3 Randomized Algorithms (January 28)

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 *worst-case* running time of an algorithm, we mean 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]

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

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.

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

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

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. In other words, if there are k nuts left to test, each one will be chosen with probability $1/k$. Now what's the expected number of tests we have to perform? Intuitively, it should be about $n/2$, but let's formalize our intuition.

Let $T(n)$ denote the number of tests our algorithm uses to find a match for a single bolt out of n nuts.[3] 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 $\min\{k, n-1\}$ tests. 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.

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

[3] 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 get 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, and with 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:

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

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

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

This recurrence translates into a simple sum, which we can easily solve and get our final answer.

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

$$\Longrightarrow \; E[T(n)] = \frac{f(n)}{n} = \frac{n+1}{2} - \frac{1}{n}$$

3.4 Finding the Smallest Bolt

If we want to find the smallest bolt, we could use the following algorithm. Choose one "test" nut and one "test" bolt. If the nut is larger than the bolt, throw out the test nut and pick a new one. Similarly, if the nut is smaller than the bolt, throw out the test bolt and pick a new one. Finally, if the test bolt and test nut match, attach the nut to the bolt and choose a new test nut. It's not hard to prove (by induction, of course) that if we either get down to a single bolt or run out of nuts, the test bolt is the smallest one. In the worst case, our algorithm performs $2n-2$ tests, since we can stop the algorithm just before comparing the last bolt against the test nut. In the best case, it performs only $n-1$ tests; we might choose the smallest bolt as out first test bolt.

Now suppose at each step we choose the next nut or bolt uniformly at random from the untested nuts or bolts. What is the expected number of tests used to find the smallest bolt? Intuitively, the answer "should" be around $3n/2$, but I haven't worked out the details. If you're interested in getting some extra credit, work out the answer and send it to me!

3.5 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 tests every nut against every bolt, for a total of n^2 tests. The next thing we might try is repeatedly finding the smallest unmatched pair, using the algorithm outlined in the previous section. Since finding the smallest pair requires $2n-2$ tests in the worst case, this is almost as bad as just testing every nut against every bolt:

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

A slightly faster method is to repeatedly find an *arbitrary* matched pair, using our very first nuts and bolts algorithm. This requires

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

tests in the worst case, exactly half the cost of the previous algorithm.

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. We then 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 base case $T(1) = 0$, this recurrence solves to

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

In the worst case, this algorithm tests every nut-bolt pair but one!

3.6 Reductions to and from Sorting

The first matching algorithm looks just like selection sort, and the last one looks just like quicksort; both algorithms not only match up the nuts and bolts, but they also sort them.

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[4], but both the algorithms and their analysis are incredibly technical, the constant hidden in the $O()$ 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 when we start talking about lower bounds and NP-completeness.

3.7 Analyzing Randomized Quicksort

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.

[4] Phillip L. Bradford, Matching nuts and bolts optimally, Technical Report MPI-I-95-1-025, Max-Planck-Institut für Informatik, September 1995. See also 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.

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:

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

The recurrence looks horrible, but we can solve it. First we convert the "full history" recurrence into a "limited history" recurrence by shifting and subtracting.

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

Now we solve this limited history recurrence by expanding it and looking for a pattern. To simplify things slightly, I'll remove the annoying $3/n$ term.

$$\begin{aligned}
T(n) &\le 4+\frac{n+1}{n}T(n-1)\\
&\le 4+\frac{n+1}{n}\left(4+\frac{n}{n-1}T(n-2)\right)\\
&= 4+\frac{4(n+1)}{n}+\frac{n+1}{n-1}T(n-2)\\
&\le 4+\frac{4(n+1)}{n}+\frac{4(n+1)}{n-1}+\frac{n+1}{n-2}T(n-3)\\
&\le 4+\frac{4(n+1)}{n}+\frac{4(n+1)}{n-1}+\cdots+\frac{4(n+1)}{3}+\underbrace{\frac{(n+1)}{2}T(1)}_{0}\\
&= 4(n+1)\sum_{k=3}^{n+1}\frac{1}{k}\\
&= 4(n+1)(H_{n+1}-H_2)\\
&= 4n\ln n-6n+O(\log n).
\end{aligned}$$

Another approach we could take is to 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].

3.8 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, 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 this point, 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.

Let B_i denote the ith smallest bolt, and N_j denote the jth smallest nut. For each i and J, define a variable X_{ij} that equals 1 if our algorithm tests B_i against N_j and zero otherwise. The total number of nut/bolt tests is exactly

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

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

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

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}. Whenever B_i and N_j are tested, B_j and N_i are also tested, so $X_{ij} = X_{ji}$. By definition of expectation,

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

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

The only tests our algorithm performs are between some pivot nut (or its partner) and a nut (or bolt) in the same subset. If B_k has been chosen as a pivot bolt already, for some $i < k < j$, then B_i and N_j cannot possibly be in the same subset. In other words:

> **B_i and N_j are tested if and only if the first pivot we choose from the set $\{B_i, B_{i+1}, \ldots, B_j\}$ is either B_i or B_j.**

For any subset of bolts, each bolt is equally likely to be the first pivot, so for all $i < j$, we have

$$E[X_{ij}] = \frac{2}{j - i + 1}.$$

To simplify the analysis slightly, let's pretend that every bolt is tested against its partner, so $X_{ii} = 1$ for all i.[5]

Putting everything together, we get a sum that is simple enough to solve exactly, *much* more easily than we solved the recurrence.

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

We can rewrite this double sum as a single sum by gathering equal terms. For all $2 \leq \ell \leq n$, the fraction $1/\ell$ appears exactly $n + 1 - \ell$ times.

$$\begin{aligned} \cdots &= n + 4 \sum_{\ell=2}^{n} \frac{n + 1 - \ell}{\ell} \\ &= n + 4 \left((n-1) \sum_{\ell=2}^{n} \frac{1}{\ell} - \sum_{\ell=2}^{n} 1 \right) \\ &= n + 4((n+1)(H_n - 1) - (n-1)) \\ &= 4nH_n - 7n + 4H_n. \end{aligned}$$

[5] In fact, if the pivot bolt is the only bolt in its subset, we can identify its partner without any tests.