little memory; there is simply too much data to store, and it arrives too quickly for any complex computations. Every data stream algorithm looks roughly like this:

```
DoSomethingInteresting(stream S):
    repeat
        x ← next item in S
        ⟨⟨do something fast with x⟩⟩
    until S ends
    return ⟨⟨something⟩⟩
```

Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, without knowing the length of the stream in advance. Your algorithm should spend \( O(1) \) time per stream element and use \( O(1) \) space (not counting the stream itself).

13. Consider the following randomized variant of one-armed quicksort, which selects the \( k \)th smallest element in an unsorted array \( A[1..n] \). As usual, Partition\((A[1..n], p)\) partitions the array \( A \) into three parts by comparing the pivot element \( A[p] \) to every other element, using \( n - 1 \) comparisons, and returns the new index of the pivot element.

```
QuickSelect(A[1..n], k):
    r ← Partition(A[1..n], Random(n))
    if k < r
        return QuickSelect(A[1..r − 1], k)
    else if k > r
        return QuickSelect(A[r + 1..n], k − r)
    else
        return A[k]
```
(a) State a recurrence for the expected running time of QuickSelect, as a function of \( n \) and \( k \).

(b) What is the exact probability that QuickSelect compares the \( i \)th smallest and \( j \)th smallest elements in the input array? The correct answer is a simple function of \( i, j \), and \( k \). [Hint: Check your answer by trying a few small examples.]

(c) What is the exact probability that in one of the recursive calls to QuickSelect, the first argument is the subarray \( A[i..j] \)? The correct answer is a simple function of \( i, j \), and \( k \). [Hint: Check your answer by trying a few small examples.]

(d) Show that for any \( n \) and \( k \), the expected running time of QuickSelect is \( \Theta(n) \). You can use either the recurrence from part (a) or the probabilities from part (b) or (c). For extra credit, find the exact expected number of comparisons, as a function of \( n \) and \( k \).

(e) What is the expected number of times that QuickSelect calls itself recursively?

14. Let \( M[1..n, 1..n] \) be an \( n \times n \) matrix in which every row and every column is sorted. Such an array is called totally monotone. No two elements of \( M \) are equal.

(a) Describe and analyze an algorithm to solve the following problem in \( O(n) \) time: Given indices \( i, j, i', j' \) as input, compute the number of elements of \( M \) smaller than \( M[i,j] \) and larger than \( M[i',j'] \).

(b) Describe and analyze an algorithm to solve the following problem in \( O(n) \) time: Given indices \( i, j, i', j' \) as input, return an element of \( M \) chosen uniformly at random from the elements smaller than \( M[i,j] \) and larger than \( M[i',j'] \). Assume the requested range is always non-empty.

(c) Describe and analyze a randomized algorithm to compute the median element of \( M \) in \( O(n \log n) \) expected time.

15. Suppose we have a circular linked list of numbers, implemented as a pair of arrays, one storing the actual numbers and the other storing successor pointers. Specifically, let \( X[1..n] \) be an array of \( n \) distinct real numbers, and let \( N[1..n] \) be an array of indices with the following property: If \( X[i] \) is the largest element of \( X \), then \( X[N[i]] \) is the smallest element of \( X \); otherwise, \( X[N[i]] \) is the smallest element of \( X \). For example:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X[i] )</td>
<td>83</td>
<td>54</td>
<td>16</td>
<td>31</td>
<td>45</td>
<td>99</td>
<td>78</td>
<td>62</td>
<td>27</td>
</tr>
<tr>
<td>( N[i] )</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Describe and analyze a randomized algorithm that determines whether a given number \( x \) appears in the array \( X \) in \( O(\sqrt{n}) \) expected time. Your algorithm may not modify the arrays \( X \) and \( N \).

16. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with \( 4^n \) leaves, each colored either black or white. There is a token at the root of the tree. To play the game, you and Death will
take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, you die; if it’s white, you will live forever. You move first, so Death gets the last turn.

You can decide whether it’s worth playing or not as follows. Imagine that the nodes at even levels (where it’s your turn) are Or gates, the nodes at odd levels (where it’s Death’s turn) are And gates. Each gate gets its input from its children and passes its output to its parent. White and black stand for True and False. If the output at the top of the tree is True, then you can win and live forever! If the output at the top of the tree is False, you should challenge Death to a game of Twister instead.

(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. [Hint: This is easy!]

(b) Unfortunately, Death won’t give you enough time to look at every node in the tree. Describe a randomized algorithm that determines whether you can win in $O(3^n)$ expected time. [Hint: Consider the case $n = 1$.]

*(c) Describe and analyze a randomized algorithm that determines whether you can win in $O(c^n)$ expected time, for some constant $c < 3$. [Hint: You may not need to change your algorithm from part (b) at all!]*

17. A majority tree is a complete binary tree with depth $n$, where every leaf is labeled either 0 or 1. The value of a leaf is its label; the value of any internal node is the majority of the values of its three children. Consider the problem of computing the value of the root of a majority tree, given the sequence of $3^n$ leaf labels as input. For example, if $n = 2$ and the leaves are labeled 1, 0, 0, 0, 1, 0, 1, 1, 1, the root has value 0.

(a) Prove that any deterministic algorithm that computes the value of the root of a majority tree must examine every leaf. [Hint: Consider the special case $n = 1$. Recurse.]

(b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. [Hint: Consider the special case $n = 1$. Recurse.]
I thought the following four rules would be enough, provided that I made a firm and constant resolution not to fail even once in the observance of them. The first was never to accept anything as true if I had not evident knowledge of its being so... The second, to divide each problem I examined into as many parts as was feasible, and as was requisite for its better solution. The third, to direct my thoughts in an orderly way... establishing an order in thought even when the objects had no natural priority one to another. And the last, to make throughout such complete enumerations and such general surveys that I might be sure of leaving nothing out.

— René Descartes, Discours de la Méthode (1637)

What is luck?
Luck is probability taken personally.
It is the excitement of bad math.


10 Randomized Binary Search Trees

In this lecture, we consider two randomized alternatives to balanced binary search tree structures such as AVL trees, red-black trees, B-trees, or splay trees, which are arguably simpler than any of these deterministic structures.

10.1 Treaps

10.1.1 Definitions

A treap is a binary tree in which every node has both a search key and a priority, where the inorder sequence of search keys is sorted and each node's priority is smaller than the priorities of its children.¹ In other words, a treap is simultaneously a binary search tree for the search keys and a (min-)heap for the priorities. In our examples, we will use letters for the search keys and numbers for the priorities.

A treap. Letters are search keys; numbers are priorities.

I’ll assume from now on that all the keys and priorities are distinct. Under this assumption, we can easily prove by induction that the structure of a treap is completely determined by the search keys and priorities of its nodes. Since it’s a heap, the node $v$ with highest priority must be the root. Since it’s also a binary search tree, any node $u$ with $\text{key}(u) < \text{key}(v)$ must be in the left

¹Sometimes I hate English. Normally, ‘higher priority’ means ‘more important’, but ‘first priority’ is also more important than ‘second priority’. Maybe ‘posteriority’ would be better; one student suggested ‘unimportance’.
subtree, and any node $w$ with $key(w) > key(v)$ must be in the right subtree. Finally, since the subtrees are treaps, by induction, their structures are completely determined. The base case is the trivial empty treap.

Another way to describe the structure is that a treap is exactly the binary search tree that results by inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm. This characterization is also easy to prove by induction.

A third description interprets the keys and priorities as the coordinates of a set of points in the plane. The root corresponds to a $T$ whose joint lies on the topmost point. The $T$ splits the plane into three parts. The top part is (by definition) empty; the left and right parts are split recursively. This interpretation has some interesting applications in computational geometry, which (unfortunately) we won’t have time to talk about.

A geometric interpretation of the same treap.

Treaps were first discovered by Jean Vuillemin in 1980, but he called them Cartesian trees. 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. Treaps were rediscovered and used to build randomized search trees by Cecilia Aragon and Raimund Seidel in 1989. 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.

### 10.1.2 Treap Operations

The search algorithm is the usual one for binary search trees. The time for a successful search is proportional to the depth of the node. The time for an unsuccessful search is proportional to the depth of either its successor or its predecessor.

To insert a new node $z$, we start by using the standard binary search tree insertion algorithm to insert it at the bottom of the tree. At the point, the search keys still form a search tree, but the priorities may no longer form a heap. To fix the heap property, as long as $z$ has smaller priority than its parent, perform a rotation at $z$, a local operation that decreases the depth of $z$ by one

---

5C. Martínez and S. Roura. Randomized binary search trees. *J. ACM* 45(2):288-323, 1998. The results in this paper are virtually identical (including the constant factors!) to the corresponding results for treaps, although the analysis techniques are quite different.
and increases its parent’s depth by one, while maintaining the search tree property. Rotations can be performed in constant time, since they only involve simple pointer manipulation.

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

The overall time to insert z is proportional to the depth of z before the rotations—we have to walk down the treap to insert z, and then walk back up the treap doing rotations. Another way to say this is that the time to insert z is roughly twice the time to perform an unsuccessful search for key(z).

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) = π and priority(z) = −∞. 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 \(O(\text{depth}(v))\) 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 \(O(\text{depth}(v^+))\) or \(O(\text{depth}(v^-))\).
• **Insert/Delete**: Inserting a new node with key $k$ takes either $O(\text{depth}(v^+))$ time or $O(\text{depth}(v^-))$ 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 $O(\text{depth}(v^+))$ time or $O(\text{depth}(v^-))$ time, since it costs the same as inserting a new dummy root with search key $k$ and priority $-\infty$. Merging is just splitting in reverse.

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

### 10.1.3 Random Priorities

A randomized treap is a treap in which the priorities are independently and uniformly distributed continuous random variables. That means that whenever we insert a new search key into the treap, we generate a random real number between (say) 0 and 1 and use that number as the priority of the new node. The only reason we're using real numbers is so that the probability of two nodes having the same priority is zero, since equal priorities make the analysis slightly messier. In practice, we could just choose random integers from a large range, like 0 to $2^{31} - 1$, or random floating point numbers. Also, since the priorities are independent, each node is equally likely to have the smallest priority.

The cost of all the operations we discussed—search, insert, delete, split, join—is proportional to the depth of some node in the tree. Here we'll see that the expected depth of any node is $O(\log n)$, which implies that the expected running time for any of those operations is also $O(\log n)$.

Let $x_k$ denote the node with the $k$th smallest search key. To simplify notation, let us write $i \uparrow k$ (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:

$$\text{depth}(x_k) = \sum_{i=1}^{n} [i \uparrow k].$$

(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.

$$E[\text{depth}(x_k)] = \sum_{i=1}^{n} E[i \uparrow k] = \sum_{i=1}^{n} \Pr[i \uparrow k]$$

(Just as in our analysis of matching nuts and bolts, we're using linearity of expectation and the fact that $E[X] = \Pr[X = 1]$ 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, 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 \( j > k \), 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:

\[
\Pr[i \uparrow k] = \frac{[i \neq k]}{|k - i| + 1} = \begin{cases} 
\frac{1}{k - i + 1} & \text{if } i < k \\
0 & \text{if } i = k \\
\frac{1}{i - k + 1} & \text{if } i > k 
\end{cases}
\]

To compute the expected depth of a node \( x_k \), we just plug this probability into our formula and grind through the algebra.

\[
E[\text{depth}(x_k)] = \sum_{i=1}^{n} \Pr[i \uparrow k] = \sum_{i=1}^{k-1} \frac{1}{k - i + 1} + \sum_{i=k+1}^{n} \frac{1}{i - k + 1} \\
= \sum_{j=2}^{k} \frac{1}{j} + \sum_{i=2}^{n-k+1} \frac{1}{j} \\
= H_k - 1 + H_{n-k+1} - 1 \\
< \ln k + \ln(n - k + 1) - 2 \\
< 2 \ln n - 2.
\]

In conclusion, every search, insertion, deletion, split, and join operation in an \( n \)-node randomized binary search tree takes \( O(\log n) \) expected time.

Since a treap is exactly the binary tree that results when you insert the keys in order of increasing priority, a randomized treap is the result of inserting the keys in random order. So our analysis also automatically gives us the expected depth of any node in a binary tree built by random insertions (without using priorities).

10.1.4 Randomized Quicksort (Again!)

We’ve already seen two completely different ways of describing randomized quicksort. The first is the familiar recursive one: choose a random pivot, partition, and recurse. The second is a less familiar iterative version: repeatedly choose a new random pivot, partition whatever subset contains it, and continue. But there’s a third way to describe randomized quicksort, this time in terms of binary search trees.

\[ ^6 \text{See Larry Niven and Jerry Pournelle, } \textit{The Gripping Hand}, \textit{Pocket Books}, 1994. \]
Our treap analysis tells us 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 $i$th largest and $k$th largest elements is exactly $2/((k-i)+1)$. The binary tree version of quicksort compares $x_i$ and $x_k$ if and only if $i \uparrow k$ or $k \uparrow i$, so the probabilities are exactly the same.

### 10.2 Skip Lists

Skip lists, which were first discovered by Bill Pugh in the late 1980’s, have many of the usual desirable properties of balanced binary search trees, but their structure is very different.

At a high level, a skip list is just a sorted linked list with some random shortcuts. To do a search in a normal singly-linked list of length $n$, we obviously need to look at $n$ items in the worst case. To speed up this process, we can make a second-level list that contains roughly half the items from the original list. Specifically, for each item in the original list, we duplicate it with probability $1/2$. We then string together all the duplicates into a second sorted linked list, and add a pointer from each duplicate back to its original. Just to be safe, we also add sentinel nodes at the beginning and end of both lists.

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

---

phase has a duplicate. The probability that any node is followed by \( k \) nodes without duplicates is
\( 2^{-k} \), so the expected number of nodes examined in the second phase is at most
\( 1 + \sum_{k \geq 0} 2^{-k} = 2 \).
Thus, by adding these random shortcuts, we’ve reduced the cost of a search from \( n \) to \( n/2 + 2 \),
roughly a factor of two in savings.

### 10.2.1 Recursive Random Shortcuts

Now there’s an obvious improvement—add shortcuts to the shortcuts, and repeat recursively.

That’s exactly how skip lists are constructed. For each node in the original list, we flip a coin
over and over until we get tails. Each time we get heads, we make a duplicate of the node.
The duplicates are stacked up in levels, and the nodes on each level are strung together into sorted
linked lists. Each node \( v \) stores a search key \( \text{key}(v) \), a pointer to its next lower copy \( \text{down}(v) \),
and a pointer to the next node in its level \( \text{right}(v) \).

![A skip list is a linked list with recursive random shortcuts.](image)

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.

\[
\text{SkipListFind}(x, L):
\]

\[
\begin{align*}
v & \leftarrow L \\
\text{while} \ (v \neq \text{null} \text{ and } \text{key}(v) \neq x) \\
& \quad \text{if } \text{key(right}(v)) > x \\
& \quad \quad v \leftarrow \text{down}(v) \\
& \quad \text{else} \\
& \quad \quad v \leftarrow \text{right}(v) \\
\text{return } v
\end{align*}
\]

Intuitively, since each level of the skip lists has about half the number of nodes as the previous
level, the total number of levels should be about \( O(\log n) \). Similarly, each time we add another
level of random shortcuts to the skip list, we cut the search time roughly in half, except for
a constant overhead, so after \( O(\log n) \) levels, we should have a search time of \( O(\log n) \). Let’s
formalize each of these two intuitive observations.
10.2.2 Number of Levels

The actual values of the search keys don’t affect the skip list analysis, so let’s assume the keys are the integers 1 through \( n \). Let \( L(x) \) be the number of levels of the skip list that contain some search key \( x \), not counting the bottom level. Each new copy of \( x \) is created with probability \( \frac{1}{2} \) from the previous level, essentially by flipping a coin. We can compute the expected value of \( L(x) \) recursively—with probability \( \frac{1}{2} \), we flip tails and \( L(x) = 0 \); and with probability \( \frac{1}{2} \), we flip heads, increase \( L(x) \) by one, and recurse:

\[
E[L(x)] = \frac{1}{2} \cdot 0 + \frac{1}{2} \left( 1 + E[L(x)] \right)
\]

Solving this equation gives us \( E[L(x)] = 1 \).

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 \( \ell \), it must have flipped \( \ell \) heads in a row when it was inserted, so \( \Pr[L(x) \geq \ell] = 2^{-\ell} \). The skip list has at least \( \ell \) levels if and only if \( L(x) \geq \ell \) for at least one of the \( n \) search keys.

\[
\Pr[L \geq \ell] = \Pr[(L(1) \geq \ell) \lor (L(2) \geq \ell) \lor \cdots \lor (L(n) \geq \ell)]
\]

Using the union bound — \( \Pr[A \lor B] \leq \Pr[A] + \Pr[B] \) 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 \( \ell \leq \log n \), this bound is trivial. However, for any constant \( c > 1 \), we have a strong upper bound

\[
\Pr[L \geq c \log n] \leq \frac{1}{n^{c-1}}.
\]

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:

\[ E[L] = \sum_{\ell \geq 0} \ell \cdot \Pr[L = \ell] = \sum_{\ell \geq 1} \Pr[L \geq \ell] \]

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:

\[
E[L(x)] = \sum_{\ell \geq 1} \Pr[L \geq \ell] = \sum_{\ell = 1}^{\log n} \Pr[L \geq \ell] + \sum_{\ell \geq \log n + 1} \Pr[L \geq \ell] \\
\leq \sum_{\ell = 1}^{\log n} 1 + \sum_{\ell \geq \log n + 1} \frac{n}{2^\ell} \\
= \log n + \sum_{i \geq 1} \frac{1}{2^i} [i = \ell - \log n] \\
= \log n + 2
\]

So in expectation, a skip list has at most two more levels than an ideal version where each level contains exactly half the nodes of the next level below.

### 10.2.3 Logarithmic Search Time

It’s a little easier to analyze the cost of a search if we imagine running the algorithm backwards. \texttt{SkiplistFind} takes the output from \texttt{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 u’ or ‘(v)\rightarrow l’.

```
SkiplistFind(v):
while (v \neq L)
    if up(v) exists
        v \leftarrow up(v)
    else
        v \leftarrow left(v)
```

Now for every node \( v \) in the skip list, \( up(v) \) exists with probability 1/2. So for purposes of analysis, \texttt{SkiplistFind} is equivalent to the following algorithm:

```
FlipWalk(v):
while (v \neq L)
    if CoinFlip = Heads
        v \leftarrow up(v)
    else
        v \leftarrow left(v)
```

Obviously, the expected number of heads is exactly the same as the expected number of \texttt{Tails}. Thus, the expected running time of this algorithm is twice the expected number of upward jumps. Since we already know that the number of upward jumps is \( O(\log n) \) with high probability, we can conclude that the worst-case search time is \( O(\log n) \) with high probability (and therefore in expectation).

---

8 Leonardo da Vinci wrote all his notes using mirror-writing, but not because he wanted to keep his discoveries secret. He just had really bad arthritis in his right hand!
Exercises

1. Prove that a treap is exactly the binary search tree that results from inserting the nodes one at a time into an initially empty tree, in order of increasing priority, using the standard textbook insertion algorithm.

2. Consider a treap $T$ with $n$ vertices. As in the notes, identify nodes in $T$ by the ranks of their search keys; thus, ‘node 5’ means the node with the 5th smallest search key. Let $i$, $j$, and $k$ be integers such that $1 \leq i \leq j \leq k \leq n$.

   (a) The left spine of a binary tree is a path starting at the root and following only left-child pointers down to a leaf. What is the expected number of nodes in the left spine of $T$?
   (b) What is the expected number of leaves in $T$? [Hint: What is the probability that node $k$ is a leaf?]
   (c) What is the expected number of nodes in $T$ with two children?
   (d) What is the expected number of nodes in $T$ with exactly one child?
   *(e) What is the expected number of nodes in $T$ with exactly one grandchild?*
   (f) Prove that the expected number of proper descendants of any node in a treap is exactly equal to the expected depth of that node.
   (g) What is the exact probability that node $j$ is a common ancestor of node $i$ and node $k$? [Hint: Don’t use part (a)!]
   (h) What is the exact expected length of the unique path from node $i$ to node $k$ in $T$?

3. Recall that a priority search tree is a binary tree in which every node has both a search key and a priority, arranged so that the tree is simultaneously a binary search tree for the keys and a min-heap for the priorities. A heater is a priority search tree in which the priorities are given by the user, and the search keys are distributed uniformly and independently at random in the real interval $[0, 1]$. Intuitively, a heater is a sort of anti-treap.⁹

   The following problems consider an $n$-node heater $T$ whose priorities are the integers from 1 to $n$. We identify nodes in $T$ by their priorities; thus, ‘node 5’ means the node in $T$ with priority 5. For example, the min-heap property implies that node 1 is the root of $T$. Finally, let $i$ and $j$ be integers with $1 \leq i < j \leq n$.

   (a) Prove that in a random permutation of the $(i + 1)$-element set $\{1, 2, \ldots, i, j\}$, elements $i$ and $j$ are adjacent with probability $2/(i + 1)$.
   (b) Prove that node $i$ is an ancestor of node $j$ with probability $2/(i + 1)$. [Hint: Use part (a)!]
   (c) What is the probability that node $i$ is a descendant of node $j$? [Hint: Don’t use part (a)!]
   (d) What is the exact expected depth of node $j$?
   (e) Describe and analyze an algorithm to insert a new item into a heater. Express the expected running time of the algorithm in terms of the rank of the newly inserted item.

---

⁹There are those who think that life has nothing left to chance, a host of holy horrors to direct our aimless dance.
(f) Describe an algorithm to delete the minimum-priority item (the root) from an \( n \)-node heater. What is the expected running time of your algorithm?

*4. In the usual theoretical presentation of treaps, the priorities are random real numbers chosen uniformly from the interval \([0, 1]\). In practice, however, computers have access only to random bits. This problem asks you to analyze an implementation of treaps that takes this limitation into account.

Suppose the priority of a node \( v \) is abstractly represented as an infinite sequence \( \pi_v[1..\infty] \) of random bits, which is interpreted as the rational number

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

However, only a finite number \( \ell_v \) 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:

```
LARGER_PRIORITY(v, w):
for i ← 1 to ∞
    if i > ℓ_v
        ℓ_v ← i; \( \pi_v[i] \) ← RANDOM_BIT
    if i > ℓ_w
        ℓ_w ← i; \( \pi_w[i] \) ← RANDOM_BIT
    if \( \pi_v[i] > \pi_w[i] \)
        return v
    else if \( \pi_v[i] < \pi_w[i] \)
        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 LARGER_PRIORITY during these insertions.

(a) Prove that \( E[L] = \Theta(n) \).
(b) Prove that \( E[\ell_v] = \Theta(1) \) for any node \( v \). [Hint: This is equivalent to part (a). Why?]
(c) Prove that \( E[\ell_{\text{root}}] = \Theta(\log n) \). [Hint: Why doesn’t this contradict part (b)?]

5. Prove the following basic facts about skip lists, where \( n \) is the number of keys.

(a) The expected number of nodes is \( O(n) \).
(b) A new key can be inserted in \( O(\log n) \) time with high probability.
(c) A key can be deleted in \( O(\log n) \) time with high probability.

6. Suppose we are given two skip lists, one storing a set \( A \) of \( m \) keys the other storing a set \( B \) of \( n \) keys. Describe and analyze an algorithm to merge these into a single skip list storing the set \( A \cup B \) in \( O(n) \) expected time. Here we do not assume that every key in \( A \) is smaller than every key in \( B \); the two sets maybe arbitrarily intermixed. [Hint: Do the obvious thing.]
Any skip list $L$ can be transformed into a binary search tree $T(L)$ as follows. The root of $T(L)$ is the leftmost node on the highest non-empty level of $L$; 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 $T(L)$ a skip list tree.

(a) Show that any search in $T(L)$ is no more expensive than the corresponding search in $L$. (Searching in $T(L)$ could be considerably cheaper—why?)

(b) Describe an algorithm to insert a new search key into a skip list tree in $O(\log n)$ expected time. Inserting key $x$ into $T(L)$ should produce exactly the same tree as inserting $x$ into $L$ and then transforming $L$ into a tree. [Hint: You will need to maintain some additional information in the tree nodes.]

(c) Describe an algorithm to delete a search key from a skip list tree in $O(\log n)$ expected time. Again, deleting key $x$ from $T(L)$ should produce exactly the same tree as deleting $x$ from $L$ and then transforming $L$ into a tree.

8. A meldable priority queue stores a set of keys from some totally-ordered universe (such as the integers) and supports the following operations:

- **MAKEQUEUE**: Return a new priority queue containing the empty set.
- **FINDMIN(Q)**: Return the smallest element of $Q$ (if any).
- **DELETEMIN(Q)**: Remove the smallest element in $Q$ (if any).
- **INSERT(Q, x)**: Insert element $x$ into $Q$, if it is not already there.
- **DECREASEKEY(Q, x, y)**: Replace an element $x \in Q$ with a smaller key $y$. (If $y > x$, the operation fails.) The input is a pointer directly to the node in $Q$ containing $x$.
- **DELETE(Q, x)**: Delete the element $x \in Q$. The input is a pointer directly to the node in $Q$ containing $x$.
- **MELD(Q1, Q2)**: 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(Q1, Q2)** is $O(\log n)$, where $n = |Q_1| + |Q_2|$. [Hint: How long is a random root-to-leaf path in an $n$-node binary tree if each left/right choice is made with equal probability?]
(b) Prove that MELD\((Q_1, Q_2)\) runs in \(O(\log n)\) time with high probability.

(c) Show that each of the other meldable priority queue operations can be implemented with at most one call to MELD and \(O(1)\) additional time. (This implies that every operation takes \(O(\log n)\) time with high probability.)
But, on the other hand, Uncle Abner said that the person that had took a bull by the tail once had learnt sixty or seventy times as much as a person that hadn’t, and said a person that started in to carry a cat home by the tail was getting knowledge that was always going to be useful to him, and warn’t ever going to grow dim or doubtful.

— Mark Twain, *Tom Sawyer Abroad* (1894)

**11 Tail Inequalities**

The simple recursive structure of skip lists made it relatively easy to derive an upper bound on the expected worst-case search time, by way of a stronger high-probability upper bound on the worst-case search time. We can prove similar results for treaps, but because of the more complex recursive structure, we need slightly more sophisticated probabilistic tools. These tools are usually called tail inequalities; intuitively, they bound the probability that a random variable with a bell-shaped distribution takes a value in the tails of the distribution, far away from the mean.

### 11.1 Markov’s Inequality

Perhaps the simplest tail inequality was named after the Russian mathematician Andrey Markov; however, in strict accordance with Stigler’s Law of Eponymy, it first appeared in the works of Markov’s probability teacher, Pafnuty Chebyshev.¹

**Markov’s Inequality.** Let $X$ be a non-negative integer random variable. For any $t > 0$, we have $\Pr[X \geq t] \leq E[X]/t$.

**Proof:** The inequality follows from the definition of expectation by simple algebraic manipulation.

\[
E[X] = \sum_{k=0}^{\infty} k \cdot \Pr[X = k] \quad \text{[definition of $E[X]$]}
\]

\[
= \sum_{k=0}^{\infty} \Pr[X \geq k] \quad \text{[algebra]}
\]

\[
\geq \sum_{k=0}^{t-1} \Pr[X \geq k] \quad \text{[since $t < \infty$]}
\]

\[
\geq \sum_{k=0}^{t-1} \Pr[X \geq t] \quad \text{[since $k < t$]}
\]

\[
= t \cdot \Pr[X \geq t] \quad \text{[algebra]}
\]

Unfortunately, the bounds that Markov’s inequality implies (at least directly) are often very weak, even useless. (For example, Markov’s inequality implies that with high probability, every node in an $n$-node treap has depth $O(n^2 \log n)$. Well, *duh!* To get stronger bounds, we need to exploit some additional structure in our random variables.

¹The closely related tail bound traditionally called Chebyshev’s inequality was actually discovered by the French statistician Irénée-Jules Bienaymé, a friend and colleague of Chebyshev’s.
11.2 Independence

A set of random variables $X_1, X_2, \ldots, X_n$ are said to be mutually independent if and only if

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

for all possible values $x_1, x_2, \ldots, x_n$. For examples, different flips of the same fair coin are mutually independent, but the number of heads and the number of tails in a sequence of $n$ coin flips are not independent (since they must add to $n$). Mutual independence of the $X_i$'s implies that the expectation of the product of the $X_i$'s is equal to the product of the expectations:

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

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

— Discuss limited independence? —
— Add Chebychev and other moment inequalities? —

11.3 Chernoff Bounds

— Replace with Mihai’s exponential-moment derivation! —

Suppose $X = \sum_{i=1}^{n} X_i$ is the sum of $n$ mutually independent random indicator variables $X_i$. For each $i$, let $p_i = \Pr[X_i = 1]$, and let $\mu = E[X] = \sum_i E[X_i] = \sum_i p_i$.

Chernoff Bound (Upper Tail).  \[ \Pr[X > (1 + \delta)\mu] < \left( \frac{e^\delta}{(1 + \delta)^{1+\delta}} \right)^\mu \text{ for any } \delta > 0. \]

Proof: The proof is fairly long, but it relies on just a few basic components: a clever substitution, Markov’s inequality, the independence of the $X_i$’s, The World’s Most Useful Inequality $e^x > 1 + x$, a tiny bit of calculus, and lots of high-school algebra.

We start by introducing a variable $t$, whose role will become clear shortly.

$$Pr[X > (1 + \delta)\mu] = \Pr[e^{tX} > e^{(1+\delta)t\mu}]$$

To cut down on the superscripts, I’ll usually write $\exp(x)$ instead of $e^x$ in the rest of the proof. Now apply Markov’s inequality to the right side of this equation:

$$Pr[X > (1 + \delta)\mu] < \frac{E[\exp(tX)]}{\exp(t(1 + \delta)\mu)}.$$

We can simplify the expectation on the right using the fact that the terms $X_i$ are independent.

$$E[\exp(tX)] = E \left[ \exp \left( t \sum_i X_i \right) \right] = E \left[ \prod_i \exp(tX_i) \right] = \prod_i E[\exp(tX_i)]$$
We can bound the individual expectations $E[\exp(tX_i)]$ using The World’s Most Useful Inequality:

$$E[\exp(tX_i)] = p_i e^t + (1 - p_i) = 1 + (e^t - 1) p_i < \exp((e^t - 1)p_i)$$

This inequality gives us a simple upper bound for $E[\exp(tX)]$:

$$E[\exp(tX)] \leq \prod_i \exp((e^t - 1)p_i) < \exp((e^t - 1)\mu)$$

Substituting this back into our original fraction from Markov’s inequality, we obtain

$$Pr[X > (1 + \delta)\mu] < \frac{E[\exp(tX)]}{\exp(t(1 + \delta)\mu)} < \frac{\exp((e^t - 1)\mu)}{\exp(t(1 + \delta)\mu)} = \left(\frac{\exp(e^t - 1 - t(1 + \delta))}{\mu}\right)^\mu$$

Notice that this last inequality holds for all possible values of $t$. To obtain the final tail bound, we will choose $t$ to make this bound as small as possible. To minimize $e^t - 1 - t(1 + \delta)$, we take its derivative with respect to $t$ and set it to zero:

$$\frac{d}{dt} (e^t - 1 - t(1 + \delta)) = e^t - 1 - \delta = 0.$$ 

(And you thought calculus would never be useful!) This equation has just one solution $t = \ln(1+\delta)$. Plugging this back into our bound gives us

$$Pr[X > (1 + \delta)\mu] < \left(\frac{\exp(\delta - (1 + \delta)\ln(1 + \delta))}{\mu}\right)^\mu$$

And we’re done! □

This form of the Chernoff bound can be a bit clumsy to use. A more complicated argument gives us the bound

$$Pr[X > (1 + \delta)\mu] < e^{-\mu\delta^2/3}$$

for any $0 < \delta < 1$.

A similar argument gives us an inequality bounding the probability that $X$ is significantly smaller than its expected value:

**Chernoff Bound (Lower Tail).** $Pr[X < (1 - \delta)\mu] < e^{-\mu\delta^2/2}$ for any $\delta > 0$.

### 11.4 Back to Treaps

In our analysis of randomized treaps, we wrote $i \uparrow k$ to indicate that the node with the $i$th smallest key (‘node $i$’) was a proper ancestor of the node with the $k$th 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

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

To prove a worst-case expected bound on the depth of the tree, we need to argue that the maximum depth of any node is small. Chernoff bounds make this argument easy, once we establish that the relevant indicator variables are mutually independent.
Lemma 1. For any index k, the k−1 random variables \([i \uparrow k]\) with \(i < k\) are mutually independent. Similarly, for any index k, the \(n−k\) random variables \([i \uparrow k]\) with \(i > k\) are mutually independent.

Proof: We explicitly consider only the first half of the lemma when \(k = 1\), although the argument generalizes easily to other values of \(k\). To simplify notation, let \(X_i\) denote the indicator variable \([i \uparrow 1]\). Fix \(n−1\) arbitrary indicator values \(x_2, x_3, \ldots, x_n\). We prove the lemma by induction on \(n\), with the vacuous base case \(n = 1\). The definition of conditional probability gives us

\[
\Pr \left[ \bigwedge_{i=2}^{n} (X_i = x_i) \right] = \Pr \left[ \bigwedge_{i=2}^{n-1} (X_i = x_i) \land X_n = x_n \right]
\]

\[
= \Pr \left[ \bigwedge_{i=2}^{n-1} (X_i = x_i) \right] \cdot \Pr [X_n = x_n]
\]

Now recall that \(X_n = 1\) (which means \(1 \uparrow n\)) if and only if node \(n\) has the smallest priority of all nodes. The other \(n−2\) indicator variables \(X_i\) depend only on the order of the priorities of nodes 1 through \(n−1\). There are exactly \((n−1)!\) permutations of the \(n\) priorities in which the \(n\)th priority is smallest, and each of these permutations is equally likely. Thus,

\[
\Pr \left[ \bigwedge_{i=2}^{n-1} (X_i = x_i) \right] = \frac{1}{(n−1)!} \] \(\Pr [X_n = x_n]\).
\]

The inductive hypothesis implies that the variables \(X_2, \ldots, X_{n−1}\) are mutually independent, so

\[
\Pr \left[ \bigwedge_{i=2}^{n-1} (X_i = x_i) \right] \cdot \prod_{i=2}^{n−1} \Pr [X_i = x_i] = \prod_{i=1}^{n−1} \Pr [X_i = x_i],
\]

or in other words, that the indicator variables are mutually independent. □

Theorem 2. The depth of a randomized treap with \(n\) nodes is \(O(\log n)\) with high probability.

Proof: First let’s bound the probability that the depth of node \(k\) is at most \(8 \ln n\). There’s nothing special about the constant 8 here; I’m being generous to make the analysis easier.

The depth is a sum of \(n\) indicator variables \(A_k[i]\), as \(i\) ranges from 1 to \(n\). Our Observation allows us to partition these variables into two mutually independent subsets. Let \(d_<(k) = \sum_{i < k}[i \uparrow k]\) and \(d_>(k) = \sum_{i > k}[i \uparrow k]\), so that \(\text{depth}(k) = d_<(k) + d_>(k)\). If \(\text{depth}(k) > 8 \ln n\), then either \(d_<(k) > 4 \ln n\) or \(d_>(k) > 4 \ln n\).

Chernoff’s inequality, with \(\mu = \mathbb{E}[d_<(k)] = H_k - 1 < \ln n\) and \(\delta = 3\), bounds the probability that \(d_<(k) > 4 \ln n\) as follows.

\[
\Pr[d_<(k) > 4 \ln n] < \Pr[d_<(k) > 4 \mu] < \left( \frac{e^3}{4^\delta} \right)^{\frac{\ln n}{\mu}} = \left( \frac{e^3}{4^3} \right)^{\ln n} = n^{\ln(e^3/4^3)} = n^{3-4\ln 4} < \frac{1}{n^2}.
\]

(The last step uses the fact that \(4 \ln 4 \approx 5.54518 > 5\).) The same analysis implies that \(\Pr[d_>(k) > 4 \ln n] < 1/n^2\). These inequalities imply the crude bound \(\Pr[\text{depth}(k) > 4 \ln n] < 2/n^2\).
Now consider the probability that the treap has depth greater than $10 \ln n$. Even though the distributions of different nodes' depths are not independent, we can conservatively bound the probability of failure as follows:

$$
\Pr\left[ \max_k \text{depth}(k) > 8 \ln n \right] = \Pr\left[ \bigwedge_{k=1}^n (\text{depth}(k) > 8 \ln n) \right] \leq \sum_{k=1}^n \Pr[\text{depth}(k) > 8 \ln n] < \frac{2}{n}.
$$

This argument implies more generally that for any constant $c$, the depth of the treap is greater than $c \ln n$ with probability at most $2/n^{c \ln c - c}$. We can make the failure probability an arbitrarily small polynomial by choosing $c$ appropriately. \qed

This lemma implies that any search, insertion, deletion, or merge operation on an $n$-node treap requires $O(\log n)$ time with high probability. In particular, the expected worst-case time for each of these operations is $O(\log n)$.

**Exercises**

1. Prove that for any integer $k$ such that $1 < k < n$, the $n - 1$ indicator variables $[i \uparrow k]$ with $i \neq k$ are not mutually independent. [Hint: Consider the case $n = 3$.]

2. Recall from Exercise 1 in the previous note that the expected number of descendants of any node in a treap is $O(\log n)$. Why doesn't the Chernoff-bound argument for depth imply that, with high probability, every node in a treap has $O(\log n)$ descendants? The conclusion is clearly bogus—Every treap has a node with $n$ descendants!—but what's the hole in the argument?

3. Recall from the previous lecture note that a heater is a sort of anti-treap, in which the priorities of the nodes are given, but their search keys are generated independently and uniformly from the unit interval $[0, 1]$.

   Prove that an $n$-node heater has depth $O(\log n)$ with high probability.
Insanity is repeating the same mistakes and expecting different results.

Calvin: There! I finished our secret code!

Hobbes: Let’s see.

Calvin: I assigned each letter a totally random number, so the code will be hard

Hobbes: That’s a good code all right.

Calvin: Now we just commit this to memory.

Calvin: Did you finish your map of our neighborhood?

Hobbes: Not yet. How many bricks does the front walk have?

— Bill Watterson, “Calvin and Hobbes” (August 23, 1990)

```c
int getRandomNumber() {
    return 4; // chosen by fair dice roll.
    // guaranteed to be random.
}
```

[RFC 1149.5 specifies 4 as the standard IEEE-vetted random number.]

— Randall Munroe, xkcd (http://xkcd.com/221/)
Reproduced under a Creative Commons Attribution-NonCommercial 2.5 License

12 Hash Tables

12.1 Introduction

A hash table is a data structure for storing a set of items, so that we can quickly determine
whether an item is or is not in the set. The basic idea is to pick a hash function \( h \) that maps every
possible item \( x \) to a small integer \( h(x) \). Then we store \( x \) in slot \( h(x) \) in an array. The array is the
hash table.

Let’s be a little more specific. We want to store a set of \( n \) items. Each item is an element of
a fixed set \( \mathbb{U} \) called the universe; we use \( u \) to denote the size of the universe, which is just the
number of items in \( \mathbb{U} \). 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

\[
h: \mathbb{U} \rightarrow \{0, 1, \ldots, m - 1\},
\]

mapping each possible item in \( \mathbb{U} \) to a slot in the hash table. We say that an item \( x \) hashes to the
slot \( T[h(x)] \).

Of course, if \( u = m \), then we can always just use the trivial hash function \( h(x) = x \); in other
words, we can use the item itself as the index into the table. This is called a direct access table, or
more commonly, an array. In most applications, though, this approach requires much more space
than we can reasonably allocate; on the other hand, we rarely need need to store more than a
tiny fraction of \( \mathbb{U} \). Ideally, the table size \( m \) should be roughly equal to the number \( n \) of items we
actually want to store.

The downside of using a smaller table is that we must deal with collisions. We say that two
items \( x \) and \( y \) collide if their hash values are equal: \( h(x) = h(y) \). We are now left with two
different (but interacting) design decisions. First, how to we choose a hash function \( h \) that can
be evaluated quickly and that keeps the number of collisions as small as possible? Second, when collisions do occur, how do we deal with them?

12.2 The Importance of Being Random

If we already knew the precise data set that would be stored in our hash table, it is possible (but not particularly easy) to find a perfect hash function that avoids collisions entirely. Unfortunately, for most applications of hashing, we don’t know what the user will put into the table. Thus, it is impossible even in principle to devise a perfect hash function in advance; no matter what hash function we choose, some pair of items from \( U \) will collide. Worse, for any fixed hash function, there is a subset of at least \( |U|/m \) items that all hash to the same location. If our input data happens to come from such a subset, either by chance or malicious intent, our code will come to a grinding halt. This is a real security issue with core Internet routers, for example; every router on the Internet backbone survives millions of attacks per day, including timing attacks, from malicious agents.

The only way to provably avoid this worst-case behavior is to choose our hash functions randomly. Specifically, we will fix a set \( \mathcal{H} \) of functions from \( U \) to \( \{0, 1, \ldots, m - 1\} \), and then at run time, we choose our hash function randomly from the set \( \mathcal{H} \) according to some fixed distribution. Different sets \( \mathcal{H} \) and different distributions over that set imply different theoretical guarantees. Screw this into your brain:

**Input data is not random!**

So good hash functions must be random!

In particular, the simple deterministic hash function \( h(x) = x \mod m \), which is often taught and recommended under the name “the division method”, is utterly stupid. Many textbooks correctly observe that this hash function is bad when \( m \) is a power of 2, because then \( h(x) \) is just the low-order bits of \( m \), but then they bizarrely recommend making \( m \) prime to avoid such obvious collisions. But even when \( m \) is prime, any pair of items whose difference is an integer multiple of \( m \) collide with absolute certainty; for all integers \( a \) and \( x \), we have \( h(x + am) = h(x) \). Why would anyone use a hash function where they know certain pairs of keys always collide? Sheesh!

12.3 ...But Not Too Random

Most theoretical analysis of hashing assumes ideal random hash functions. Ideal randomness means that the hash function is chosen uniformly at random from the set of all functions from \( U \) 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:

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

We emphasize that this condition must hold for every item $x \in U$ 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 $\text{const}_a$ denote the constant function $\text{const}_a(x) = a$ for all $x$, and let $\mathcal{K} = \{\text{const}_a \mid 0 \leq a \leq m-1\}$ 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:

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

(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:

$$\text{Near-universal: } \Pr_{h \in \mathcal{K}}[h(x) = h(y)] \leq \frac{2}{m} \text{ for all } x \neq y$$

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 $1/m^k$:

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

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

### 12.4 Chaining

One of the most common methods for resolving collisions in hash tables is called chaining. In a chained hash table, each entry $T[i]$ is not just a single item, but rather (a pointer to) a linked list of all the items that hash to $T[i]$. Let $\ell(x)$ denote the length of the list $T[h(x)]$. To see if

¹Confusingly, universality is often called the uniform hashing assumption, even though it is not an assumption that the hash function is uniform.
an item $x$ is in the hash table, we scan the entire list $T[h(x)]$. The worst-case time required to search for $x$ is $O(1)$ to compute $h(x)$ plus $O(1)$ for every element in $T[h(x)]$, or $O(1 + \ell(x))$ overall. Inserting and deleting $x$ also take $O(1 + \ell(x))$ time.

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

$$C_{x,y} = \left[h(x) = h(y)\right].$$

(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

$$E[C_{x,y}] = \Pr[C_{x,y} = 1] = \begin{cases} 1 & \text{if } x = y \\ 1/m & \text{otherwise} \end{cases}$$

Now we just have to grind through the definitions.

$$E[\ell(x)] = \sum_{y \in T} E[C_{x,y}] = \sum_{y \in T} \frac{1}{m} = \frac{n}{m}$$

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 $\alpha$.

$$\alpha := \frac{n}{m}$$

Similarly, if $h$ is chosen from a near-universal set of hash functions, then $E[\ell(x)] \leq 2\alpha$. 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 $U$ has a total ordering, we can store each chain in a balanced binary search tree. This reduces the expected time for any search to $O(1 + \log \ell(x))$, and under the simple uniform hashing assumption, the expected time for any search is $O(1 + \log \alpha)$.

Another natural possibility is to work recursively! Specifically, for each $T[i]$, we maintain a hash table $T_i$ containing all the items with hash value $i$. Collisions in those secondary tables are
resolved recursively, by storing secondary overflow lists in tertiary hash tables, and so on. The resulting data structure is a tree of hash tables, whose leaves correspond to items that (at some level of the tree) are hashed without any collisions. If every hash table in this tree has size $m$, then the expected time for any search is $O(\log_m n)$. In particular, if we set $m = \sqrt{n}$, the expected time for any search is constant. On the other hand, there is no inherent reason to use the same hash table size everywhere; after all, hash tables deeper in the tree are storing fewer items.

**Caveat Lector!** The preceding analysis does not imply bounds on the expected worst-case search time is constant. The expected worst-case search time is $O(1 + L)$, where $L = \max_x \ell(x)$. Under the uniform hashing assumption, the maximum list size $L$ is very likely to grow faster than any constant, unless the load factor $\alpha$ is significantly smaller than 1. For example, $E[L] = \Theta(\log n / \log \log n)$ when $\alpha = 1$. We’ve stumbled on a powerful but counterintuitive fact about probability: When several individual items are distributed independently and uniformly at random, the resulting distribution is not uniform in the traditional sense! Later in this lecture, I’ll describe how to achieve constant expected worst-case search time using secondary hash tables.

### 12.5 Multiplicative Hashing

Perhaps the simplest technique for near-universal hashing, first described by Carter and Wegman in the 1970s, is called **multiplicative hashing**. I’ll describe two variants of multiplicative hashing, one using modular arithmetic with prime numbers, the other using modular arithmetic with powers of two. In both variants, a hash function is specified by an integer parameter $a$, called a salt. The salt is chosen uniformly at random when the hash table is created and remains fixed for the entire lifetime of the table. All probabilities are defined with respect to the random choice of salt.

For any non-negative integer $n$, let $[n]$ denote the $n$-element set $\{0, 1, \ldots, n-1\}$, and let $[n]^+ = \{1, 2, \ldots, n-1\}$.

#### 12.5.1 Prime multiplicative hashing

The first family of multiplicative hash function is defined in terms of a prime number $p > |[n]|$. For any integer $a \in [p]^+$, define a function $\text{multip}_a : U \to [m]$ by setting

$$\text{multip}_a(x) = (ax \mod p) \mod m$$

and let

$$\mathcal{M}^p := \{\text{multip}_a | a \in [p]^+\}$$

denote the set of all such functions. Here, the integer $a$ is the salt for the hash function $\text{multip}_a$. We claim that this family of hash functions is universal.

The use of prime modular arithmetic is motivated by the fact that division modulo prime numbers is well-defined.

**Lemma 1.** For every integer $z \in [p]^+$, there is a unique integer $a \in [p]^+$ such that $az \mod p = 1$.

**Proof:** Let $z$ be an arbitrary integer in $[p]^+$.

Suppose $az \mod p = bz \mod p$ for some integers $a, b \in [p]^+$. Then $(a - b)z \mod p = 0$, which means $(a - b)z$ is divisible by $p$. Because $p$ is prime, the inequality $1 \le z \le p - 1$ implies that $a - b$ must be divisible by $p$. Similarly, the inequality $2 - p < a - b < p - 2$ implies that $a$ and $b$ must be equal. Thus, for each $z \in [p]^+$, there is at most one $a \in [p]^+$ such that $ax \mod p = z$. 

5
Similarly, suppose $az \mod p = 0$ for some integer $a \in [p]^+$. Then because $p$ is prime, either $a$ or $z$ is divisible by $p$, which is impossible.

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

For any integers $x, y \in \mathbb{Z}$ and any salt $a \in [p]^+$, we have

$$
mult_p(a) - mult_p(a) = (ax \mod p) \mod m - (ay \mod p) \mod m
= (ax \mod p - ay \mod p) \mod m
= ((ax - ay) \mod p) \mod m
= (a(x - y) \mod p) \mod m
= mult_p(x - y).
$$

Thus, we have a collision $mult_p(x) = mult_p(y)$ if and only if $mult_p(x - y) = 0$. Thus, to prove that $MP$ is universal, it suffices to prove the following lemma.

**Lemma 2.** For any $z \in [p]^+$, we have $\Pr_a [mult_p(z) = 0] \leq 1/m$.

**Proof:** Fix an arbitrary integer $z \in [p]^+$. The previous lemma implies that for any integer $1 \leq x \leq p-1$, there is a unique integer $a$ such that $(az \mod p) = x$; specifically, $a = x \cdot z^{-1} \mod p$. There are exactly $\lfloor (p-1)/m \rfloor$ integers $k$ such that $1 \leq km \leq p-1$. Thus, there are exactly $\lfloor (p-1)/m \rfloor$ salts $a$ such that $mult_p(z) = 0$. □

### 12.5.2 Binary multiplicative hashing

A slightly simpler variant of multiplicative hashing that avoids the need for large prime numbers was first analyzed by Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen in 1997. For this variant, we assume that $\mathbb{Z} = [2^w]$ and that $m = 2^\ell$ for some integers $w$ and $\ell$. Thus, our goal is to hash $w$-bit integers (“words”) to $\ell$-bit integers (“labels”).

For any odd integer $a \in [2^w]$, we define the hash function $mult_b : \mathbb{Z} \rightarrow [m]$ as follows:

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

Again, the odd integer $a$ is the salt.
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 \( \text{mult}_a(x) \) consists of the top \( \ell \) bits of the bottom half:

\[
\text{mult}_a(x) := (a \cdot x)[w-1..w-\ell]
\]

Most programming languages automatically perform integer arithmetic modulo some power of two. If we are using an integer type with \( w \) bits, the function \( \text{mult}_a(x) \) can be implemented by a single multiplication followed by a single right-shift. For example, in C:

```c
#define hash(a,x) ((a)*(x) >> (WORDSIZE-HASHBITS))
```

Now we claim that the family \( \mathcal{MB} := \{ \text{mult}_a \mid a \text{ is odd} \} \) of all such functions is near-universal. To prove this claim, we again need to argue that division is well-defined, at least for a large subset of possible words. Let \( W \) denote the set of odd integers in \( [2^w] \).

**Lemma 3.** For any integers \( x,z \in W \), there is exactly one integer \( a \in W \) such that \( ax \mod 2^w = z \).

**Proof:** Fix an integer \( x \in W \). Suppose \( ax \mod 2^w = bx \mod 2^w \) for some integers \( a,b \in W \). Then \( (b-a)x \mod 2^w = 0 \), 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 \( ax \mod 2^w = z \). In other words, the function \( f_x : W \to W \) defined by \( f_x(a) := ax \mod 2^w \) is injective. Every injective function from a finite set to itself is a bijection. \( \square \)

**Lemma 4.** \( \mathcal{MB} \) is near-universal.

**Proof:** Fix two distinct words \( x,y \in \llbracket w \rrbracket \) such that \( x < y \). If \( \text{mult}_a(x) = \text{mult}_a(y) \), then the top \( \ell \) bits of \( a(y-x) \mod 2^w \) are either all 0s (if \( ax \mod 2^w \leq ay \mod 2^w \)) or all 1s (otherwise). Equivalently, if \( \text{mult}_a(x) = \text{mult}_a(y) \), then either \( \text{mult}_a(y-x) = 0 \) or \( \text{mult}_a(y-x) = m-1 \). Thus,

\[
\Pr[\text{mult}_a(x) = \text{mult}_a(y)] \leq \Pr[\text{mult}_a(y-x) = 0] + \Pr[\text{mult}_a(y-x) = m-1].
\]

We separately bound the terms on the right side of this inequality.

Because \( x \neq y \), we can write \( (y-x) \mod 2^w = q2^r \) for some odd integer \( q \) and some integer \( 0 \leq r \leq w-1 \). The previous lemma implies that \( aq \mod 2^w \) consists of \( w-1 \) random bits followed by a 1. Thus, \( aq2^r \mod 2^w \) 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 \( \text{mult}_a(y-x) \) consists of \( \ell \) random bits, so
  \[
  \Pr[\text{mult}_a(y-x) = 0] = \Pr[\text{mult}_a(y-x) = m-1] = 1/2^\ell.
  \]

- If \( r = w-\ell \), then \( \text{mult}_a(y-x) \) consists of \( \ell-1 \) random bits followed by a 1, so
  \[
  \Pr[\text{mult}_a(y-x) = 0] = 0 \quad \text{and} \quad \Pr[\text{mult}_a(y-x) = m-1] = 2/2^\ell.
  \]

- Finally, if \( r < w-\ell \), then \( \text{mult}_a(y-x) \) consists of zero or more random bits, followed by a 1, followed by one or more 0s, so
  \[
  \Pr[\text{mult}_a(y-x) = 0] = \Pr[\text{mult}_a(y-x) = m-1] = 0.
  \]

In all cases, we have \( \Pr[\text{mult}_a(x) = \text{mult}_a(y)] \leq 2/2^\ell \), as required. \( \square \)
\section*{12.6 High Probability Bounds: Balls and Bins}

Although any particular search in a chained hash tables requires only constant expected time, but what about the worst search time? Assuming that we are using ideal random hash functions, this question is equivalent to the following more abstract problem. Suppose we toss \( n \) balls independently and uniformly at random into one of \( m \) 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 \( m \) 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, \( \E[X_j] = 1 \) for all \( j \).

Now consider the probability that bin \( j \) contains at least \( k \) balls. There are \( \binom{n}{k} \) choices for those \( k \) balls; each chosen ball has probability \( \frac{1}{n} \) of landing in bin \( j \). Thus,

\[
\Pr[X_j \geq k] = \binom{n}{k} \left( \frac{1}{n} \right)^k \leq \frac{n^k}{k!} \left( \frac{1}{n} \right)^k = \frac{1}{k!}
\]

Setting \( k = 2c \log n / \log \log n \), we have

\[
k! \geq k^{k/2} \left( \frac{2c \log n}{\log \log n} \right)^{2c \log n / \log \log n} \geq \left( \frac{\sqrt{\log n}}{2c \log n / \log \log n} \right)^{2c \log n / \log \log n} = 2^{c \log n} = n^c,
\]

which implies that

\[
\Pr \left[ X_j \geq \frac{2c \log n}{\log \log n} \right] < \frac{1}{n^c}.
\]

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

\[
\Pr \left[ \max_j X_j > \frac{2c \log n}{\log \log n} \right] = \Pr \left[ X_j > \frac{2c \log n}{\log \log n} \text{ for all } j \right] \leq \sum_{j=1}^{n} \Pr \left[ X_j > \frac{2c \log n}{\log \log n} \right] < \frac{1}{n^{c-1}}.
\]

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

However, if we make the hash table large enough, we can expect every ball to land in its own bin. Suppose there are \( m \) bins. Let \( C_{ij} \) be the indicator variable that equals 1 if and only if \( i \neq j \) and ball \( i \) and ball \( j \) land in the same bin, and let \( C = \sum_{i<j} C_{ij} \) be the total number of pairwise collisions. Since the balls are thrown uniformly at random, the probability of a collision is exactly \( 1/m \), so \( \E[C] = \binom{n}{2}/m \). In particular, if \( m = n^2 \), the expected number of collisions is less than \( 1/2 \).

To get a high probability bound, let \( X_j \) denote the number of balls in bin \( j \), as in the previous proof. We can easily bound the probability that bin \( j \) is empty, by taking the two most significant terms in a binomial expansion:

\[
\Pr[X_j = 0] = \left( 1 - \frac{1}{m} \right)^n = \sum_{i=1}^{n} \binom{n}{i} \left( \frac{1}{m} \right)^i = 1 - \frac{n}{m} + \Theta \left( \frac{n^2}{m^2} \right) > 1 - \frac{n}{m}
\]

We can similarly bound the probability that bin \( j \) contains exactly one ball:

\[
\Pr[X_j = 1] = n \cdot \frac{1}{m} \left( 1 - \frac{1}{m} \right)^{n-1} = \frac{n}{m} \left( 1 - \frac{n-1}{m} + \Theta \left( \frac{n^2}{m^2} \right) \right) > \frac{n}{m} - \frac{n(n-1)}{m^2}
\]
It follows immediately that \( \Pr[X_j > 1] < n(n-1)/m^2 \). The union bound now implies that \( \Pr[\hat{X} > 1] < n(n-1)/m \). If we set \( m = n^{2+\epsilon} \) for any constant \( \epsilon > 0 \), then the probability that no bin contains more than one ball is at least \( 1 - 1/n^\epsilon \).

**Lemma 6.** For any \( \epsilon > 0 \), if \( n \) balls are thrown independently and uniformly into \( n^{2+\epsilon} \) bins, then with high probability, no bin contains more than one ball.

We can give a slightly weaker version of this lemma that assumes only near-universal hashing. Suppose we hash \( n \) items into a table of size \( m \). Linearity of expectation implies that the expected number of pairwise collisions is

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

In particular, if we set \( m = cn^2 \), the expected number of collisions is less than \( 1/c \), which implies that the probability of even a single collision is less than \( 1/c \).

### 12.7 Perfect Hashing

So far we are faced with two alternatives. If we use a small hash table to keep the space usage down, even if we use ideal random hash functions, the resulting worst-case expected search time is \( \Theta(\log n/\log \log n) \) with high probability, which is not much better than a binary search tree. On the other hand, we can get constant worst-case search time, at least in expectation, by using a table of roughly quadratic size, but that seems unduly wasteful.

Fortunately, there is a fairly simple way to combine these two ideas to get a data structure of linear expected size, whose expected worst-case search time is constant. At the top level, we use a hash table of size \( m = n \), but instead of linked lists, we use secondary hash tables to resolve collisions. Specifically, the \( j \)th 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 \( \mathbb{E}[\sum_i n_i^2] < 3n \).

**Proof:** let \( h(x) \) denote the position of \( x \) in the primary hash table. We rewrite \( \sum_i \mathbb{E}[n_i^2] \) in terms of the indicator variables \( [h(x) = i] \) as follows. The first equation uses the definition of \( n_i \); the rest is just routine algebra.
\[
\sum_i n_i^2 = \sum_i \left( \sum_x [h(x) = i] \right)^2 \\
= \sum_i \left( \sum_x \sum_y [h(x) = i] [h(y) = i] \right) \\
= \sum_i \left( \sum_x [h(x) = i]^2 + 2 \sum_x [h(x) = i] \sum_y [h(y) = i] \right) \\
= \sum_x [h(x) = i]^2 + 2 \sum_{x < y} [h(x) = i] [h(y) = i] \\
= \sum_x [h(x) = i] + 2 \sum_{x < y} [h(x) = h(y)]
\]

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
\[
E\left[ \sum_i n_i^2 \right] = n + 2 \sum_{x < y} \Pr[h(x) = h(y)] \leq n + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{2}{n} = 3n - 2. \quad \square
\]

This lemma immediately implies that the expected size of our two-level hash table is \( O(n) \).

By our earlier analysis, the expected worst-case search time is \( O(1) \).

### 12.8 Open Addressing

Another method used to resolve collisions in hash tables is called open addressing. Here, rather than building secondary data structures, we resolve collisions by looking elsewhere in the table. Specifically, we have a sequence of hash functions \( \langle h_0, h_1, h_2, \ldots, h_{m-1} \rangle \), 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 are no empty slots.

```plaintext
OPENADDRESSSEARCH(x):
for i ← 0 to m - 1
    if T[h_i(x)] = x
        return h_i(x)
    else if T[h_i(x)] = Ø
        return 'absent'
return 'full'
```

The algorithm for inserting a new item into the table is similar; only the second-to-last line is changed to \( T[h_i(x)] \leftarrow x \). Notice that for an open-addressed hash table, the load factor is never bigger than 1.

Just as with chaining, we’d like to pretend that the sequence of hash values is truly random, for purposes of analysis. Specifically, most open-addressed hashing analysis uses the following assumption, which is impossible to enforce in practice, but leads to reasonably predictive results for most applications.
Strong uniform hashing assumption:

For any item \(x\), the probe sequence \(\langle h_0(x), h_1(x), \ldots, h_{m-1}(x) \rangle\) is equally likely to be any permutation of the set \{0, 1, 2, \ldots, m-1\}.

Let’s compute the expected time for an unsuccessful search in light of this assumption. Suppose there are currently \(n\) elements in the hash table. The strong uniform hashing assumption has two important consequences:

- **Uniformity**: Each hash value \(h_i(x)\) is equally likely to be any integer in the set \{0, 1, 2, \ldots, m-1\}.
- **Independence**: If we ignore the first probe, the remaining probe sequence \(\langle h_1(x), h_2(x), \ldots, h_{m-1}(x) \rangle\) is equally likely to be any permutation of the smaller set \{0, 1, 2, \ldots, m-1\} \{\(h_0(x)\)\}.

The first sentence implies that the probability that \(T[h_0(x)]\) is occupied is exactly \(n/m\). The second sentence implies that if \(T[h_0(x)]\) is occupied, our search algorithm recursively searches the rest of the hash table! Since the algorithm will never again probe \(T[h_0(x)]\), for purposes of analysis, we might as well pretend that slot in the table no longer exists. Thus, we get the following recurrence for the expected number of probes, as a function of \(m\) and \(n\):

\[
E[T(m, n)] = 1 + \frac{n}{m} E[T(m-1, n-1)].
\]

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 \(E[T(m, n)] \leq m/(m-n)\):

\[
\begin{align*}
E[T(m, n)] &= 1 + \frac{n}{m} E[T(m-1, n-1)] \\
&\leq 1 + \frac{n}{m} \cdot \frac{m-1}{m-n} \quad \text{[induction hypothesis]} \\
&< 1 + \frac{n}{m} \cdot \frac{m}{m-n} \quad \text{[m-1 < m]} \\
&= \frac{m}{m-n} \quad \text{[algebra]}
\end{align*}
\]

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

### 12.9 Linear and Binary Probing

In practice, however, we can’t generate ideal random probe sequences, so we must rely on a simpler probing scheme to resolve collisions. Perhaps the simplest scheme is **linear probing**—use a single hash function \(h(x)\) and define

\[
h_i(x) := (h(x) + i) \mod m
\]

This strategy has several advantages, in addition to its obvious simplicity. First, because the probing strategy visits consecutive entries in the hash 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 \( \ell \) (in a binary multiplicative hashing), and define

\[
    h_i(x) := h(x) \oplus i
\]

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 \( h_i(x) := (h(x) + i^2) \mod m \), and double hashing, where we use two hash functions \( h \) and \( h' \) and set \( h_i(x) := (h(x) + i \cdot h'(x)) \mod m \). These methods have some theoretical advantages over linear and binary probing, but they are not as efficient in practice, primarily due to cache effects.

**12.10 Analysis of Binary Probing**

**Lemma 8.** In a hash table of size \( m = 2^\ell \) containing \( n \leq m/4 \) 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 \( \ell \), 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 \( \ell \).

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 \( H[h(x)] \):

\[
    B_k(x) = \left[2^k [h(x)/2^k] .. 2^k [h(x)/2^k] + 2^k − 1\right]
\]

Similarly, let \( B'_k(x) \) denote the sibling of \( B_k(x) \) in the block tree; that is, \( B'_k(x) = B_{k+1}(x) \setminus B_k(x) \). We refer to each \( B_k(x) \) as an ancestor of \( x \) and each \( B'_k(x) \) as an uncle of \( x \). The proper ancestors of any uncle of \( x \) are also proper ancestors of \( x \).

The binary probing algorithm can be recast conservatively as follows:

```plaintext
BinaryProbe(x):
    if H[h(x)] = x
        return True
    if H[h(x)] is empty
        return False
    for k = 0 to \( \ell − 1 \)
        for each index j in \( B'_k(x) \)
            if H[j] = x
                return True
            if H[j] is empty
                return False
```

12
For purposes of analysis, suppose the target item $x$ is not in the table. (The time to search for an item that is in the table can only be faster.) Then the expected running time of BinaryProbe($x$) can be expressed as follows:

$$E[T(x)] \leq \sum_{k=0}^{\ell-1} O(2^k) \cdot \Pr[B'_k(x) \text{ is full}].$$

Assuming ideal random hashing, all blocks at the same level have equal probability of being full. Let $F_k$ denote the probability that a fixed level-$k$ block is full. Then we have

$$E[T(x)] \leq \sum_{k=0}^{\ell-1} O(2^k) \cdot F_k.$$

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 $j \geq k$ was already full. Let $B'_j(x)$ be the first uncle of $B_k(x)$ to become full. The only blocks that can overflow into $B_j(y)$ are its uncles, which are all either ancestors or uncles of $B_k(x)$. But when $B_j(y)$ became full, no other uncle of $B_k(x)$ was full. Moreover, $B_k(x)$ was not yet full (because there was still room for $y$), so no ancestor of $B_k(x)$ was full. It follows that $B'_j(x)$ is popular.

We conclude that if a block is full, then either that block or one of its uncles is popular. Thus, if we write $P_k$ to denote the probability that a fixed level-$k$ block is popular, we have

$$F_k \leq 2P_k + \sum_{j>k} P_j.$$

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,

$$P_k = \left( \frac{n}{2^k} \right) \left( \frac{2^k}{m} \right)^2 \leq \frac{n^2 2^{k^2}}{(2^k)! m^{2k}} < \left( \frac{en}{m} \right)^{2^k}$$

(The last inequality uses a crude form of Stirling’s approximation: $n! > n^n/e^n$.) Our assumption $n \leq m/4$ implies the simpler inequality $P_k < (e/4)^{2^k}$. Because $e < 4$, it is easy to see that $P_k < 4^{-k}$ for all sufficiently large $k$.

It follows that $F_k = O(4^{-k})$, which implies that the expected search time is at most $\sum_{k \geq 0} O(2^k) \cdot O(4^{-k}) = \sum_{k \geq 0} O(2^{-k}) = O(1)$.

\[\Box\]

### 12.11 Cuckoo Hashing

Write this.

### Exercises

1. Your boss wants you to find a **perfect** hash function for mapping a known set of $n$ items into a table of size $m$. A hash function is perfect if there are no collisions; each of the $n$ items
is mapped to a different slot in the hash table. Of course, a perfect hash function is only possible if \( m \geq n \). (This is a different definition of “perfect” than the one considered in the lecture notes.) After cursing your algorithms instructor for not teaching you about (this kind of) perfect hashing, you decide to try something simple: repeatedly pick ideal random hash functions until you find one that happens to be perfect.

(a) Suppose you pick an ideal random hash function \( h \). What is the exact expected number of collisions, as a function of \( n \) (the number of items) and \( m \) (the size of the table)? Don’t worry about how to resolve collisions; just count them.

(b) What is the exact probability that a random hash function is perfect?

(c) What is the exact expected number of different random hash functions you have to test before you find a perfect hash function?

(d) What is the exact probability that none of the first \( N \) random hash functions you try is perfect?

(e) How many ideal random hash functions do you have to test to find a perfect hash function with high probability?

2. (a) Describe a set of hash functions that is uniform but not (near-)universal.

(b) Describe a set of hash functions that is universal but not (near-)universal.

(c) Describe a set of hash functions that is universal but (near-)universal.

(d) A family of hash functions is **pairwise independent** if knowing the hash value of any one item gives us absolutely no information about the hash value of any other item; more formally,

\[
\Pr_{h \in \mathcal{H}} [h(x) = i \mid h(y) = j] = \Pr_{h \in \mathcal{H}} [h(x) = i]
\]

or equivalently,

\[
\Pr_{h \in \mathcal{H}} [(h(x) = i) \land (h(y) = j)] = \Pr_{h \in \mathcal{H}} [h(x) = i] \cdot \Pr_{h \in \mathcal{H}} [h(y) = j]
\]

for all distinct items \( x \neq y \) and all (possibly equal) hash values \( i \) and \( j \).

Describe a set of hash functions that is uniform but not pairwise independent.

(e) Describe a set of hash functions that is pairwise independent but not (near-)universal.

(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-)universal.

(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 \( \mathcal{M}^B \) of binary multiplicative hash functions described in Section 12.5 is not uniform. [*Hint: What is \( \text{mult}_{b}(0) \)?]*

(b) Prove that \( \mathcal{M}^B \) is not pairwise independent. [*Hint: Compare \( \text{mult}_{b}(0) \) and \( \text{mult}_{b}(2^{w-1}) \).*]
(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

\[
h_{a,b}(x) := \left( (a \cdot x + b) \mod 2^{w+\ell} \right) \div 2^w = \left( \frac{(a \cdot x + b) \mod 2^{w+\ell}}{2^w} \right),
\]

and let \( \mathcal{M}^+ = \{ h_{a,b} \mid a, b \in [2^{w+\ell}] \text{ and } a \text{ odd} \} \). Prove that the family of hash functions \( \mathcal{M}^+ \) is strongly near-universal:

\[
\Pr_{h \in \mathcal{M}^+} \left[ (h(x) = i) \land (h(y) = j) \right] \leq \frac{2}{m^2}
\]

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 \( i \)th 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 \( \mathbb{E}[X] = O(\log n) \).
13 String Matching

13.1 Brute Force

The basic object that we consider in this lecture note is a string, which is really just an array. The elements of the array come from a set \( \Sigma \) called the alphabet; the elements themselves are called characters. Common examples are ASCII text, where each character is an seven-bit integer, or proteins, 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 'AMANPLANACATACANALPANAMA' and the pattern is 'CAN', then the output should be 15. If the pattern is 'SPAM', then the answer should be NONE. In most cases the pattern is much smaller than the text; to make this concrete, I’ll assume that \( m < n/2 \).

---

1Dan 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 canal—Panama!

---

© Copyright 2014 Jeff Erickson.
This work is licensed under a Creative Commons License (http://creativecommons.org/licenses/by-nc-sa/4.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden.
See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ for the most recent revision.
Here’s the ‘obvious’ brute force algorithm, but with one immediate improvement. The inner while loop compares the substring $T_s$ with $P$. If the two strings are not equal, this loop stops at the first character mismatch.

```
ALMOSTBRUTEFORCE(T[1..n], P[1..m]):
    for s ← 1 to n − m + 1
        equal ← TRUE
        i ← 1
        while equal and i ≤ m
            if T[s + i − 1] ≠ P[i]
                equal ← FALSE
            else
                i ← i + 1
            if equal
                return s
        return NONE
```

In the worst case, the running time of this algorithm is $O((n − m)m) = O(nm)$, and we can actually achieve this running time by searching for the pattern AAA...AAAB with $m − 1$ A’s, in a text consisting of $n$ A’s.

In practice, though, breaking out of the inner loop at the first mismatch makes this algorithm quite practical. We can wave our hands at this by assuming that the text and pattern are both random. Then on average, we perform a constant number of comparisons at each position $i$, so the total expected number of comparisons is $O(n)$. Of course, neither English nor DNA is really random, so this is only a heuristic argument.

### 13.2 Strings as Numbers

For the moment, let’s assume that the alphabet consists of the ten digits 0 through 9, so we can interpret any array of characters as either a string or a decimal number. In particular, let $p$ be the numerical value of the pattern $P$, and for any shift $s$, let $t_s$ be the numerical value of $T_s$:

$$p = \sum_{i=1}^{m} 10^{m-i} \cdot P[i] \quad t_s = \sum_{i=1}^{m} 10^{m-i} \cdot T[s + i - 1]$$

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:

$$p = P[m] + 10\left(P[m - 1] + 10(P[m - 2] + \cdots + 10(P[2] + 10 \cdot P[1])\cdots)\right)$$

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 $T[r + m]$:

$$t_{s+1} = 10(t_s - 10^{m-1} \cdot T[s]) + T[s + m]$$

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:

```
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 \((p \mod q)\) and \((t_s \mod q)\) are called the fingerprints of \( P \) and \( T_s \), respectively. We can now compute \((p \mod q)\) and \((t_1 \mod q)\) in \( O(m) \) time using Horner’s rule:

\[
p \mod q = P[m] + (\cdots + (10 \cdot (P[2] + (10 \cdot P[1] \mod q) \mod q) \mod q) \cdots) \mod q.
\]

Similarly, given \((t_s \mod q)\), we can compute \((t_{s+1} \mod q)\) in constant time as follows:

\[
t_{s+1} \mod q = (10 \cdot (t_s - ((10^{m-1} \mod q) \cdot T[s] \mod q) \mod q) \mod q) + T[s + m] \mod q.
\]

Again, we have to precompute the value \((10^{m-1} \mod q)\) to make this fast.

If \((p \mod q) \neq (t_s \mod q)\), then certainly \( P \neq T_s \). However, if \((p \mod q) = (t_s \mod q)\), 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, \( \bar{p} = p \mod q \) and \( \bar{t}_s = t_s \mod q \).) 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.

\[
\text{KARP RABIN}(T[1..n], P[1..m]):
\]

1. $q \leftarrow \text{a random prime number between 2 and } \lceil m^2 \log m \rceil$
2. $\sigma \leftarrow 10^{m-1} \mod q$
3. $\tilde{p} \leftarrow 0$
4. $\tilde{t}_1 \leftarrow 0$
5. for $i \leftarrow 1$ to $m$
   a. $\tilde{p} \leftarrow (10 \cdot \tilde{p} \mod q) + P[i] \mod q$
   b. $\tilde{t}_1 \leftarrow (10 \cdot \tilde{t}_1 \mod q) + T[i] \mod q$
6. for $s \leftarrow 1$ to $n - m + 1$
   a. if $\tilde{p} = \tilde{t}_s$
      i. $P = T_s$
      (brute-force $O(m)$-time comparison)
     b. \text{return } s$
     c. $\tilde{t}_{s+1} \leftarrow (10 \cdot \tilde{t}_s - (\sigma \cdot T[s] \mod q) \mod q) \mod q + T[s + m] \mod q$
8. \text{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 $\lfloor \log x \rfloor$ distinct prime divisors.

**Proof:** If $x$ has $k$ distinct prime divisors, then $x \geq 2^k$, since every prime number is bigger than 1. \qed

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 KARP RABIN runs in $O(n + E[F]m) = O(n)$ 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.

\[
\begin{array}{|l|}
\hline
\text{Polynomial Hashing} \\
\hline
\end{array}
\]

\[
\begin{array}{|l|}
\hline
\text{CARTERWEGMANKARPRABIN}(T[1..n], P[1..m]): \\
q \leftarrow \text{prime number larger than } m^2 \\
b \leftarrow \text{RANDOM}(q) - 1 \\
\sigma \leftarrow b^{m-1} \mod q \\
\tilde{p} \leftarrow 0 \\
\tilde{t}_1 \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } m \\
\quad \tilde{p} \leftarrow (b \cdot \tilde{p} \mod q) + P[i] \mod q \\
\quad \tilde{t}_1 \leftarrow (b \cdot \tilde{t}_1 \mod q) + T[i] \mod q \\
\text{for } s \leftarrow 1 \text{ to } n - m + 1 \\
\quad \text{if } \tilde{p} = \tilde{t}_s \\
\quad \quad \text{if } P = T_s \quad \langle \text{brute-force } O(m) \text{-time comparison} \rangle \\
\quad \quad \text{return } s \\
\quad \tilde{t}_{s+1} \leftarrow (b \cdot (\tilde{t}_s - (\sigma \cdot T[s] \mod q) \mod q) \mod q) + T[s + m] \mod q \\
\text{return None} \\
\end{array}
\]

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:

\[
p(b) = \sum_{i=1}^{m} b^i \cdot P[m-i] \quad t_s(b) = \sum_{i=1}^{m} b^i \cdot T[s-1+m-i],
\]

Now define \( \tilde{p}(b) = p(b) \mod q \) and \( \tilde{t}_s(b) = t_s(b) \mod q \).

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 \( \Pr_b[\tilde{p}(b) = \tilde{t}_s(b)] \leq \frac{(m-1)}{q} < \frac{1}{m} \). Linearity of expectation now implies that the expected number of false positives is \( O(n/m) \), so the modified Rabin-Karp algorithm also runs in \( O(n) \) expected time.

13.4 Redundant Comparisons

Let’s go back to the character-by-character method for string matching. Suppose we are looking for the pattern ‘ABRACADABRA’ in some longer text using the (almost) brute force algorithm described in the previous lecture. Suppose also that when \( s = 11 \), the substring comparison fails at the fifth position; the corresponding character in the text (just after the vertical line below) is not a C. At this point, our algorithm would increment \( s \) and start the substring comparison from scratch.

\[
\begin{array}{ccccccc}
\text{HOCUSPOCUSABRACADABRA} & \ldots \\
\text{ABRACADABRA} \\
\text{ABRACADABRA} \\
\end{array}
\]
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?

HOCUSPOCUSABRACADABRA... 
ABRA/CADABRA 
ABRACADABRA 
ABRACADABRA 
ABRACADABRA

Finally, when we get to $s = 15$, we can’t immediately rule out a match based on earlier comparisons. However, for precisely the same reason, we shouldn’t start the substring comparison over from scratch — we already know that $T[15] = P[4] = A$. Instead, we should start the substring comparison at the second character of the pattern, since we don’t yet know whether or not it matches the corresponding text character.

If you play with this idea long enough, you’ll notice that the character comparisons should always advance through the text. Once we’ve found a match for a text character, we never need to do another comparison with that text character again. In other words, we should be able to optimize the brute-force algorithm so that it always advances through the text.

You’ll also eventually notice a good rule for finding the next ‘reasonable’ shift $s$. A prefix of a string is a substring that includes the first character; a suffix is a substring that includes the last character. A prefix or suffix is proper if it is not the entire string. Suppose we have just discovered that $T[i] ≠ P[j]$. The next reasonable shift is the smallest value of $s$ such that $T[s..i−1]$, which is a suffix of the previously-read text, is also a proper prefix of the pattern.

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

### 13.5 Finite State Machines

We can interpret any string matching algorithm that always advance through the text as feeding the text through a special type of finite-state machine. A finite state machine is a directed graph. Each node (or state) in the string-matching machine is labeled with a character from the pattern, except for two special nodes labeled $\mathbf{5}$ and $\mathbf{10}$. 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 $\mathbf{5}$ and ending at $\mathbf{10}$. 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 $\mathbf{5}$, follow the success edge to the next node and increment $i$. (So there is no failure edge from the start node $\mathbf{5}$.)
- If $T[i] ≠ P[j]$, 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 $\mathbf{10}$, 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 $\mathbf{10}$, then there is no match.
A finite state machine for the string 'ABRADACABRA'. Thick arrows are the success edges; thin arrows are the failure edges.

The finite state machine is really just a (very!) convenient metaphor. In a real implementation, we would not construct the entire graph. Since the success edges always traverse the pattern characters in order, and each state has exactly one outgoing failure edge, we only have to remember the targets of the failure edges. We can encode this failure function in an array $\text{fail}[1..n]$, where for each index $j$, the failure edge from node $j$ leads to node $\text{fail}[j]$. Following a failure edge back to an earlier state corresponds exactly, in our earlier formulation, to shifting the pattern forward. The failure function $\text{fail}[j]$ tells us how far to shift after a character mismatch $T[i] \neq P[j]$. Here’s the actual algorithm:

```
KNUTH-MORRIS-PRATT(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 KNUTH-MORRIS-PRATT 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 KNUTH-MORRIS-PRATT in the worst case is $O(n)$.

### 13.6 Computing the Failure Function

We can now rephrase our second intuitive rule about how to choose a reasonable shift after a character mismatch $T[i] \neq P[j]$: $P[1..\text{fail}[j]−1]$ 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:

$$P[1..\text{fail}[j]−1]$$ 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 $\text{fail}[j]$ for all $j > 1$. By convention we set $\text{fail}[1] = 0$; 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.

<table>
<thead>
<tr>
<th>$P[i]$</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>D</th>
<th>A</th>
<th>B</th>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>fail[$i$]</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Failure function for the string ‘ABRACADABRA’
(Compare with the finite state machine on the previous page.)

We could easily compute the failure function in $O(m^2)$ 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 KMP, 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 $\text{fail}[1] = 0$ is obvious. Assuming inductively that we correctly computed $\text{fail}[i]$ through $\text{fail}[i−1]$ in line (*), we need to show that $\text{fail}[i]$ is also correct. Just after the $i$th iteration of line (*), we have $j = \text{fail}[i]$, 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 $\text{fail}^c[j]$ inductively as follows: $\text{fail}^0[j] = j$, and

$$\text{fail}^c[j] = \text{fail}[\text{fail}^{c−1}[j]] = \text{fail}[^c[\cdots[\text{fail}[j]]\cdots]].$$

In particular, if $\text{fail}^{c−1}[j] = 0$, then $\text{fail}^c[j]$ is undefined. We can easily show by induction that every string of the form $P[1..\text{fail}^c[j]−1]$ 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 $P[1..\text{fail}^c[j]]$—the one with smallest $c$—such that $P[\text{fail}^c[j]] = P[i]$. This is exactly what the while loop in ComputeFailure computes;
### Lecture 13: String Matching

#### ComputeFailure in action. Do this yourself by hand!
the \((c + 1)\)th iteration compares \(P[\text{fail}^c[j]] = P[\text{fail}^{c+1}[i]]\) against \(P[i]\). **ComputeFailure** is actually a dynamic programming implementation of the following recursive definition of \(\text{fail}[i]\):

\[
\text{fail}[i] = \begin{cases} 
0 & \text{if } i = 0, \\
\max_{c \geq 1} \{ \text{fail}^c[i - 1] + 1 \mid P[i - 1] = P[\text{fail}^c[i - 1]] \} & \text{otherwise.}
\end{cases}
\]

### 13.7 Optimizing the Failure Function

We can speed up **KnuthMorrisPratt** slightly by making one small change to the failure function. Recall that after comparing \(T[i]\) against \(P[j]\) and finding a mismatch, the algorithm compares \(T[i]\) against \(P[\text{fail}[j]]\). With the current definition, however, it is possible that \(P[j]\) and \(P[\text{fail}[j]]\) 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:

```plaintext
OPTIMIZE_FAILURE(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.

```plaintext
COMPUTE_OPT_FAIL(P[1..m]):
  j ← 0
  for i ← 1 to m
    if P[i] = P[j]
      fail[i] ← fail[j]
    else
      fail[i] ← j
    while j > 0 and P[i] ≠ P[j]
      j ← fail[j]
    j ← j + 1
```

This optimization slows down the preprocessing slightly, but it may significantly decrease the number of comparisons at each text character. The worst-case running time is still \(O(n)\); however, the constant is about half as big as for the unoptimized version, so this could be a significant improvement in practice. Several examples of this optimization are given on the next page.

### Exercises

1. Describe and analyze a two-dimensional variant of **KarpRabin** that searches for a given two-dimensional pattern \(P[1..p][1..q]\) within a given two-dimensional “text” \(T[1..m][1..n]\). Your algorithm should report all index pairs \((i, j)\) such that the subarray \(T[i..i+p-1][j..j+q-1]\) is identical to the given pattern, in \(O(pq + mn)\) expected time.

2. A palindrome is any string that is the same as its reversal, such as \(X\), \(ABBA\), or \(REDIVIDER\). Describe and analyze an algorithm that computes the longest palindrome that is a (not
Algorithms Lecture /one.Alt.oldstyle/three.oldstyle: String Matching

\[ A \quad B \quad R \quad A \quad C \quad A \quad D \quad A \quad B \quad R \quad A \]

$\begin{array}{c|cccccccc}
\hline
\text{unoptimized fail}[i] & 0 & 1 & 1 & 2 & 1 & 2 & 1 & 2 & 3 & 4 \\
\text{optimized fail}[i] & 0 & 1 & 1 & 0 & 2 & 0 & 2 & 0 & 1 & 1 & 1 \\
\end{array}$

Optimized finite state machine and failure function for the string 'ABRADACABRA'

\[ P[i] \begin{array}{ccccccccc}
\hline
\text{unoptimized fail}[i] & 0 & 1 & 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 & 5 & 6 & 5 \\
\text{optimized fail}[i] & 0 & 1 & 0 & 1 & 0 & 4 & 0 & 1 & 0 & 1 & 0 & 6 & 0 \\
\end{array} \]

\[ P[i] \begin{array}{ccccccccc}
\hline
\text{unoptimized fail}[i] & 0 & 1 & 1 & 2 & 3 & 1 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\text{optimized fail}[i] & 0 & 1 & 0 & 1 & 3 & 0 & 1 & 0 & 1 & 3 & 0 & 1 & 8 \\
\end{array} \]

\[ P[i] \begin{array}{ccccccccc}
\hline
\text{unoptimized fail}[i] & 0 & 1 & 1 & 1 & 2 & 3 & 4 & 5 & 6 & 2 & 3 & 4 & 5 \\
\text{optimized fail}[i] & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 6 & 1 & 1 & 0 & 1 \\
\end{array} \]

Failure functions for four more example strings.

necessarily proper) prefix of a given string \(T[1..n]\). Your algorithm should run in \(O(n)\) time (either expected or worst-case).

⋆3. How important is the requirement that the fingerprint modulus \(q\) is prime in the original Karp-Rabin algorithm? Specifically, suppose \(q\) is chosen uniformly at random in the range \(1..N\). If \(t_s \neq p\), what is the probability that \(\tilde{t}_s = \tilde{p}\)? What does this imply about the expected number of false matches? How large should \(N\) be to guarantee expected running time \(O(m + n)\)? [Hint: This will require some additional number theory.]

4. Describe a modification of \textsc{KnuthMorrisPratt} in which the pattern can contain any number of wildcard symbols $\bullet$, each of which matches an arbitrary string. For example, the pattern ABR$\bullet$CAD$\bullet$BRA appears in the text SCHABRAINAINAINCADBRA; in this case, the second $\bullet$ 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 \textsc{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?$\bullet$CAD?$\bullet$BRA appears in the text SCHABRUCADIBRANCH. Your algorithm should run in \(O(m + q n)\) 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 KNUTH-MORRIS-PRATT in which the pattern can contain any number of wildcard symbols $=$, each of which matches the same arbitrary single character. For example, the pattern $\text{WHUHOCUSPOC} = \text{S} \text{CADABRA}$ appears in the texts $\text{WHUUHOCUUSPOCOT}$ and $\text{ABRAHOCASPOCAS}$, but not in the text $\text{FRISHUHOCUSPOC} = \text{ESTIX}$. 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 $j \rightarrow \text{fail}[j] \rightarrow \text{fail}[\text{fail}[j]] \rightarrow \cdots \rightarrow 0$, or equivalently, the maximum number of iterations of the inner loop of KNUTH-MORRIS-PRATT. 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.

   (a) Describe and analyze a variant of KARP-RABIN that solves this problem in $O(m + n)$ expected time.
   (b) Describe and analyze a variant of KNUTH-MORRIS-PRATT 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 KARP-RABIN that solves this problem in $O(m + n)$ expected time.

(b) Describe and analyze a variant of KNUTH-MORRIS-PRATT that solves this problem in $O(m + n)$ expected time.
Jaques: But, for the seventh cause; how did you find the quarrel on the seventh cause?
Touchstone: Upon a lie seven times removed—bear your body more seeming, Audrey—as thus, sir. I did dislike the cut of a certain courtier’s beard: he sent me word, if I said his beard was not cut well, he was in the mind it was: this is called the Retort Courteous. If I sent him word again ‘it was not well cut,’ he would send me word, he cut it to please himself: this is called the Quip Modest. If again ‘it was not well cut,’ he disabled my judgment: this is called the Reply Churlish. If again ‘it was not well cut,’ he would answer, I spake not true: this is called the Reproof Valiant. If again ‘it was not well cut,’ he would say I lied: this is called the Counter-cheque Quarrelsome: and so to the Lie Circumstantial and the Lie Direct.

Jaques: And how oft did you say his beard was not well cut?
Touchstone: I durst go no further than the Lie Circumstantial, nor he durst not give me the Lie Direct; and so we measured swords and parted.

—William Shakespeare, As You Like It, Act V, Scene 4 (1600)

13 Randomized Minimum Cut

13.1 Setting Up the Problem

This lecture considers a problem that arises in robust network design. Suppose we have a connected multigraph $G$ 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.

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

A multigraph whose minimum cut has three edges.

¹A 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.
Karger’s algorithm uses a primitive operation called collapsing an edge. Suppose \( u \) and \( v \) are vertices that are connected by an edge in some multigraph \( G \). To collapse the edge \( \{u, v\} \), we create a new node called \( uv \), replace any edge of the form \( \{u, w\} \) or \( \{v, w\} \) with a new edge \( \{uv, w\} \), and then delete the original vertices \( u \) and \( v \). Equivalently, collapsing the edge shrinks the edge down to nothing, pulling the two endpoints together. The new collapsed graph is denoted \( G/\{u, v\} \). We don’t allow self-loops in our multigraphs; if there are multiple edges between \( u \) and \( v \), collapsing any one of them deletes them all.

Any edge in an \( n \)-vertex graph can be collapsed in \( O(n) \) time, assuming the graph is represented as an adjacency list; I’ll leave the precise implementation details as an easy exercise.

The correctness of our algorithms will eventually boil down the following simple observation: For any cut in \( G/\{u, v\} \), there is cut in \( G \) with exactly the same number of crossing edges. In fact, in some sense, the ‘same’ edges form the cut in both graphs. The converse is not necessarily true, however. For example, in the picture above, the original graph \( G \) has a cut of size 1, but the collapsed graph \( G/\{c, d\} \) does not.

This simple observation has two immediate but important consequences. First, collapsing an edge cannot decrease the minimum cut size. More importantly, collapsing an edge increases the minimum cut size if and only if that edge is part of every minimum cut.

### 13.2 Blindly Guessing

Let’s start with an algorithm that tries to guess the minimum cut by randomly collapsing edges until the graph has only two vertices left.

**GuessMinCut\((G)\):**

```
for \( i \leftarrow n \) downto 2
    pick a random edge \( e \) in \( G \)
    \( G \leftarrow G/e \)
return the only cut in \( G \)
```

Because each collapse requires \( O(n) \) time, this algorithm runs in \( O(n^2) \) time. Our earlier observations imply that as long as we never collapse an edge that lies in every minimum cut, our algorithm will actually guess correctly. But how likely is that?

Suppose \( G \) has only one minimum cut—if it actually has more than one, just pick your favorite—and this cut has size \( k \). Every vertex of \( G \) must lie on at least \( k \) edges; otherwise, we could separate that vertex from the rest of the graph with an even smaller cut. Thus, the number of incident vertex-edge pairs is at least \( kn \). Since every edge is incident to exactly two vertices, \( G \) must have at least \( kn/2 \) edges. That implies that if we pick an edge in \( G \) uniformly at random, the probability of picking an edge in the minimum cut is at most \( 2/n \). In other words, the probability that we don’t screw up on the very first step is at least \( 1 - 2/n \).
Once we’ve collapsed the first random edge, the rest of the algorithm proceeds recursively (with independent random choices) on the remaining \((n - 1)\)-node graph. So the overall probability \(P(n)\) that \textsc{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.

\[
P(n) \geq \prod_{i=3}^{n} \frac{i - 2}{i} = \frac{\prod_{i=3}^{n} (i - 2)}{\prod_{i=3}^{n} i} = \frac{\prod_{j=1}^{n-2} j}{\prod_{i=3}^{n} i} = \frac{2}{n(n-1)}
\]

### 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 \textit{amplification}.

```plaintext
KARGERMINCUT(G):
    mink \leftarrow \infty
    for i \leftarrow 1 to N
    do
        X \leftarrow \textsc{GuessMinCut}(G)
        if |X| < mink
            mink \leftarrow |X|
        minX \leftarrow 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 \textsc{KargerMinCut} returns the actual minimum cut. The only way for the algorithm to return the wrong answer is if \textsc{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 \(N = \binom{n}{2} \ln n\) for some constant \(c\), then \textsc{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 \textit{with high probability}. Thus, \textsc{KargerMinCut} computes the minimum cut of any \(n\)-node graph in \(O(n^4 \log n)\) time.

If we make the number of iterations even larger, say \(N = n^2(n - 1)/2\), the success probability becomes \(1 - e^{-n}\). When the failure probability is exponentially small like this, we say that the algorithm is correct \textit{with very high probability}. In practice, very high probability is usually overkill; high probability is enough. (Remember, there is a small but non-zero probability that your computer will transform itself into a kitten before your program is finished.)
13.4 Not-So-Blindly Guessing

The $O(n^4 \log n)$ running time is actually comparable to some of the simpler flow-based algorithms, but it’s nothing to get excited about. But we can improve our guessing algorithm, and thus decrease the number of iterations in the outer loop, by observing that as the graph shrinks, the probability of collapsing an edge in the minimum cut increases. At first the probability is quite small, only $2/n$, but near the end of execution, when the graph has only three vertices, we have a $2/3$ chance of screwing up!

A simple technique for working around this increasing probability of error was developed by David Karger and Cliff Stein. Their idea is to group the first several random collapses a ‘safe’ phase, so that the cumulative probability of screwing up is small—less than $1/2$, say—and a ‘dangerous’ phase, which is much more likely to screw up.

The safe phase shrinks the graph from $n$ nodes to $n/\sqrt{2} + 1$ nodes, using a sequence of $n - n/\sqrt{2} - 1$ random collapses. Following our earlier analysis, the probability that none of these safe collapses touches the minimum cut is at least

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

Now, to get around the danger of the dangerous phase, we use amplification. However, instead of running through the dangerous phase once, we run it twice and keep the best of the two answers. Naturally, we treat the dangerous phase recursively, so we actually obtain a binary recursion tree, which expands as we get closer to the base case, instead of a single path. More formally, the algorithm looks like this:

```
CONTRACT(G, m):
    for i ← n downto m
        pick a random edge e in G
        G ← G/e
    return G

BE betterGuess(G):
    if G has more than 8 vertices
        X₁ ← betterGuess(CONTRACT(G, n/\sqrt{2} + 1))
        X₂ ← betterGuess(CONTRACT(G, n/\sqrt{2} + 1))
        return min{X₁, X₂}
    else
        use brute force
```

This might look like we’re just doing to same thing twice, but remember that CONTRACT (and thus BETTERGUESS) is randomized. Each call to CONTRACT contracts an independent random set of edges; $X₁$ and $X₂$ are almost always different cuts.

 BETTERGUESS correctly returns the minimum cut unless both recursive calls return the wrong result. $X₁$ is the minimum cut of $G$ if and only if (1) none of the edges of the minimum cut are CONTRACTed and (2) the recursive call to BETTERGUESS returns the minimum cut of the CONTRACTed graph. Thus, if $P(n)$ denotes the probability that BETTERGUESS returns a minimum cut of an $n$-node graph, then $X₁$ is the minimum cut with probability at least $1/2 \cdot P(n/\sqrt{2} + 1)$. The same argument implies that $X₂$ is the minimum cut with probability at least $1/2 \cdot P(n/\sqrt{2} + 1)$. Because these two events are independent, we have the following recurrence, with base case $P(n) = 1$ for all $n ≤ 6$.

$$P(n) ≥ 1 - \left(1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2$$

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

For the running time, we get a simple recurrence that is easily solved using recursion trees or the Master theorem (after a domain transformation to remove the +1 from the recurrence).

\[ T(n) = O(n^2) + 2T\left(\frac{n}{\sqrt{2}} + 1\right) = O(n^2 \log n) \]

So all this splitting and recursing has slowed down the guessing algorithm slightly, but the probability of failure is exponentially smaller!

Let’s express the lower bound \( P(n) = \Omega(1/\log n) \) explicitly as \( P(n) \geq \alpha/\ln n \) for some constant \( \alpha \). (Karger and Stein’s proof implies \( \alpha > 2 \). If we call BETTERGUESS \( N = c \ln^2 n \) times, for some new constant \( c \), the overall probability of success is at least

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

By setting \( c \) sufficiently large, we can bound the probability of failure by an arbitrarily small polynomial function of \( n \). In other words, we now have an algorithm that computes the minimum cut with high probability in only \( O(n^2 \log^2 n) \) time!

### 13.5 Solving the Karger-Stein recurrence

Recall the following recurrence for the probability that BETTERGUESS successfully finds a minimum cut of an \( n \)-node graph:

\[ P(n) \geq 1 - \left(1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2}} + 1\right)\right)^2 \]

Karger and Stein solve this rather ugly recurrence through a series of functional transformations. Let \( p(k) \) denote the probability of success at the \( k \)-th level of recursion, counting upward from the base case. This function satisfies the recurrence

\[ p(k) \geq 1 - \left(1 - \frac{p(k-1)}{2}\right)^2 = p(k-1) - \frac{p(k-1)^2}{4} \]

with base case \( p(0) = 1 \). Let \( \tilde{p}(k) \) be the function that satisfies this recurrence with equality; clearly, \( p(k) \geq \tilde{p}(k) \). Substituting the function \( z(k) = 4/\tilde{p}(k) - 1 \) into this recurrence implies (after a bit of algebra) gives a new recurrence

\[ z(k) = z(k-1) + 2 + \frac{1}{z(k-1)} \]

with base case \( z(0) = 3 \). Clearly \( z(k) > 1 \) for all \( k \), so we have a conservative upper bound \( z(k) < z(k-1) + 3 \), which implies (by induction) that \( z(k) \leq 3k + 3 \). Substituting \( \tilde{p}(k) = 4/(z(k)+1) \) into this solution, we conclude that

\[ p(k) \geq \tilde{p}(k) > \frac{1}{3k+6} = \Omega(1/k). \]

To compute the number of levels of recursion that BETTERGUESS executes for an \( n \)-node graph, we solve the secondary recurrence

\[ k(n) = 1 + k\left(\frac{n}{\sqrt{2}} + 1\right) \]

with base cases \( k(n) = 0 \) for all \( n \leq 8 \). After a domain transformation to remove the +1 from the right side, the recursion tree method (or the Master theorem) implies that \( k(n) = \Theta(\log n) \).

We conclude that \( P(n) = p(k(n)) = \Omega(1/\log n) \), as promised. Whew!
Exercises

1. Suppose you had an algorithm to compute the minimum spanning tree of a graph in \( O(m) \) time, where \( m \) is the number of edges in the input graph. Use this algorithm as a subroutine to improve the running time of \textsc{GuessMinCut} from \( O(n^2) \) to \( O(m) \).

   (In fact, there is a randomized algorithm—due to Philip Klein, David Karger, and Robert Tarjan—that computes the minimum spanning tree of any graph in \( O(m) \) expected time. The fastest deterministic algorithm known in 2013 runs in \( O(m\alpha(m)) \) time.)

2. Suppose you are given a graph \( G \) with weighted edges, and your goal is to find a cut whose total weight (not just number of edges) is smallest.

   (a) Describe an algorithm to select a random edge of \( G \), where the probability of choosing edge \( e \) is proportional to the weight of \( e \).

   (b) Prove that if you use the algorithm from part (a), instead of choosing edges uniformly at random, the probability that \textsc{GuessMinCut} returns a minimum-weight cut is still \( \Omega(1/n^2) \).

   (c) What is the running time of your modified \textsc{GuessMinCut} algorithm?

3. Prove that \textsc{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 \textsc{BetterGuess} algorithm, where we pass in a real parameter \( \alpha > 1 \) in addition to the graph \( G \).

   \[
   \textsc{BetterGuess}(G, \alpha):
   \]
   \[
   n \leftarrow \text{number of vertices in } G
   \]
   \[
   \text{if } n > 8
   \]
   \[
   X_1 \leftarrow \textsc{BetterGuess} \left( \textsc{Contract}(G, n/\alpha), \alpha \right)
   \]
   \[
   X_2 \leftarrow \textsc{BetterGuess} \left( \textsc{Contract}(G, n/\alpha), \alpha \right)
   \]
   \[
   \text{return } \min\{X_1, X_2\}
   \]
   \[
   \text{else}
   \]
   \[
   \text{use brute force}
   \]

   Assume for this question that the input graph \( G \) has a unique minimum cut.

   (a) What is the running time of the modified algorithm, as a function of \( n \) and \( \alpha \)? [Hint: Consider the cases \( \alpha < \sqrt{2} \), \( \alpha = \sqrt{2} \), and \( \alpha > \sqrt{2} \) separately.]

   (b) What is the probability that \textsc{Contract}(\( G, n/\alpha \)) does not contract any edge in the minimum cut in \( G \)? Give both an exact expression involving both \( n \) and \( \alpha \), and a simple approximation in terms of just \( \alpha \). [Hint: When \( \alpha = \sqrt{2} \), the probability is approximately 1/2.]

   (c) Estimate the probability that \textsc{BetterGuess}(\( G, \alpha \)) returns the minimum cut in \( G \), by adapting the solution to the Karger-Stein recurrence. [Hint: Consider the cases \( \alpha < \sqrt{2} \), \( \alpha = \sqrt{2} \), and \( \alpha > \sqrt{2} \) separately.]
(d) Suppose we iterate \textsc{BetterGuess}(G, \alpha) 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 \alpha is this running time minimized?

(e) Suppose we modify \textsc{BetterGuess}(G, \alpha) further, to recurse four times instead of only twice. Now what is the best choice of \alpha? What is the resulting running time?