3.1 Introduction

As we discussed the first lecture, one issue with trying to do an "average-case" analysis of an algorithm is figuring out what the average case actually is. Different applications might result in vastly different empirical input distributions, and when we design basic, low-level algorithms and data structures we want our guarantees to hold in every possible application.

On the other hand, as we'll see today, sometimes it turns out that *adding* randomness can help, even in the worst-case! This is the paradigm of *randomized algorithms*: given a (worst-case) input, we use randomness in our algorithm and provide guarantees either in expectation or with high-probability. Note how different this is from trying to do an average-case analysis: instead of using randomness to model the input, we still allow arbitrary inputs but use randomness internally in our algorithm.

The first approach, of modeling the input using a distribution, is commonly seen in machine learning, computational biology, and other applications. There is much interesting work there, but if we model the input as a distribution we are fundamentally tied to that distribution, and lose generality. Adding randomness can be counterintuitive to some people, but let's us design algorithms that do not lose generality but instead have guarantees that hold (for any input) in expectation or with high probability.

As a side note, there is a class being taught right now and (I think) during most fall semesters by Vova Braverman on randomized algorithms. I highly recommend this class. This lecture (and possibly others) is my attempt to add some randomized algorithms to this class, since in the past 10-15 years randomized algorithms have moved from a specialized topic to the mainstream.

3.2 Basics of Quicksort

So today we're going to discuss a particularly nice use of randomization: *randomized quicksort*. First, recall the Quicksort algorithm. For now, let's assume that we are given an array of length n in which all elements are distinct – it's easy to see that allowing equal elements only helps us. Given an array of length n, quicksort does the following:

1. If n is 0 or 1, return.
2. Pick some element p as the *pivot*.
3. Compare each element of the array to p, making an array consisting of L (the elements less than p), followed by p, and then followed by G (the elements greater than p). In other words, p is now at its final location and all elements are on the correct side of p.
4. Recursively sort L and G.

Note that this algorithm is not totally well-specified, since we have not defined how to pick the pivot p. The traditional thing to do is come up with some arbitrary but deterministic choice like, for example, always choosing the first (not necessarily the smallest) element as the pivot. Next lecture we will see next lecture a better way of deterministically choosing a pivot, but the original (and probably most common) version of quicksort just picks the first element.

What is the running time of this basic version of quicksort? Everyone should know that the worst-case running time is $\Theta(n^2)$, but let's prove it again.

For an upper bound, note that by definition if we pick a pivot p in step 2 then after step 3 p is in the right place, and thus step 2 can be executed at most n times. Each time we execute step 2 we also have to execute step 3, but this takes at most $O(n)$ time. Thus the total running time is $O(n^2)$.

For a lower bound, suppose that the array is already sorted. Then when we choose p to be the first element, all other elements are compared to it and put in G. The pivot then stays where it is, and we recurse on all remaining elements. So the running time in this case is $T(n) = T(n-1) + cn$, which we now know is $\Omega(n^2)$.

Thus the running time of basic quicksort is $\Theta(n^2)$.

Randomized quicksort is the simple variation that picks a pivot p uniformly at random from the elements. We will show that in expectation the running time of randomized quicksort is $O(n \log n)$. Moreover, since the randomness is from the algorithm itself rather than over the distribution of the inputs, this bound holds *for every possible input*. So on any fixed input the running time is a random variable, but we will show that this random variable always has expectation at most $O(n \log n)$ (and clearly has maximum at most $O(n^2)$). This is better than an average-case bound because we don't have to assume anything about the input – it could be that in some application the arrays tend to be already sorted or in some other order which makes deterministic quicksort slow, but since our bound is worst-case (although in expectation over the randomness used by the algorithm) it still holds.

Before we prove this, though, we will need to cover some basic probability.

3.3 Probability Basics

I'm not going to get too formal, but I highly, highly recommend learning basic probability theory – either taking a class or teaching yourself. It's good for you, and also super useful. What follows is semi-formal, and only what we really need for this kind of basic analysis. You should look at CLRS Chapter 5 for some more details.

The example we will use is rolling a pair of dice and looking at the sum. Then the *sample space* is the set of possible outcomes, i.e. the 36 possible pairs of rolls. An *event* is a subset of the sample space. So you could talk about "the event that the first die is 3" or "the event that the dice add up to 7 or 11", etc. A *random variable* is a function from the sample space to numbers (real or integers). For example we could have a random variable X_1 equal to the first die, and a random variable X_2 equal to the roll of the second die. Then we could also define a random variable $X = X_1 + X_2$ which is the sum of the dice.

Random variables are incredibly important, and are in some sense the "right" way to think about much of probability theory. For us, note that the running time of randomized quicksort is a random variable: the sample space is the order of the pivots we pick, and the running time is some function of that (with values between 1 and $\Theta(n^2)$). One of the most important properties of a random variable is its expectation – this is what we will analyze for the running time of randomized quicksort, for example.

Definition 3.3.1 *Let X be a random variable over sample space Ω. Then the expected value of x is*

$$\mathbf{E}[X] = \sum_{e \in \Omega} \mathbf{Pr}[e] X(e). \tag{3.3.1}$$

So the expectation of a random variable is its average value across the sample space, but weighted according to the probability of each point in the sample space. It's common to rearrange this by grouping together terms on which the random variable has the same value, giving us

$$\mathbf{E}[X] = \sum_{y} y \cdot \mathbf{Pr}[X = y] \tag{3.3.2}$$

The conditional expectation of X is exactly what it sounds like: the expected value of X conditioned on event A.

$$\mathbf{E}[X|A] = \frac{1}{\mathbf{Pr}[A]} \sum_{e \in A} \mathbf{Pr}[e] X(e)$$

One amazing feature of expectations, which we will use over and over again, is *linearity of expectations*: for any two random variables X and Y, $\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y]$. The amazing thing about this is that it is true for all random variables no matter how weirdly they are correlated. So we can many times break up very complicated random variables into sums of simpler random variables, and then it suffices to just determine the expectation of each of the simpler random variables.

Consider our dice example. What is the expected sum of the dice? Call this random variable X. We could analyze it using equation (3.3.1), but that would require a 36-term sum. Or we could analyze it using equation (3.3.2), but that would require figuring out the probability that the sum equaled 2, the probability in equaled 3, etc. Neither of these are impossible (or even particularly difficult), but they require some combinatorics.

On the other hand, we can define two random other random variables: let X_1 be the outcome of the first dies, and let X_2 be the outcome of the second. Then $X = X_1 + X_2$, so by linearity of expectations $\mathbf{E}[X] = \mathbf{E}[X_1 + X_2] = \mathbf{E}[X_1] + \mathbf{E}[X_2]$. But now by definition we have that

$$\mathbf{E}[X_1] = \mathbf{E}[X_2] = \sum_{y=1}^{6} \frac{1}{6} y = \frac{21}{6} = 3.5,$$

and thus $\mathbf{E}[X] = 7$.

Theorem 3.3.2 (Linearity of Expectations) *For any two random variables X and Y, and any constants α and β, $\mathbf{E}[\alpha X + \beta Y] = \alpha \mathbf{E}[X] + \beta \mathbf{E}[Y]$.*

Proof: We will simply use equation (3.3.1):

$$\begin{aligned}\mathbf{E}[\alpha X + \beta Y] &= \sum_{e\in\Omega} \mathbf{Pr}[x](\alpha X(e) + \beta Y(e)) = \alpha \sum_{e\in\Omega} \mathbf{Pr}[e]X(e) + \beta \sum_{e\in\Omega} \mathbf{Pr}[e]Y(e) \\ &= \alpha\mathbf{E}[X] + \beta\mathbf{E}[Y],\end{aligned}$$

as claimed. ■

3.4 Randomized Quicksort

There are a few ways to analyze Randomized Quicksort – I think that this analysis is really slick, and demonstrated the power of linearity of expectations. Recall that we're assuming all elements are distinct, and note that the running time is essentially the total number of comparisons.

Theorem 3.4.1 *The expected number of comparisons in Randomized Quicksort is at most $O(n \log n)$.*

Proof: Let X be the total number of comparisons made (so X is a random variable). Let e_i be the ith smallest value in the given array, for $i \in \{1, \ldots, n\}$. We define another random variable X_{ij} for each $i, j \in \{1, \ldots, n\}$ with $i < j$. We set $X_{ij} = 1$ if the algorithm at some point compares e_i and e_j, and otherwise we set $X_{ij} = 0$.

It is easy to see that Randomized Quicksort will never compare the same two elements more than once, since if we compare e_i and e_j then we must have picked either e_i or e_j as the pivot, so after that iteration whichever one is the pivot will not be compared to any other elements. This means that $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$, so by linearity of expectations we know that

$$\mathbf{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}]. \tag{3.4.3}$$

So now we just need to understand $\mathbf{E}[X_{ij}]$. Let's start by considering two extremes. First, suppose that $j = i + 1$. Then $X_{ij} = 1$, since we will inevitably compare e_i and e_{i+1} (they will stay in the same subarray throughout the recursion until eventually one of them is chosen as the pivot, when they will be compared).

On the other hand, suppose that $i = 1$ and $j = n$. Then when we choose the first pivot, if we choose neither e_1 nor e_n then $e_1 \in L$ and $e_n \in G$, so the recursive calls separate them and we will never compare them. So the probability that we compare e_1 and e_n is $2/n$, and thus $\mathbf{E}[X_{1n}] = 2/n$.

Now consider general i and j. Let e_p be the pivot that we pick. If $p = i$ or $p = j$ then we will compare e_i and e_j. If $i < p < j$ then $e_i \in L$ while $e_j \in G$, so we will not compare them.

The more subtle cases are if $p < i$ or $p > j$. Suppose $p < i$. Then e_i and e_j are both in G, so are in the same subarray in the recursive call. So they may be compared, but they may not. Similarly, is $p > j$ then e_i and d_j are both in L, so may be compared later on but may not.

But let's look more closely. What if we condition on $i \le p \le j$? Then the probability that $X_{ij} = 1$ is exactly $\frac{2}{j-i+1}$. This is independent of n, and since if we pick $p < i$ or $p > j$ we simply retry (albeit

with a smaller value of n, this means that eventually $i \leq p \leq j$ and so $X_{ij} = 1$ with probability $\frac{2}{j-i+1}$. Thus $\mathbf{E}[X_{ij}] = \frac{2}{j-i+1}$.

Putting this together with equation (3.4.3), we get that

$$\mathbf{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} 2\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-i+1}\right) \leq 2nH_n = O(n \log n),$$

where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$ is the nth harmonic number, which is $O(\log n)$ (you can prove this by considering the integral of $1/x$). ■

A possibly easier way to see the bound on $\mathbf{E}[X_{ij}]$ is through a "dart game"[1]. When we pick a pivot p, we are essentially throwing a dart uniformly at random at the (sorted) array. If $p < i$ then since we recurse on G and e_i and e_j are both in G, we simply throw another dart but now uniformly distributed over G. Similarly, if $p > j$ we throw another dart but uniformly distributed over L. We keep doing this until eventually a dart lands in $\{i, i+1, \ldots, j\}$, when we stop. If this final dart hits i or j then $X_{ij} = 1$, otherwise $X_{ij} = 0$.

3.4.1 Another analysis

There's another analysis of randomized quicksort that is much less slick, but is a little more "direct". It goes via a recurrence relation. Let $T(n)$ denote the expected number of comparisons on an input of size n. When we pick a pivot e_p, we do $n-1$ comparisons and then call quicksort recursively on pieces of size $p-1$ and $n-p$. Since each p is equally likely, by linearity of expectations we get that

$$T(n) = (n-1) + \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i)) \tag{3.4.4}$$

Clearly $T(0) = 0$ (since there are no comparisons made for an array of size 1), so we can rewrite equation (3.4.4) as

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

To solve this recurrence, we will use the "guess and check" method. Let's guess that $T(n) \leq cn \ln n$ for some constant c. This is clearly true in the base case of $T(1)$. Now to verify it inductively, we do the following:

$$\begin{aligned} T(n) &= (n-1) + \frac{2}{n} \sum_{i=1}^{n-1} T(i) \\ &\leq (n-1) + \frac{2}{n} \sum_{i=1}^{n-1} ci \ln i \\ &\leq (n-1) + \frac{2c}{n} \int_{1}^{n} (x \ln x) dx \end{aligned}$$

[1] I first saw this analysis from Avrim Blum in the undergraduate algorithms class at CMU.

$$
\begin{aligned}
&= (n-1) + \frac{2c}{n}\left(\frac{1}{2}x^2 \ln x - \frac{1}{4}x^2\right)_{x=1}^{n} \\
&= (n-1) + \frac{2c}{n}\left(\frac{1}{2}n^2 \ln n - \frac{1}{4}n^2 + \frac{1}{4}\right) \\
&\leq n - 1 + cn \ln n - \frac{c}{2}n + \frac{c}{2n} \\
\leq cn \ln n,
\end{aligned}
$$

as long as $c \geq 2$.