*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

# 3 Randomized Algorithms (September 12)

## 3.1 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?

## 3.2 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:

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

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]

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

[1] "Whenever you lose something, it's always in the last place you look. So why not just look there first?"

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

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.

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!

## 3.3 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:

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

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:

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

[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 $T(X)$.

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.

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

## 3.4 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.

## 3.5 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, the constant hidden in the $O(\cdot)$ notation is extremely large, and worst of all, the solutions are nonconstructive—We know the algorithms exist, but we don't know what they look like!

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

## 3.6 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,

[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.

or $k$th smallest for any $k$, the expected number of tests performed by our algorithm is given by the following recurrence:

$$\begin{aligned}\overline{T}(n) &= 2n - 1 + \mathrm{E}_k\big[\overline{T}(k-1) + \overline{T}(n-k)\big] \\ &= \boxed{2n - 1 + \frac{1}{n}\sum_{k=0}^{n-1}\big(\overline{T}(k-1) + \overline{T}(n-k)\big)}\end{aligned}$$

The base case is $T(0) = 0$. (We can save a few tests by setting $T(1) = 1$, 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 understood that proof anyway.

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:

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

A little algebra simplifies this even further:

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

We can easily solve this recurrence using the recursion tree method, giving us the completely unsurprising upper bound $\boxed{\overline{T}(n) = O(n \log n)}$. Similar observations give us the matching lower bound $\overline{T}(n) = \Omega(n \log n)$.

## 3.7 Iterative Analysis

By making a simple change to our algorithm, which will have no effect on the number of tests, we can analyze it much more directly and exactly, without needing to solve a recurrence.

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 means our randomized iterative algorithm performs *exactly* the same set of tests as our randomized recursive algorithm, just 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:

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

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. In fact, with just a few lines of algebra, we can turn it into an exact, closed-form expression for the expected number of comparisons.

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

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

## *3.9 Masochistic Analysis

If we're feeling particularly masochistic, it is possible to 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.

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

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 slightly easier, we first multiply both sides of the recurrence by $n$ to get rid of the fractions.

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

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

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

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 $T(n) = (n+1)t(n)$ and simplifying gives us the exact solution to the original recurrence.

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

Surprise, surprise, we get exactly the same solution!