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

2 Nuts and Bolts

The following interesting problem was first proposed by Gregory Rawlins around 1992. Suppose we are given n nuts and n bolts of different sizes. Each nut matches exactly one bolt and vice versa. The nuts and bolts are all almost exactly the same size, and we're playing with them in the dark, so we can't tell if one bolt is bigger than the other, or if one nut is bigger than the other. If we try to match a nut witch a bolt, however, the nut will be either too big, too small, or just right for the bolt. How quickly can we match each but to the corresponding bolt?

Before we attack this problem, let's think about something simpler. Suppose we want to find the nut that matches a particular bolt. The obvious algorithm — test every nut until we find a match — requires exactly $n-1$ tests in the worst case. We might have to check every but except one; if we get to the last nut without finding a match, we know that the last nut is the one we're looking for.

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

2.1 Deterministic vs. Randomized Algorithms

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

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

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

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

The problem with this definition is that we rarely, if ever, know the probability of getting any particular input X. We could compute average-case running times by *assuming* a particular probability distribution—for example, every possible input is equally likely—but this assumption

doesn't describe reality very well. Most real-life data, if it can be considered random at all, is random in some interesting way that can't be predicted in advance.

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

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

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

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

2.2   Finding One Match

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

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

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

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

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

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

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

---

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

We can derive 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.

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

2.3 Finding All Matches

Not let's go back to Rawlin's original problem: finding the unique matching nut for every bolt. Obviously we can compare every nut with every bolt, for a total of n^2 comparisons, but we should be able to do better than that. The next thing we might try is repeatedly finding an arbitrary matched pair, using our first nuts and bolts algorithm. This algorithm 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.

Rawlins proposed the following more interesting strategy when he introduced the problem. Choose a *pivot* bolt, and test it against every nut. Then test the matching pivot nut against every other bolt. After these $2n - 1$ tests, we have one matched pair, and the remaining nuts and bolts are partitioned into two subsets: those smaller than the pivot pair and those larger than the pivot pair. Finally, recursively match up the two subsets. The worst-case number of tests made by this algorithm is given by the recurrence

$$T(n) = 2n - 1 + \max_{1 \le k \le n} \left\{ T(k-1) + T(n-k) \right\}$$
$$= 2n - 1 + T(n-1)$$

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

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

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

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

2.4   Reductions to and from Sorting

Rawlins' algorithm for matching the nuts and bolts strongly resembles quicksort. The algorithm not only matches up the nuts and bolts, but also sorts them by size.

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

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

Unfortunately, since we aren't allowed to directly compare two bolts or two nuts, we can't use heapsort or mergesort to sort the nuts and bolts in $O(n \log n)$ worst case time. In fact, the problem of sorting nuts and bolts *deterministically* in $O(n \log n)$ time was only "solved" in 1995[2], but both the algorithms and their analysis are incredibly technical and the constant hidden in the $O(\cdot)$ notation is quite large.

2.5   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:

$$\overline{T}(n) = 2n - 1 + \mathrm{E}_k\big[\overline{T}(k-1) + \overline{T}(n-k)\big]$$

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

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

Yuck. At this point, we could simply *guess* the solution, based on the incessant rumors that quicksort runs in $O(n \log n)$ time in the average case, and prove our guess correct by induction. (See Section 2.7 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

---

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

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:

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

A little algebra simplifies this even further:

$$\overline{T}(n) \leq 4n - 2 + \overline{T}\left(\frac{3n}{4}\right) + \overline{T}\left(\frac{n}{4}\right)$$
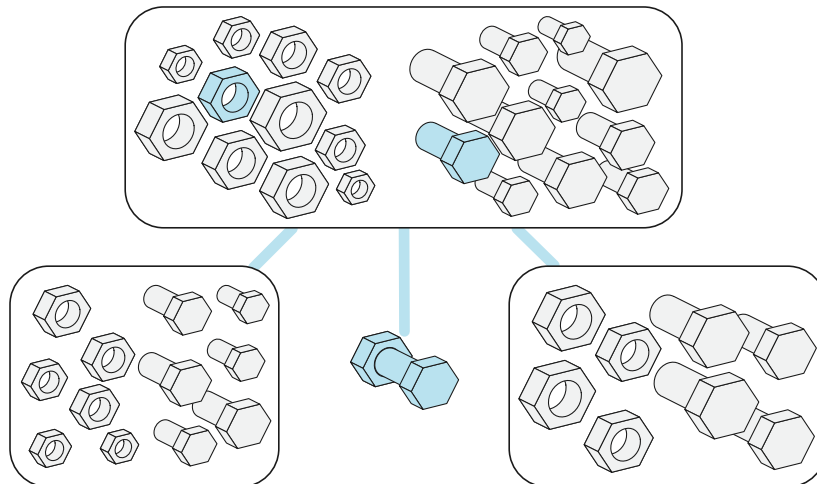
We can solve this recurrence using the recursion tree method, giving us the unsurprising upper bound $\overline{T}(n) = O(n \log n)$. A similar argument gives us the matching lower bound $\overline{T}(n) = \Omega(n \log n)$.

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

2.6   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 Rawlins' algorithm can be laid out in a binary tree, where each node corresponds to a pair of matching subsets of the nuts and bolts. In the usual recursive formulation, the algorithm performs a left-to-right depth-first traversal of this recursion tree, completely sorting all the nuts and bolts in the left subsets before making any progress on the right subsets.

Depth-first order is *convenient*, but it's not the only traversal order we can use. In fact, we can visit the nodes in the recursion tree in any order we like, as long as we visit each node before we visit any of is descendants.[3] This freedom allows us to recast Rawlins' quicksort-like algorithm in the following iterative form. The algorithm proceeds in n stages. In each stage, the algorithm randomly chooses an unmatched bolt b and partitions *whatever pair of subsets* contains b into nuts and bolts smaller than b, the nut that matches b, and nuts and bolts larger than b. After the nth stage, every bolt has been chosen as the pivot, and therefore all nuts and bolts have matched and sorted.

Suppose we always choose the next pivot bolt *uniformly at random* from the unmatched bolts. Then no matter which subset contains this bolt, the pivot bolt is equally likely to be any bolt *in that subset*. It follows (by induction) that our randomized iterative algorithm performs *exactly* the same set of tests as Rawlins' randomized recursive algorithm, but possibly (in fact, usually) 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 equals 0 otherwise. Then the total number of nut/bolt comparisons is exactly

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

We are interested in the expected value of this double summation; linearity of expectation allows us to write this expected value as follows:

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

By definition of expectation,

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

Thus, to analyze our algorithm, we only need to calculate $\Pr[X_{ij} = 1]$ for all i and j.

First let's assume that $i < j$. The only comparisons our algorithm performs are between some pivot bolt (or its partner) and a nut (or bolt) in the same subset. The only event that can prevent a comparison between B_i and N_j is choosing some intermediate pivot bolt B_k, with $i < k < j$, before choosing both B_i and 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.[4]

---

[3] In other words, in any *topological order*.

[4] 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.

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

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

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

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

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

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

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

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

$$= n + 4((n+1)(H_n - 1) - (n-1))$$

$$= 4nH_n - 7n + 4H_n$$

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

*2.7 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.

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

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.

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

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

7

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

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

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

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 $t(n-1)$:

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

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

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

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

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

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

Exercises

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(a) Describe and analyze an algorithm that chooses one element uniformly at random from a data stream, *without knowing the length of the stream in advance*. Your algorithm should spend $O(1)$ time per stream element and use $O(1)$ space (not counting the stream itself). You may assume that the stream is non-empty.

(b) Describe and analyze an algorithm that chooses k elements uniformly at random from a data stream, *without knowing the length of the stream in advance*. Your algorithm should spend $O(1)$ time per stream element and use $O(k)$ space (not counting the stream itself). You may assume that the stream has at least k elements.

5. Yet another method to analyze Rawlins' randomized matching algorithm considers all possible *intervals* of the nuts and bolts, rather than considering all possible *pairs*. For any indices $i \le j$, let $R(i, j)$ denote the probability that some *Recursive* subproblem asks us to match the nuts and bolts with ranks i through j. For example, $R(1, n) = 1$, because the *initial* call asks us to match the nuts and bolts with ranks 1 through n.

(a) Find the *exact* value of $R(i, j)$, as a simple function of i and j. [Hint: The cases $i = 1$ and $j = n$ are special.]

(b) We can compute the expected running time of Rawlins' algorithm as follows:

$$\mathrm{E}[T(n)] = \sum_{i \le j} \Pr[i..j \text{ appears}] \cdot \mathrm{E}[\text{time to partition } i..j \mid i..j \text{ appears}]$$
$$= \sum_{i \le j} R(i, j) \cdot (2j - 2i + 1)$$

Using your answer from part (a), compute the expected running time of Rawlins' algorithm.

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

```
QuickSelect(A[1..n], k):
    r ← Partition(A[1..n], Random(n))
    if k < r
        return QuickSelect(A[1..r − 1], k)
    else if k > r
        return QuickSelect(A[r + 1..n], k − r)
    else
        return A[k]
```

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

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

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

(d) Show that for any n and k, the expected running time of QUICKSELECT is $\Theta(n)$. You can use either the recurrence from part (a), the probability from part (b), or the probability from part (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?

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

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

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

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

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

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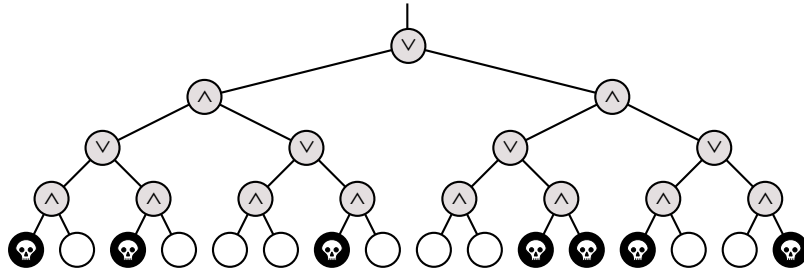- Otherwise, $X[next[i]]$ is the smallest element of X that is larger than $X[i]$.

For example:

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

Describe and analyze a randomized algorithm that determines whether a given number x appears in the array X in $O(\sqrt{n})$ expected time **without modifying the input arrays**.
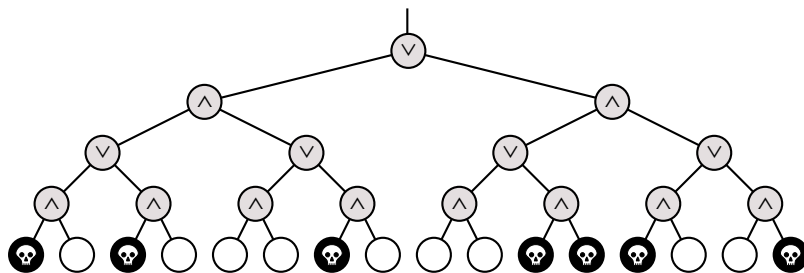
9. Death knocks on your door one cold blustery morning and challenges you to a game. Death knows that you are an algorithms student, so instead of the traditional game of chess, Death presents you with a complete binary tree with 4^n leaves, each colored either black

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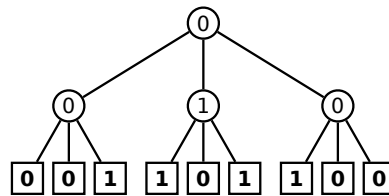
(a) Describe and analyze a deterministic algorithm to determine whether or not you can win. *[Hint: This is easy!]*

(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!]*

10. Death knocks on Dirk Gently's door one cold blustery morning and challenges him to a game. Emboldened by his experience with algorithms students, Death presents Dirk with a complete binary tree with 4^n leaves, each colored either black or white. There is a token at the root of the tree. To play the game, Dirk and Death will take turns moving the token from its current node to one of its children. The game will end after $2n$ moves, when the token lands on a leaf. If the final leaf is black, Dirk dies; if it's white, Dirk lives forever. Dirk moves first, so Death gets the last turn.[5]

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

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

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

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

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

A majority tree with depth $n = 2$.

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

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

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