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

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

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

— Karl S. Kruszelnicki ('Dr. Karl'), *Karl Trek*, December 1997

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

— William S. Burroughs, "The Cut-Up Method of Brion Gysin"
in *The Third Mind* by William S. Burroughs and Brion Gysin (1978)

5 Randomized Algorithms

5.1 Nuts and Bolts

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

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

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

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

5.2 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?"

© Copyright 2010 Jeff Erickson. Released under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License (http://creativecommons.org/licenses/by-nc-sa/3.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden. See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ for the most recent revision.

The *average-case* running time is best defined by the *expected value*, over all inputs X of a certain size, of the algorithm's running time for X:[2]

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

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

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

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

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

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

5.3 Back to Nuts and Bolts

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

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

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

[2]The notation $\mathrm{E}[\,]$ for expectation has nothing to do with the shift operator **E** used in the annihilator method for solving recurrences!

[3]This is what most people think 'random' means, but they're wrong.

[4]Note that for this algorithm, the input is completely specified by the number n. Since we're choosing the nuts to test at random, even the order in which the nuts and bolts are presented doesn't matter. That's why I'm using the simpler notation $T(n)$ instead of $T(X)$.

If the target nut is the kth nut tested, our algorithm performs $\min\{k, n-1\}$ comparisons. In particular, if the target nut is the last nut chosen, we don't actually test it. Because we choose the next nut to test uniformly at random, the target nut is equally likely—with probability exactly $1/n$—to be the first, second, third, or kth bolt tested, for any k. Thus:

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

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

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

We can get exactly the same answer by thinking of this algorithm recursively. We always have to perform at least one test. With probability $1/n$, we successfully find the matching nut and halt. With the remaining probability $1 - 1/n$, we recursively solve the same problem but with one fewer nut. We get the following recurrence for the expected number of tests:

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

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

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

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

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

5.4 Finding All Matches

Not let's go back to the problem introduced at the beginning of the lecture: finding the matching nut for every bolt. The simplest algorithm simply compares every nut with every bolt, for a total of n^2 comparisons. The next thing we might try is repeatedly finding an arbitrary matched pair, using our very first nuts and bolts algorithm. This requires

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

comparisons in the worst case. So we save roughly a factor of two over the really stupid algorithm. Not very exciting.

Here's another possibility. Choose a *pivot* bolt, and test it against every nut. Then test the matching pivot nut against every other bolt. After these $2n-1$ tests, we have one matched pair, and the remaining nuts and bolts are partitioned into two subsets: those smaller than the pivot pair and those larger than the pivot pair. Finally, recursively match up the two subsets. The worst-case number of tests made by this algorithm is given by the recurrence

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

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

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

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

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

5.5 Reductions to and from Sorting

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

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

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

Unfortunately, since we aren't allowed to directly compare two bolts or two nuts, we can't use heapsort or mergesort to sort the nuts and bolts in $O(n\log n)$ worst case time. In fact, the problem of sorting nuts and bolts *deterministically* in $O(n\log n)$ time was only 'solved' in 1995[5], but both the algorithms and their analysis are incredibly technical and the constant hidden in the $O(\cdot)$ notation is quite large.

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

[5] János Komlós, Yuan Ma, and Endre Szemerédi, Sorting nuts and bolts in $O(n\log n)$ time, *SIAM J. Discrete Math* 11(3):347–372, 1998. See also Phillip G. Bradford, Matching nuts and bolts optimally, Technical Report MPI-I-95-1-025, Max-Planck-Institut für Informatik, September 1995. Bradford's algorithm is *slightly* simpler.

5.6 Recursive Analysis

Intuitively, we can argue that our quicksort-like algorithm will usually choose a bolt of approximately median size, and so the average numbers of tests should be $O(n \log n)$. We can now finally formalize this intuition. To simplify the notation slightly, I'll write $\overline{T}(n)$ in place of $\mathrm{E}[T(n)]$ everywhere.

Our randomized matching/sorting algorithm chooses its pivot bolt *uniformly at random* from the set of unmatched bolts. Since the pivot bolt is equally likely to be the smallest, second smallest, or kth smallest for any k, the expected number of tests performed by our algorithm is given by the following recurrence:

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

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

Yuck. At this point, we could simply *guess* the solution, based on the incessant rumors that quicksort runs in $O(n \log n)$ time in the average case, and prove our guess correct by induction. A similar inductive proof appears in [CLR, pp. 166–167], but it was removed from the new edition [CLRS]. That's okay; nobody ever really understood that proof anyway. (See Section 5.8 below for details.)

However, if we're only interested in asymptotic bounds, we can afford to be a little conservative. What we'd *really* like is for the pivot bolt to be the median bolt, so that half the bolts are bigger and half the bolts are smaller. This isn't very likely, but there is a good chance that the pivot bolt is close to the median bolt. Let's say that a pivot bolt is *good* if it's in the middle half of the final sorted set of bolts, that is, bigger than at least $n/4$ bolts and smaller than at least $n/4$ bolts. If the pivot bolt is good, then the *worst* split we can have is into one set of $3n/4$ pairs and one set of $n/4$ pairs. If the pivot bolt is bad, then our algorithm is still better than starting over from scratch. Finally, a randomly chosen pivot bolt is good with probability $1/2$.

These simple observations give us the following simple recursive *upper bound* for the expected running time of our algorithm:

$$\overline{T}(n) \le 2n - 1 + \frac{1}{2}\left(\overline{T}\left(\frac{3n}{4}\right) + \overline{T}\left(\frac{n}{4}\right)\right) + \frac{1}{2} \cdot \overline{T}(n)$$

A little algebra simplifies this even further:

$$\overline{T}(n) \le 4n - 2 + \overline{T}\left(\frac{3n}{4}\right) + \overline{T}\left(\frac{n}{4}\right)$$

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

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

5.7 Iterative Analysis

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

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

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

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

Now let B_i denote the ith smallest bolt, and N_j denote the jth smallest nut. For each i and j, define an indicator variable X_{ij} that equals 1 if our algorithm compares B_i with N_j and zero otherwise. Then the total number of nut/bolt comparisons is exactly

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} X_{ij}.$$

We are interested in the expected value of this double summation:

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

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

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

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

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

First let's assume that $i < j$. The only comparisons our algorithm performs are between some pivot bolt (or its partner) and a nut (or bolt) in the same subset. The only thing that can prevent us from comparing B_i and N_j is if some intermediate bolt B_k, with $i < k < j$, is chosen as a pivot before B_i or B_j. In other words:

Our algorithm compares B_i and N_j if and only if the first pivot chosen from the set $\{B_i, B_{i+1}, \ldots, B_j\}$ is either B_i or B_j.

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

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

Symmetric arguments give us $\mathrm{E}[X_{ij}] = \frac{2}{i-j+1}$ for all $i > j$. Since our algorithm is a little stupid, every bolt is compared with its partner, so $X_{ii} = 1$ for all i. (In fact, if a pivot bolt is the only bolt in its subset, we don't need to compare it against its partner, but this improvement complicates the analysis.)

Putting everything together, we get the following summation.

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

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

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

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

⋆5.8 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.

$$\begin{aligned}
\overline{T}(n) &= 2n - 1 + \frac{1}{n}\sum_{k=1}^{n}\left(\overline{T}(k-1) + \overline{T}(n-k)\right) \\
&= 2n - 1 + \frac{2}{n}\sum_{k=0}^{n-1}\overline{T}(k)
\end{aligned}$$

We then convert this 'full history' recurrence into a 'limited history' recurrence by shifting and subtracting away common terms. (I call this "Magic step #1".) To make this step slightly easier, we first multiply both sides of the recurrence by n to get rid of the fractions.

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

To solve this limited-history recurrence, we define a new function $t(n) = \overline{T}(n)/(n+1)$. (I call this "Magic step #2".) This gives us an even simpler recurrence for $t(n)$ in terms of $t(n-1)$:

$$\begin{aligned}
t(n) &= \frac{\overline{T}(n)}{n+1} \\
&= \frac{1}{n+1}\left(4 - \frac{3}{n} + (n+1)\frac{T(n-1)}{n}\right) \\
&= \frac{4}{n+1} - \frac{3}{n(n+1)} + t(n-1) \\
&= \frac{7}{n+1} - \frac{3}{n} + t(n-1)
\end{aligned}$$

I used the technique of partial fractions (remember calculus?) to replace $\frac{1}{n(n+1)}$ with $\frac{1}{n} - \frac{1}{n+1}$ in the last step. The base case for this recurrence is $t(0) = 0$. Once again, we have a recurrence that translates directly into a summation, which we can solve with just a few lines of algebra.

$$\begin{aligned}
t(n) &= \sum_{i=1}^{n}\left(\frac{7}{i+1} - \frac{3}{i}\right) \\
&= 7\sum_{i=1}^{n}\frac{1}{i+1} - 3\sum_{i=1}^{n}\frac{1}{i} \\
&= 7(H_{n+1} - 1) - 3H_n \\
&= 4H_n - 7 + \frac{7}{n+1}
\end{aligned}$$

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

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

Surprise, surprise, we get exactly the same solution!

Exercises

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

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

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

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

```
RANDOMMIN(A[1..n]):
  min ← ∞
  for i ← 1 to n in random order
      if A[i] < min
          min ← A[i]   (⋆)
  return min
```

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

   (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 (⋆)?

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

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

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

4. Suppose we want to write an efficient function RANDOMPERMUTATION(n) that returns a permutation of the integers $\langle 1, \ldots, n \rangle$ chosen uniformly at random.

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

```
RANDOMPERMUTATION(n):
  for i ← 1 to n
      π[i] ← i
  for i ← 1 to n
      swap π[i] ↔ π[RANDOM(n)]
```

(b) Consider the following implementation of RANDOMPERMUTATION.

```
RANDOMPERMUTATION(n):
  for i ← 1 to n
      π[i] ← NULL
  for i ← 1 to n
      j ← RANDOM(n)
      while (π[j] != NULL)
          j ← RANDOM(n)
      π[j] ← i
  return π
```

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

(c) Consider the following partial implementation of RANDOMPERMUTATION.

```
RANDOMPERMUTATION(n):
  for i ← 1 to n
      A[i] ← RANDOM(n)
  π ← SOMEFUNCTION(A)
  return π
```

Prove that if the subroutine SOMEFUNCTION is deterministic, then this algorithm cannot be correct. *[Hint: There is a one-line proof.]*

*(d) 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.

(e) Describe and analyze an implementation of RANDOMPERMUTATION that runs in expected worst-case time $O(n)$.

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

```
ONEINTHREE:
  if FAIRCOIN = 0
      return 0
  else
      return 1 − ONEINTHREE
```

(a) Prove that ONEINTHREE returns 1 with probability $1/3$.

(b) What is the *exact* expected number of times that this algorithm calls FAIRCOIN?

(c) Now suppose you are *given* a subroutine ONEINTHREE that generates a random bit that is equal to 1 with probability $1/3$. Describe a FAIRCOIN algorithm that returns either 0 or 1 with equal probability, using ONEINTHREE as a subroutine. **For this question, your *only* source of randomness is ONEINTHREE; in particular, you may not use the RANDOM function.**

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

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

```
LIGHTTHECIRCLE(n):
  k ← 0
  turn on light 0
  while at least one light is off
      with probability 1/2
          k ← (k + 1) mod n
      else
          k ← (k − 1) mod n
      if light k is off, turn it on
```

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

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

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

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

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

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

   (d) Suppose we start at vertex $n/2$ instead. State and prove a tight Θ-bound on the expected length of the random walk in this case.

8. 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).

9. The following randomized algorithm, sometimes called 'one-armed quicksort', selects the rth smallest element in an unsorted array $A[1..n]$. For example, to find the smallest element, you would call RANDOMSELECT$(A, 1)$; to find the median element, you would call RANDOMSELECT$(A, \lfloor n/2 \rfloor)$. Recall from lecture that PARTITION splits the array into three parts by comparing the pivot element $A[p]$ to every other element of the array, using $n-1$ comparisons altogether, and returns the new index of the pivot element. The subroutine RANDOM(n) returns an integer chosen uniformly at random between 1 and n, in $O(1)$ time.

```
RANDOMSELECT(A[1..n], r):
  k ← PARTITION(A[1..n], RANDOM(n))

  if r < k
      return RANDOMSELECT(A[1..k − 1], r)
  else if r > k
      return RANDOMSELECT(A[k + 1..n], r − k)
  else
      return A[k]
```

   (a) State a recurrence for the expected running time of RANDOMSELECT, as a function of n and r.

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

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

   (d) What is the expected number of times that RANDOMSELECT calls itself recursively?

10. 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.

11. Clock Solitaire is played with a standard deck of 52 cards, containing 13 ranks of cards in four different suits. 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.

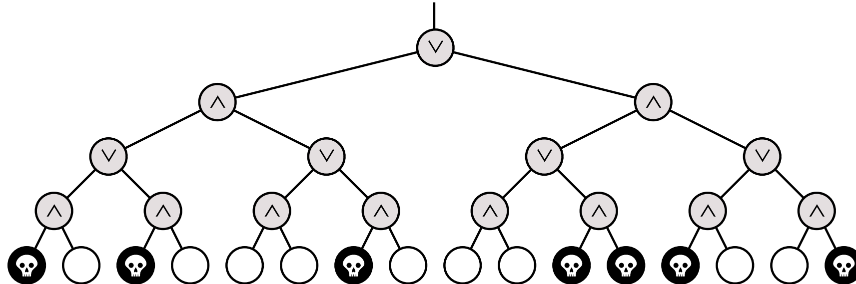
What is the *exact* probability that you win a game of Clock Solitaire, assuming that the cards are permuted uniformly at random before they are dealt into their piles?

12. 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 that is larger than $X[i]$. For example:

i	1	2	3	4	5	6	7	8	9
$X[i]$	83	54	16	31	45	99	78	62	27
$N[i]$	6	8	9	5	2	3	1	7	4

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

13. 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.



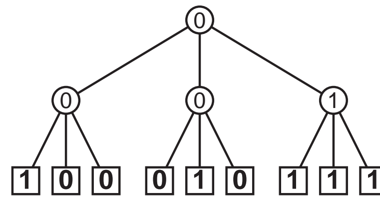
You can decide whether it's worth playing or not as follows. Imagine that the nodes at even levels (where it's your turn) are OR gates, the nodes at odd levels (where it's Death's turn) are AND gates. Each gate gets its input from its children and passes its output to its parent. White and black stand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

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



A majority tree with depth $n = 2$.

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

   (b) Describe and analyze a randomized algorithm that computes the value of the root in worst-case expected time $O(c^n)$ for some constant $c < 3$. *[Hint: Consider the special case $n = 1$. Recurse.]*

© Copyright 2010 Jeff Erickson. Released under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License (http://creativecommons.org/licenses/by-nc-sa/3.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden. See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ for the most recent revision.