Lecture 3

Probabilistic Analysis and Randomized Quicksort

3.1 Overview

In this lecture we begin by introducing randomized (probabilistic) algorithms and the notion of worst-case expected time bounds. We make this concrete with a discussion of a randomized version of the Quicksort sorting algorithm, which we prove has worst-case expected running time $O(n \log n)$. In the process, we discuss basic probabilistic concepts such as events, random variables, and linearity of expectation.

3.2 The notion of randomized algorithms

As we have discussed previously, we are interested in how the running time of an algorithm scales with the size of the input. In addition, we will usually be interested in *worst-case* running time, meaning the worst-case over all inputs of a given size. That is, if I is some input and $T(I)$ is running time of our algorithm on input I, then $T(n) = \max\{T(I)\}_{\text{inputs } I \text{ of size } n}$. One can also look at notions of *average-case* running time, where we are concerned with our performance on "typical" inputs I. However, one difficulty with average-case bounds is that it is often unclear in advance what typical inputs for some problem will really look like, and furthermore this gets more difficult if our algorithm is being used as a subroutine inside some larger computation. In particular, if we have a bound on the worst-case running time of an algorithm for some problem A, it means that we can now consider solving other problems B by somehow converting instances of B to instances of problem A. We will see many examples of this later when we talk about network flow and linear programming as well as in our discussions of NP-completeness.

On the other hand, there *are* algorithms that have a large gap between their performance "on average" and their performance in the worst case. Sometimes, in this case we can improve the worst-case performance by actually adding randomization into the algorithm itself. One classic example of this is the Quicksort sorting algorithm.

Quicksort: Given array of some length n,

1. Pick an element p of the array as the pivot (or halt if the array has size 0 or 1).

2. Split the array into sub-arrays LESS, EQUAL, and GREATER by comparing each element to the pivot. (LESS has all elements less than p, EQUAL has all elements equal to p, and GREATER has all elements greater than p).
3. recursively sort LESS and GREATER.

The Quicksort algorithm given above is not yet fully specified because we have not stated how we will pick the pivot element p. For the first version of the algorithm, let's always choose the leftmost element.

Basic-Quicksort: Run the Quicksort algorithm as given above, always choosing the leftmost element in the array as the pivot.

What is worst-case running time of Basic-Quicksort? We can see that if the array is already sorted, then in Step 2, all the elements (except p) will go into the GREATER bucket. Furthermore, since the GREATER array is in sorted order,[1] this process will continue recursively, resulting in time $\Omega(n^2)$. We can also see that the running time is $O(n^2)$ on any array of n elements because Step 1 can be executed at most n times, and Step 2 takes at most n steps to perform. Thus, the worst-case running time is $\Theta(n^2)$.

On the other hand, it turns out (and we will prove) that the average-case running time for Basic-Quicksort (averaging over all different initial orderings of the n elements in the array) is $O(n \log n)$. This fact may be small consolation if the inputs we are faced with are the bad ones (e.g., if our lists are nearly sorted already). One way we can try to get around this problem is to add randomization into the algorithm itself:

Randomized-Quicksort: Run the Quicksort algorithm as given above, each time picking a *random* element in the array as the pivot.

We will prove that for *any* given array input array I of n elements, the expected time of this algorithm $\mathbf{E}[T(I)]$ is $O(n \log n)$. This is called a Worst-case Expected-Time bound. Notice that this is better than an average-case bound because we are no longer assuming any special properties of the input. E.g., it could be that in our desired application, the input arrays tend to be mostly sorted or in some special order, and this does not affect our bound because it is a *worst-case* bound with respect to the input. It is a little peculiar: making the algorithm probabilistic gives us *more* control over the running time.

To prove these bounds, we first detour into the basics of probabilistic analysis.

3.3 The Basics of Probabilistic Analysis

Consider rolling two dice and observing the results. We call this an *experiment*, and it has 36 possible outcomes: it could be that the first die comes up 1 and the second comes up 2, or that the first comes up 2 and the second comes up 1, and so on. Each of these outcomes has probability 1/36 (assuming these are fair dice). Suppose we care about some quantity such as "what is the

[1]Technically, this depends on how the partitioning step is implemented, but will be the case for any reasonable implementation.

probability the sum of the dice equals 7?" We can compute that by adding up the probabilities of all the outcomes satisfying this condition (there are six of them, for a total probability of 1/6).

In the language of probability theory, such a probabilistic setting is defined by a *sample space* S and a *probability measure* p. The points of the sample space are the possible outcomes of the experiment and are called *elementary events*. E.g., in our case, the elementary events are the 36 possible outcomes for the pair of dice. In a discrete probability distribution (as opposed to a continuous one), the probability measure is a function $p(e)$ over elementary events e such that $p(e) \geq 0$ for all $e \in S$, and $\sum_{e \in S} p(e) = 1$. We will also use $\Pr(e)$ interchangeably with $p(e)$.

An *event* is a subset of the sample space. For instance, one event we might care about is the event that the first die comes up 1. Another is the event that the two dice sum to 7. The probability of an event is just the sum of the probabilities of the elementary events contained inside it (again, this is just for discrete distributions[2]).

A *random variable* is a function from elementary events to integers or reals. For instance, another way we can talk formally about these dice is to define the random variable X_1 representing the result of the first die, X_2 representing the result of the second die, and $X = X_1 + X_2$ representing the sum of the two. We could then ask: what is the probability that $X = 7$?

One property of a random variable we often care about is its *expectation*. For a discrete random variable X over sample space S, the expected value of X is:

$$\mathbf{E}[X] \quad = \quad \sum_{e \in S} \Pr(e) X(e). \tag{3.1}$$

In other words, the expectation of a random variable X is just its average value over S, where each elementary event e is weighted according to its probability. For instance, if we roll a single die and look at the outcome, the expected value is 3.5, because all six elementary events have equal probability. Often one groups together the elementary events according to the different values of the random variable and rewrites the definition like this:

$$\mathbf{E}[X] \quad = \quad \sum_{a} \Pr(X = a) a. \tag{3.2}$$

More generally, for any partition of the probability space into disjoint events $A_1, A_2, \ldots$, we can rewrite the expectation of random variable X as:

$$\mathbf{E}[X] = \sum_i \sum_{e \in A_i} \Pr(e) X(e) = \sum_i \Pr(A_i) \mathbf{E}[X | A_i], \tag{3.3}$$

where $\mathbf{E}[X|A_i]$ is the expected value of X given A_i, defined to be $\frac{1}{Pr(A_i)} \sum_{e \in A_i} \Pr(e) X(e)$. The formula (3.3) will be useful when we analyze Quicksort. In particular, note that the running time of Randomized Quicksort is a random variable, and our goal is to analyze its expectation.

3.3.1 Linearity of Expectation

An important fact about expected values is Linearity of Expectation: for any two random variables X and Y, $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$. This fact is incredibly important for analysis of algorithms because it allows us to analyze a complicated random variable by writing it as a sum of simple

[2]For a continuous distribution, the probability would be an integral over a density function.

random variables and then separately analyzing these simple RVs. Let's first prove this fact and then see how it can be used.

Theorem 3.1 (Linearity of Expectation) *For any two random variables X and Y, $\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y]$.*

Proof (for discrete RVs): This follows directly from the definition as given in (3.1).

$$\mathbf{E}[X+Y] = \sum_{e \in S} \Pr(e)(X(e)+Y(e)) = \sum_{e \in S} \Pr(e)X(e) + \sum_{e \in S} \Pr(e)Y(e) = \mathbf{E}[X] + \mathbf{E}[Y]. \quad \blacksquare$$

3.3.2 Example 1: Card shuffling

Suppose we unwrap a fresh deck of cards and shuffle it until the cards are completely random. How many cards do we expect to be in the same position as they were at the start? To solve this, let's think formally about what we are asking. We are looking for the expected value of a random variable X denoting the number of cards that end in the same position as they started. We can write X as a sum of random variables X_i, one for each card, where $X_i = 1$ if the ith card ends in position i and $X_i = 0$ otherwise. These X_i are easy to analyze: $\Pr(X_i = 1) = 1/n$ where n is the number of cards. $\Pr(x_i = 1)$ is also $\mathbf{E}[X_i]$. Now we use linearity of expectation:

$$\mathbf{E}[X] = \mathbf{E}[X_1 + \ldots + X_n] = \mathbf{E}[X_1] + \ldots + \mathbf{E}[X_n] = 1.$$

So, this is interesting: no matter how large a deck we are considering, the expected number of cards that end in the same position as they started is 1.

3.3.3 Example 2: Inversions in a random permutation

[hmm, lets leave this for homework]

3.4 Analysis of Randomized Quicksort

We now give two methods for analyzing randomized quicksort. The first is more intuitive but the details are messier. The second is a neat tricky way using the power of linearity of expectation: this will be a bit less intuitive but the details come out nicer.

3.4.1 Method 1

For simplicity, let us assume no two elements in the array are equal — when we are done with the analysis, it will be easy to look back and see that allowing equal keys could only improve performance. We now prove the following theorem.

Theorem 3.2 *The expected number of comparisons made by randomized quicksort on an array of size n is at most $2n \ln n$.*

Proof: First of all, when we pick the pivot, we perform $n-1$ comparisons (comparing all other elements to it) in order to split the array. Now, depending on the pivot, we might split the array into a LESS of size 0 and a GREATER of size $n-1$, or into a LESS of size 1 and a GREATER of size $n-2$, and so on, up to a LESS of size $n-1$ and a GREATER of size 0. All of these are equally likely with probability $1/n$ each. Therefore, we can write a recurrence for the expected number of comparisons $T(n)$ as follows:

$$T(n) \quad = \quad (n-1) + \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1)). \tag{3.4}$$

Formally, we are using the expression for Expectation given in (3.3), where the n different possible splits are the events A_i.[3] We can rewrite equation (3.4) by regrouping and getting rid of $T(0)$:

$$T(n) \quad = \quad (n-1) + \frac{2}{n}\sum_{i=1}^{n-1}T(i) \tag{3.5}$$

Now, we can solve this by the "guess and prove inductively" method. In order to do this, we first need a good guess. Intuitively, most pivots should split their array "roughly" in the middle, which suggests a guess of the form $cn\ln n$ for some constant c. Once we've made our guess, we will need to evaluate the resulting summation. One of the easiest ways of doing this is to upper-bound the sum by an integral. In particular if $f(x)$ is an increasing function, then

$$\sum_{i=1}^{n-1} f(i) \quad \leq \quad \int_1^n f(x)dx,$$

which we can see by drawing a graph of f and recalling that an integral represents the "area under the curve". In our case, we will be using the fact that $\int(cx\ln x)dx = (c/2)x^2\ln x - cx^2/4$.

So, let's now do the analysis. We are guessing that $T(i) \leq ci\ln i$ for $i \leq n-1$. This guess works for the base case $T(1) = 0$ (if there is only one element, then there are no comparisons). Arguing by induction we have:

$$\begin{aligned}
T(n) &\leq (n-1) + \frac{2}{n}\sum_{i=1}^{n-1}(ci\ln i) \\
&\leq (n-1) + \frac{2}{n}\int_1^n (cx\ln x)dx \\
&\leq (n-1) + \frac{2}{n}\Big((c/2)n^2\ln n - cn^2/4 + c/4\Big) \\
&\leq cn\ln n, \quad \text{for } c = 2. \quad \blacksquare
\end{aligned}$$

In terms of the number of comparisons it makes, Randomized Quicksort is equivalent to randomly shuffling the input and then handing it off to Basic Quicksort. So, we have also proven that Basic Quicksort has $O(n\log n)$ *average-case* running time.

[3] In addition, we are using Linearity of Expectation to say that the expected time *given* one of these events can be written as the sum of two expectations.

3.4.2 Method 2

Here is a neat alternative way to analyze randomized quicksort that is very similar to how we analyzed the card-shuffling example.

Alternative proof (Theorem 3.2): As before, let's assume no two elements in the array are equal since it is the worst case and will make our notation simpler. The trick will be to write the quantity we care about (the total number of comparisons) as a sum of simpler random variables, and then just analyze the simpler ones.

Define random variable X_{ij} to be 1 if the algorithm *does* compare the ith smallest and jth smallest elements in the course of sorting, and 0 if it does not. Let X denote the total number of comparisons made by the algorithm. Since we never compare the same pair of elements twice, we have

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij},$$

and therefore,

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

Let us consider one of these X_{ij}'s for $i < j$. Denote the ith smallest element in the array by e_i and the jth smallest element by e_j, and conceptually imagine lining up the elements in sorted order. If the pivot we choose is between e_i and e_j then these two end up in different buckets and we will never compare them to each other. If the pivot we choose *is* either e_i or e_j then we *do* compare them. If the pivot is less than e_i or greater than e_j then both e_i and e_j end up in the same bucket and we have to pick another pivot. So, we can think of this like a dart game: we throw a dart at random into the array: if we hit e_i or e_j then X_{ij} becomes 1, if we hit between e_i and e_j then X_{ij} becomes 0, and otherwise we throw another dart. At each step, the probability that $X_{ij} = 1$ conditioned on the event that the game ends in that step is exactly $2/(j-i+1)$. Therefore, overall, the probability that $X_{ij} = 1$ is $2/(j-i+1)$.

In other words, for a given element i, it is compared to element $i+1$ with probability 1, to element $i+2$ with probability $2/3$, to element $i+3$ with probability $2/4$, to element $i+4$ with probability $2/5$ and so on. So, we have:

$$\mathbf{E}[X] = \sum_{i=1}^{n} 2\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n-i+1}\right).$$

The quantity $1 + 1/2 + 1/3 + \ldots + 1/n$, denoted H_n, is called the "nth harmonic number" and is in the range $[\ln n, 1 + \ln n]$ (this can be seen by considering the integral of $f(x) = 1/x$). Therefore,

$$\mathbf{E}[X] < 2n(H_n - 1) \leq 2n \ln n. \quad \blacksquare$$

3.5 Further Discussion

3.5.1 More linearity of expectation: a random walk stock market

Suppose there is a stock with the property that each day, it has a 50:50 chance of going either up or down by \$1, unless the stock is at 0 in which case it stays there. You start with \$m. Each day you can buy or sell as much as you want, until at the end of the year all your money is converted back into cash. What is the best strategy for maximizing your expected gain?

The answer is that no matter what strategy you choose, your expected gain by the end of the year is 0 (i.e., you expect to end with the same amount of money as you started). Let's prove that this is the case.

Define random variable X_t to be the gain of our algorithm on day t. Let X be the overall gain at the end of the year. Then,

$$X \quad = \quad X_1 + \ldots + X_{365}.$$

Notice that the X_t's can be highly dependent, based on our strategy. For instance, if our strategy is to pull all our money out of the stock market the moment that our wealth exceeds \$m, then X_2 depends strongly on the outcome of X_1. Nonetheless, by linearity of expectation,

$$\mathbf{E}[X] \quad = \quad \mathbf{E}[X_1] + \ldots + \mathbf{E}[X_{365}].$$

Finally, no matter how many shares s of stock we hold at time t, $\mathbf{E}[X_t|s] = 0$. So, using (3.3), whatever probability distribution over s is induced by our strategy, $\mathbf{E}[X_t] = 0$. Since this holds for every t, we have $\mathbf{E}[X] = 0$.

This analysis can be generalized to the case of gambling in a "fair casino". In a fair casino, there are a number of games with different kinds of payoffs, but each one has the property that your expected gain for playing it is zero. E.g., there might be a game where with probability 99/100 you lose but with probability 1/100 you win 99 times your bet. In that case, no matter what strategy you use for which game to play and how much to bet, the expected amount of money you will have at the end of the day is the same as the amount you had going in.

3.5.2 Yet another way to analyze quicksort: run it backwards

Here's another way to analyze quicksort — run the algorithm backwards. Actually, to do this analysis, it is better to think of a version of Quicksort that instead of being recursive, at each step it picks a random bucket in proportion to its size to work on next. The reason this version is nice is that if you imagine watching the pivots get chosen and where they would be on a sorted array, they are coming in completely at random. Looking at the algorithm run backwards, at a generic point in time, we have k pivots (producing $k+1$ buckets) and we "undo" one of our pivot choices at random, merging the two adjoining buckets. [The tricky part here is showing that this is really a legitimate way of looking at Quicksort in reverse.] The cost for an undo operation is the sum of the sizes of the two buckets joined (since this was the number of comparisons needed to split them). Notice that for each undo operation, if you sum the costs over all of the k possible pivot choices, you count each bucket twice (or once if it is the leftmost or rightmost) and get a total of $< 2n$. Since we are picking one of these k possibilities at random, the *expected* cost is $2n/k$. So, we get $\sum_k 2n/k = 2nH_n$.