Chapter 9

Randomized Algorithms

The theme of this chapter is ~~*madendrizo*~~ *randomized* *algorithms*. These are algorithms that make use of randomness in their computation. You might know of quicksort, which is efficient on average when it uses a random pivot, but can be bad for any pivot that is selected without randomness.

Analyzing randomized algorithms can be difficult, so you might wonder why randomization in algorithms is so important and worth the extra effort. Well it turns out that for certain problems randomized algorithms are simpler or faster than algorithms that do not use randomness. The problem of primality testing (PT), which is to determine if an integer is prime, is a good example. In the late 70s Miller and Rabin developed a famous and simple randomized algorithm for the problem that only requires polynomial work. For over 20 years it was not known whether the problem could be solved in polynomial work without randomization. Eventually a polynomial time algorithm was developed, but it is much more complicated and computationally more costly than the randomized version. Hence in practice everyone still uses the randomized version.

There are many other problems in which a randomized solution is simpler or cheaper than the best non-randomized solution. In this chapter, after covering the prerequisite background, we will consider some such problems. The first we will consider is the following simple problem:

> **Question:** How many comparisons do we need to find the top two largest numbers in a sequence of n distinct numbers?

Without the help of randomization, there is a trivial algorithm for finding the top two largest numbers in a sequence that requires about $2n - 3$ comparisons. We show, however, that if the order of the input is randomized, then the same algorithm uses only $n + O(\log n)$ comparisons in expectation (on average). This matches a more complicated deterministic version based on tournaments.

Randomization plays a particular important role in developing parallel algorithms, and analyzing such algorithms introduces some new challenges. In this chapter we will look at two randomized algorithms with significant parallelism: one for finding the k^{th} smallest element in a sequences, and the other is quicksort. In future chapters we will cover many other randomized algorithms.

In this book we will require that randomized algorithms always return the correct answer, but their costs (work and span) will depend on random choices. Such algorithms are sometimes called *Las Vegas algorithms*. Algorithms that run in a fixed amount of time, but may or may not return the corret answer, depending on random choices, are called *Monte Carlo* algorithms.

9.1 Discrete Probability: Let's Toss Some Dice

Not surprisingly analyzing randomized algorithms requires understanding some (discrete) probability. In this section we will cover a variety of ideas that will be useful in analyzing algorithms.

Expected vs. High Probability Bounds. In analyzing costs for a randomized algorithms there are two types of bounds that we will find useful: expected bounds, and high-probability bounds. **Expected bounds** basically tell us about the average case across all random choices used in the algorithm. For example if an algorithm has $O(n)$ expected work, it means that on average across all random choices it makes, the algorithm has $O(n)$ work. Once it a while, however, the work could be much larger, for example once in every $\sqrt{n}$ tries the algorithm might require $O(n^{3/2})$ work.

Exercise 9.1. *Explain why if an algorithm has $\Theta(n)$ expected (average) work it cannot be the case that once in every n or so tries the algorithm requires $\Theta(n^3)$ work?*

High-probability bounds on the other hand tell us that it is very unlikely that the cost will be above some bound. Typically the probability is stated in terms of the problem size n. For example, we might determine that an algorithm on n elements has $O(n)$ work with probability at least $1 - 1/n^5$. This means that only once in about n^5 tries will the algorithm require more than $O(n)$ work. High-probability bounds are typically stronger than expectation bounds.

Exercise 9.2. *Argue that if an algorithm has $O(\log n)$ span with probability at least $1 - 1/n^5$, and if its worst case span is $O(n)$, then it must also have $O(\log n)$ span in expectation.*

Expected bounds are often very convenient when analyzing work (or running time in traditional sequential algorithms). This is because of the linearity of expectations, discussed further below. This allows one to add expectations across the components of an algorithm to get the overall expected work. For example, if we have 100 students and they each take on average 2

Figure 9.1: Every year around the middle of April the Computer Science Department at Carnegie Mellon University holds an event called the "Random Distance Run". It is a running event around the track, where the official dice tosser rolls a dice immediately before the race is started. The dice indicates how many initial laps everyone has to run. When the first person is about to complete the laps, a second dice is thrown indicating how many more laps everyone has to run. Clearly, some understanding of probability can help one decide how to practice for the race and how fast to run at the start. Thanks to Tom Murphy for the design of the 2007 T-shirt.

hours to finish an exam, then the total time spent on average across all students (the work) will be $100 \times 2 = 200$ hours. This makes analysis reasonably easy since it is compositional. Indeed many textbooks that cover sequential randomized algorithms only cover expected bounds.

Unfortunately such composition does not work when taking a maximum, which is what we need to analyze span. Again if we had 100 students starting at the same time and they take 2 hours on average, we cannot say that the average maximum time across students (the span) is 2 hours. It could be that most of the time each student takes 1 hour, but that once on every 100 exams or so, each student gets hung up and takes 101 hours. The average for each student is $(99 \times 1 + 1 \times 101)/100 = 2$ hours, but on most exams with a hundred students one student will get hung up, so the expected maximum will be close to 100 hours, not 2 hours. We therefore cannot compose the expected span from each student by taking a maximum. However, if we have high-probability bounds we can do a sort of composition. Lets say we know that every student will finish in 2 hours with probability $1 - 1/n^5$, or equivalently that they will take more than two hours with probability $1/n^5$. Now lets say n students take the exam, what is the probability that any takes more than 2 hours? It is at most $1/n^4$, which is still very unlikely. We will come back to why when we talk about the union bound.

Because of these properties of summing vs. taking a maximum, in this book we often analyze work using expectation, but analyze span using high probability.

We now describe more formally the probability we require for analyzing randomized algorithms. We begin with an example:

Example 9.3. *Suppose we have two* fair *dice, meaning that each is equally likely to land on any of its six sides. If we toss the dice, what is the chance that their numbers sum to 4? You can probably figure out that the answer is*

$$\frac{\text{\# of outcomes that sum to 4}}{\text{\# of total possible outcomes}} = \frac{3}{36} = \frac{1}{12}$$

since there are three ways the dice could sum to 4 (1 and 3, 2 and 2, and 3 and 1), out of the $6 \times 6 = 36$ *total possibilities.*

Such throwing of dice is called a *probabilistic experiment* since it is an "experiment" (we can repeat it many time) and the outcome is probabilistic (each experiment might lead to a different outcome).

Discrete probability is really about counting, or possibly weighted counting. In the dice example we had 36 possibilities and a subset (3 of them) had the property we wanted. We assumed the dice were unbiased so all possibilities are equally likely. Indeed in general it is useful to consider all possible outcomes and then count how many of the outcomes match our criteria, or perhaps take an average of some value associated with each outcome (e.g. the sum of the values of the two dice). If the outcomes are not equally likely, then the average has to be weighted by the probability of each outcome. For these purposes we define the notion of a sample space (the set of all possible outcomes), primitive events (each element of the sample space, i.e., each possible outcome), events (subsets of the sample space), probability functions (the probability for each primitive event), and random variables (functions from the sample space to the real numbers indicating a value associated with the primitive event). We go over each in turn.

Sample Spaces and Events. A *sample space* Ω is an arbitrary and possibly infinite (but countable) set of possible outcomes of a probabilistic experiment. For the dice example, the sample space is the 36 possible outcomes of the dice. If we are using random numbers, the sample space corresponds to all possible values of the random numbers used. In the next section we analyze finding the top two largest elements in a sequence, by using as our sample space all permutations of the sequence.

A *primitive event* (or *sample point*) of a sample space Ω is any one of its elements, i.e. any one of the outcomes of the random experiment. An *event* is any subset of Ω and most often representing some property common to multiple primitive events. We typically denote events by capital letters from the start of the alphabet, e.g. A, B, C.

Probability Functions. A *probability function* $\mathbf{Pr} : \Omega \to [0, 1]$ is a function that maps each primitive event in the sample space to the probability of that primitive event. It must be the case that the probabilities add to one, i.e. $\sum_{e \in \Omega} \mathbf{Pr}\left[e\right] = 1$. The probability of an event A is simply the sum of the probabilities of its primitive events:

$$\mathbf{Pr}[A] = \sum_{e \in A} \mathbf{Pr}[e] \ .$$

If all probabilities are equal, all we have to do is figure out the size of the event (corresponding set) and divide it by $|\Omega|$. In general this might not be the case. We will use $(\Omega, \mathbf{Pr})$ to indicate the sample space along with its probability function.

Example 9.4. *For our example of throwing two dice, our* ***sample space*** *is:*

$$\Omega = \{(1,1),(1,2),\ldots,(2,1),\ldots,(6,6)\} \ ,$$

corresponding to every possible pair of values of the dice. Clearly $|\Omega| = 36$. *Having the first dice show up* 1 *and the second* 4 *is a* ***primitive event*** *since it corresponds to a single element* $(1,4)$ *of our sample space* Ω. *An example* ***event*** *is the "the first dice is 3", which corresponds to the set:*

$$E_1 = \{(d_1,d_2) \in \Omega \mid d_1 = 3\} = \{(3,1),(3,2),(3,3),(3,4),(3,5),(3,6)\} \ ,$$

or another ***event*** *is "the dice sum to 4", which corresponds to the set:*

$$E_2 = \{(d_1,d_2) \in \Omega \mid d_1 + d_2 = 4\} = \{(1,3),(2,2),(3,1)\} \ .$$

Assuming the dice are unbiased, the ***probability function*** *over our sample space is* $\mathbf{Pr}[x] = 1/36$ *and the* ***probability of the event*** E_1 *(that the first dice is 3) is:*

$$\mathbf{Pr}[E_1] = \sum_{e \in E_1} \mathbf{Pr}[e] = \frac{6}{36} = \frac{1}{6} \ .$$

If, instead, each of our dice were biased so the probability that it shows up with a particular value is proportional to the value, then we would have the ***probability function*** $\mathbf{Pr}[(x,y)] = \frac{x}{21} \times \frac{y}{21}$, *and the* ***probability of the event*** E_2 *(that the dice add to 4) is:*

$$\mathbf{Pr}[E_2] = \sum_{e \in E_2} \mathbf{Pr}[e] = \frac{1 \times 3 + 2 \times 2 + 3 \times 1}{21 \times 21} = \frac{10}{441} \ .$$

Random Variables. A *random variable* X is a real-valued function on Ω, thus having type $X : \Omega \to \mathbb{R}$, i.e., it assigns a real number to each primitive event. For a sample space there can be many random variables each keeping track of some quantity of a probabilistic experiment. A random variable is called an *indicator random variable* if it takes on the value 1 when some condition is true and 0 otherwise—in particular for a predicate $p : \Omega \to \texttt{bool}$ we have s *indicator random variable*:

$$Y(e) = \textbf{if } p(e) \textbf{ then } 1 \textbf{ else } 0 \ .$$

For a random variable X and a value $a \in \mathbb{R}$, we use the following shorthand for the event corresponding to X equaling a:

$$\{X = a\} \equiv \{\omega \in \Omega \mid X(\omega) = a\} \ .$$

We typically denote random variables by capital letters from the end of the alphabet, e.g. X, Y Z.

Example 9.5. *For throwing two dice, a useful random variables is the sum of the two dice:*

$$X(d_1, d_2) = d_1 + d_2 \ ,$$

and a useful ***indicator random variable*** *corresponds to getting doubles (the two dice have the same value):*

$$Y(d_1, d_2) = \textbf{if } (d_1 = d_2) \textbf{ then } 1 \textbf{ else } 0 \ .$$

Using our shorthand, the event $\{X = 4\}$ corresponds to the event "the dice sum to 4".

We note that the term random variable might seem counter intuitive since it is actually a function not a variable, and it is not really random at all since it is a well defined deterministic function on the sample space. However if you think of it in conjunction with the random experiment that selects a primitive element, then it is a variable that takes on its value based on a random process.

Expectation. We are now ready to talk about expectation. The *expectation* of a random variable X given a sample space $(\Omega, \mathbf{Pr})$ is the sum of the random variable over the primitive events weighted by their probability, specifically:

$$\mathop{\mathbf{E}}_{\Omega,\mathbf{Pr}}[X] = \sum_{e \in \Omega} X(e) \cdot \mathbf{Pr}[e] \ .$$

One way to think of expectation is as the following higher order function that takes as arguments the random variable, along with the sample space and corresponding probability function:

$$\mathbf{E}\ (\Omega, \mathbf{Pr})\ X = \mathtt{reduce}\ +\ 0\ \left\{X(e) \times \mathbf{Pr}[e] : e \in \Omega\right\}$$

In the notes we will most often drop the $(\Omega, \mathbf{Pr})$ subscript on $\mathbf{E}$ since it is clear from the context.

We note that it is often not practical to compute expectations directly since the sample space can be exponential in the size of the input, or even countably infinite. We therefore resort to various shortcuts. For example we can also calculate expectations by considering each value of a random variable, weighting it by its probability, and summing:

$$\mathop{\mathbf{E}}_{\Omega,\mathbf{Pr}}[X] = \sum_{a \in \text{range of } X} a \cdot \mathbf{Pr}[\{X = a\}] \ .$$

Also, the expectation of an indicator random variable Y is the probability that the associated predicate p is true (i.e. that $Y = 1$):

$$\begin{aligned}\mathbf{E}[Y] &= 0 \cdot \mathbf{Pr}[\{Y = 0\}] + 1 \cdot \mathbf{Pr}[\{Y = 1\}] \\ &= \mathbf{Pr}[\{Y = 1\}] \ .\end{aligned}$$

Example 9.6. *Assuming unbiased dice (*$\mathbf{Pr}[(d_1, d_2)] = 1/36$*), the expectation of the random variable* X *representing the sum of the two dice is:*

$$\mathop{\mathbf{E}}_{\Omega,\mathbf{Pr}}[X] = \sum_{(d_1,d_2)\in\Omega} X(d_1, d_2) \times \frac{1}{36} = \sum_{(d_1,d_2)\in\Omega} \frac{d_1 + d_2}{36} = 7 \ .$$

If we bias the coins so that for each dice the probability that it shows up with a particular value is proportional to the value, we have $\mathbf{Pr}'[(d_1, d_2)] = (d_1/21) \times (d_2/21)$ *and:*

$$\mathop{\mathbf{E}}_{\Omega,\mathbf{Pr}'}[X] = \sum_{(d_1,d_2)\in\Omega} \left((d_1 + d_2) \times \frac{d_1}{21} \times \frac{d_2}{21}\right) = 8\,\frac{2}{3} \ ,$$

although that last sum is a bit messy.

Independence. Two events A and B are *independent* if the occurrence of one does not affect the probability of the other. This is true if and only if $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$. The probability $\mathbf{Pr}[A \cap B]$ is called the *joint probability* of the events A and B and is sometimes written $\mathbf{Pr}[A, B]$. When we have multiple events, we say that $A_1, \ldots, A_k$ are *mutually independent* if and only if for any non-empty subset $I \subseteq \{1, \ldots, k\}$,

$$\mathbf{Pr}\left[\bigcap_{i\in I} A_i\right] = \prod_{i\in I} \mathbf{Pr}[A_i] \ .$$

Example 9.7. *For two dice, the events* $A = \{(d_1, d_2) \in \Omega \mid d_1 = 1\}$ *(the first dice is 1) and* $B = \{(d_1, d_2) \in \Omega \mid d_2 = 1\}$ *(the second dice is 1) are independent since*

$$\begin{aligned} & \mathbf{Pr}[A] \times \mathbf{Pr}[B] &= \tfrac{1}{6} \times \tfrac{1}{6} &= \tfrac{1}{36} \\ = \; & \mathbf{Pr}[A \cap B] &= \mathbf{Pr}[\{(1, 1)\}] &= \tfrac{1}{36} \ . \end{aligned}$$

However, the event $C \equiv \{X = 4\}$ *(the dice add to 4) is not independent of* A *since*

$$\begin{aligned} & \mathbf{Pr}[A] \times \mathbf{Pr}[C] &= \tfrac{1}{6} \times \tfrac{3}{36} &= \tfrac{1}{72} \\ \neq \; & \mathbf{Pr}[A \cap C] &= \mathbf{Pr}[\{(1, 3)\}] &= \tfrac{1}{36} \ . \end{aligned}$$

A *and* C *are not independent since the fact that the first dice is 1 increases the probability they sum to* 4 *(from* $\frac{1}{12}$ *to* $\frac{1}{6}$*).*

Two random variables X and Y are independent if fixing the value of one does not affect the probability distribution of the other. This is true if and only if for every a and b the events $\{X = a\}$ and $\{Y = b\}$ are independent. In our two dice example, a random variable X representing the value of the first dice and a random variable Y representing the value of the second dice are independent. However X is not independent of a random variable Z representing the sum of the values of the two dice.

Exercise 9.8. *For throwing two dice, are the two random variables X and Y in Example 9.5 independent.*

Composing Expectations One of the most important and useful theorems in probability is *linearity of expectations*. It says that given two random variables X and Y, $\mathbf{E}[X] + \mathbf{E}[Y] = \mathbf{E}[X + Y]$. If we write this out based on the definition of expectations we get:

$$\sum_{e\in\Omega} \mathbf{Pr}[e]\, X(e) + \sum_{e\in\Omega} \mathbf{Pr}[e]\, Y(e) = \sum_{e\in\Omega} \mathbf{Pr}[e]\, (X(e) + Y(e))$$

The algebra to show this is true is straightforward. The linearity of expectations is very powerful often greatly simplifying analysis.

Example 9.9. *In Example 9.6 we analyzed the expectation on X, the sum of the two dice, by summing across all 36 primitive events. This was particulary messy for the biased dice. Using linearity of expectations, we need only calculate the expected value of each dice, and then add them. Since the dice are the same, we can in fact just multiply by two. For example for the biased case, assuming X_1 is the value of one dice:*

$$\mathop{\mathbf{E}}_{\Omega,\mathbf{Pr}'}[X] = 2 \mathop{\mathbf{E}}_{\Omega,\mathbf{Pr}'}[X_1] = 2\times \sum_{d\in\{1,2,3,4,5,6\}} d\times\frac{d}{21} = 2\times\frac{1+4+9+16+25+36}{21} = 8\,\frac{2}{3}\,.$$

So we can add expectations, but can we multiply them? In particular is the following true: $\mathbf{E}[X]\times\mathbf{E}[Y] = \mathbf{E}[X \times Y]$? It turns out it is true when X and Y are independent, but otherwise it is generally not true. To see that it is true for independent random variables we have (we assume a and b range over the values of X and Y respectively):

$$\begin{aligned}
\mathbf{E}[X] \times \mathbf{E}[Y] &= \left(\sum_a a\,\mathbf{Pr}[\{X = a\}]\right)\left(\sum_b b\,\mathbf{Pr}[\{Y = b\}]\right)\\
&= \sum_a \sum_b (ab\,\mathbf{Pr}[\{X = a\}]\,\mathbf{Pr}[\{Y = b\}])\\
&= \sum_a \sum_b (ab\,\mathbf{Pr}[\{X = a\} \cap \{Y = b\}]) \qquad \text{due to independence}\\
&= \mathbf{E}[X \times Y]
\end{aligned}$$

Example 9.10. *Suppose we toss n coins, where each coin has a probability p of coming up heads. What is the expected value of the random variable X denoting the total number of heads?*

Solution I: We will apply the definition of expectation directly. This will rely on some messy algebra and useful equalities you might or might not know, but don't fret since this not the way we suggest you do it.

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{k=0}^{n} k \cdot \mathbf{Pr}[\{X = k\}] \\
&= \sum_{k=1}^{n} k \cdot p^k (1-p)^{n-k} \binom{n}{k} \\
&= \sum_{k=1}^{n} k \cdot \frac{n}{k} \binom{n-1}{k-1} p^k (1-p)^{n-k} && \text{[because } \binom{n}{k} = \frac{n}{k}\binom{n-1}{k-1} \text{]} \\
&= n \sum_{k=1}^{n} \binom{n-1}{k-1} p^k (1-p)^{n-k} \\
&= n \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j+1} (1-p)^{n-(j+1)} && \text{[because } k = j+1 \text{]} \\
&= np \sum_{j=0}^{n-1} \binom{n-1}{j} p^{j} (1-p)^{(n-1)-j} \\
&= np(p + (1-p))^n && \text{[Binomial Theorem]} \\
&= np
\end{aligned}$$

That was pretty tedious :(

Solution II: We'll use linearity of expectations. Let $X_i = \mathbb{I}\{i\text{-th coin turns up heads}\}$. That is, 1 if the i-th coin turns up heads and 0 otherwise. Clearly, $X = \sum_{i=1}^{n} X_i$. So then, by linearity of expectations,

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbf{E}[X_i].$$

What is the probability that the i-th coin comes up heads? This is exactly p, so $\mathbf{E}[X] = 0 \cdot (1-p) + 1 \cdot p = p$, which means

$$\mathbf{E}[X] = \sum_{i=1}^{n} \mathbf{E}[X_i] = \sum_{i=1}^{n} p = np.$$

Example 9.12. *A coin has a probability p of coming up heads. What is the expected value of Y representing the number of flips until we see a head? (The flip that comes up heads counts too.)*

Solution I: We'll directly apply the definition of expectation:

$$\begin{aligned}\mathbf{E}[Y] &= \sum_{k\geq 1} k(1-p)^{k-1}p \\ &= p\sum_{k=0}^{\infty}(k+1)(1-p)^k \\ &= p\cdot\frac{1}{p^2} && \text{[by Wolfram Alpha, though you should be able to do it.]} \\ &= 1/p\end{aligned}$$

Solution II: Alternatively, we'll write a recurrence for it. As it turns out, we know that with probability p, we'll get a head and we'll be done—and with probability $1-p$, we'll get a tail and we'll go back to square one:

$$\mathbf{E}[Y] = p\cdot 1 + (1-p)\Big(1+\mathbf{E}[Y]\Big) = 1+(1-p)\,\mathbf{E}[Y] \implies \mathbf{E}[Y] = 1/p.$$

by solving for $\mathbf{E}[Y]$ in the above equation.

For example, the expected value of the product of the values on two (independent) dice is therefore $3.5\times 3.5 = 12.25$.

So we can add expectations unconditionally, and multiply them when the random variables are independent, but what about taking the maximum (or minimum) of expectations. In particular is the following true: $\max\{\mathbf{E}[X], \mathbf{E}[Y]\} = \mathbf{E}[\max\{X,Y\}]$? The answer in this case is no, even if the random variables are independent.

Exercise 9.11. *What is the expected maximum value of throwing two dice?*

The Union Bound. The union bound, also known as Boole's inequality, is a simple way to get an upper bound on the probability of any of a collection of events happening. Specifically for a collection of events $A_1, A_2, \ldots, A_n$ the bound is:

$$\mathbf{Pr}\left[\bigcup_{1\leq i\leq n} A_i\right] \leq \sum_{i=1}^{n}\mathbf{Pr}[A_i]$$

This bound is true unconditionally, whether the events are independent or not. To see why the bound holds we note that the primitive events in the union on the left are all included in the sum on the right (since the union comes from the same set of events). In fact they might be included multiple times in the sum on the right, hence the inequality. In fact sum on the right could add to more than one, in which case the bound is not useful. However, the union bound is very useful in generating high-probability bounds, when the probability of each of n events is very low, e.g. $1/n^5$ and the sum remains very low, e.g. $1/n^4$.

Markov's Inequality. Consider a non-negative random variable X. We can ask how much larger can X's maximum value be than its expected value. With small probability it can be arbitrarily much larger. However, since the expectation is taken by averaging X over all outcomes, and it cannot take on negative values, X cannot take on a much larger value with significant probability. If it did it would contribute too much to the sum.

Question 9.13. *Can more than half the class do better than twice the average score on the midterm?*

More generally X cannot be a multiple of β larger than its expectation with probability greater than $1/\beta$. This is because this part on its own would contribute more than $\beta\,\mathbf{E}[X] \times \frac{1}{\beta} = \mathbf{E}[X]$ to the expectation, which is a contradiction. This gives us for a non-negative random variable X the inequality:

$$\mathbf{Pr}\left[X \geq \beta\,\mathbf{E}[X]\right] \leq \frac{1}{\beta}$$

or equivalently (by substituting $\beta = \alpha/\,\mathbf{E}[X]$),

$$\mathbf{Pr}[X \geq \alpha] \leq \frac{\mathbf{E}[X]}{\alpha}$$

which is known as Markov's inequality.

High-Probability Bounds. As discussed at the start of this section, it can be useful to argue that some bound on the cost of an algorithm is true with "high probability", even if not always true. Unlike expectations, high-probability bounds allow us to compose the span of components of algorithms when run in parallel. For a problem of size n we say that some property is true with high probability if it is true with probability $1 - 1/n^k$ for some constant $k > 1$. This means the inverse is true with very small probability $1/n^k$. It is often easier to work with the inverse. Now if we had n experiments each with inverse probability $1/n^k$ we can use the union bound to argue that the total inverse probability is $n \cdot 1/n^k = 1/n^{k-1}$. This means that for $k > 2$ the probability $1 - 1/n^{k-1}$ is still true with high probability.

We now consider how to derive high-probability bounds. We can start by considering the probability that when flipping n unbiased coins all of them come up heads. This is simply $1/2^n$

since the sample space is of size $1/2^n$, each primitive event has equal probability, and only one of the primitive events has all heads. Certainly asymptotically $1/2^n < 1/n^k$ so the fact that we will not flip n heads is true with high probability. Even if there are n people each flipping n coins, the probability that no one will flip all heads is still high (at least $1 - n/2^n$ using the union bound).

Another way to prove high-probability bounds is to use Markov's inequality. As we will see in Section 9.3 it is sometimes possible to bound the expectation of a random variable X to be very small, e.g. $\mathbf{E}[X] \leq 1/n^k$. Now using Markov's inequality we have that the probability that $X \geq 1$ is at most $1/n^k$.

9.2 Finding The Two Largest

The max-two problem is to find the two largest elements from a sequence of n (unique) numbers. Lets consider the following simple algorithm for the problem.

Algorithm 9.14 (Iterative Max-Two)**.**

```
 1  function max2(S) =
 2  let
 3     function replace((m_1, m_2), v) =
 4        if v ≤ m_2 then (m_1, m_2)
 5        else if v ≤ m_1 then (m_1, v)
 6        else (v, m_1)
 7     val start = if S_1 ≥ S_2 then (S_1, S_2) else (S_2, S_1)
 8  in
 9     iter replace start S⟨3, ..., n⟩
10  end
```

We assume S is indexed from 1 to n. In the following analysis, we will be meticulous about constants. The naïve algorithm requires up to $1 + 2(n-2) = 2n - 3$ comparisons since there is one comparison to compute `start` each of the $n - 2$ `replace`s requires up to two comparisons. On the surface, this may seem like the best one can do. Surprisingly, there is a divide-and-conquer algorithm that uses only about $3n/2$ comparisons (exercise to the reader). More surprisingly still is the fact that it can be done in $n + O(\log n)$ comparisons. But how?

Puzzle: How would you solve this problem using only $n + O(\log n)$ comparisons?

A closer look at the analysis above reveals that we were pessimistic about the number of comparisons; not all elements will get past the "if" statement in Line 4; therefore, only some of the elements will need the comparison in Line 5. But we didn't know how many of them, so we analyzed it in the worst possible scenario.

Let's try to understand what's happening better by looking at the worst-case input. Can you come up with an instance that yields the worst-case behavior? It is not difficult to convince yourself that there is a sequence of length n that causes this algorithm to make $2n - 3$ comparisons. In fact, the instance is simple: an increasing sequence of length n, e.g., $\langle 1, 2, 3, \ldots, n \rangle$. As we go from left to right, we find a new maximum every time we counter a new element—this new element gets compared in both Lines 4 and 5.

But perhaps it's unlikely to get such a nice—but undesirable—structure if we consider the elements in random order. With only 1 in $n!$ chance, this sequence will be fully sorted. You can work out the probability that the random order will result in a sequence that looks "approximately" sorted, and it would not be too high. Our hopes are high that *we can save a lot of comparisons in Line 5 by considering elements in random order.*

The algorithm we'll analyze is the following. On input a sequence T of n elements:

1. Let $S = \texttt{permute}(T, \pi)$, where π is a random permutation (i.e., we choose one of the $n!$ permutations).
2. Run algorithm *max2* on S.

Remarks: We don't need to explicitly construct S. We'll simply pick a random element which hasn't been considered and consider that element next until we are done looking at the whole sequence. For the analysis, it is convenient to describe the process in terms of S.

In reality, the performance difference between the $2n - 3$ algorithm and the $n - 1 + 2\log n$ algorithm is unlikely to be significant—unless the comparison function is super expensive. For most cases, the $2n - 3$ algorithm might in fact be faster to due better cache locality.

The point of this example is to demonstrate the power of randomness in achieving something that otherwise seems impossible—more importantly, the analysis hints at why on a typical "real-world" instance, the $2n-3$ algorithm does much better than what we analyzed in the worst case (real-world instances are usually not adversarial).

Analysis

After applying the random permutation we have that our sample space Ω corresponds to each permutation. Since there are $n!$ permutations on a sequence of length n and each has equal probability, we have $|\Omega| = n!$ and $\textbf{Pr}[e] = 1/n!, e \in \Omega$. However, as we will see, we do not really need to know this, all we need to know is what fraction of the sample space obeys some property.

Let i be the position in S (indexed from 1 to n). Now let X_i be an indicator random variable denoting whether Line 5 and hence its comparison gets executed for the value at S_i (i.e., Recall that an indicator random variable is actually a function that maps each primitive event (each permutation in our case) to 0 or 1. In particular given a permutation, it returns 1 iff for that permutation the comparison on Line 5 gets executed on iteration i. Lets say we want to now compute the total number of comparisons. We can define another random variable (function) Y that for any permutation returns the total number of comparison the algorithm takes on that permutation. This can be defined as:

$$Y(e) = \underbrace{1}_{\text{Line } 7} + \underbrace{n-2}_{\text{Line } 4} + \underbrace{\sum_{i=3}^{n} X_i(e)}_{\text{Line } 5}$$

It is common, however, to use the shorthand notation

$$Y = 1 + (n-2) + \sum_{i=3}^{n} X_i$$

where the argument e and function definition is implied.

We are interested in computing the expected value of Y, that is $\mathbf{E}[Y] = \sum_{e \in \Omega} \mathbf{Pr}[e] Y(e)$. By linearity of expectation, we have

$$\begin{aligned} \mathbf{E}[Y] &= \mathbf{E}\left[1 + (n-2) + \sum_{i=3}^{n} X_i\right] \\ &= 1 + (n-2) + \sum_{i=3}^{n} \mathbf{E}[X_i]. \end{aligned}$$

Our tasks therefore boils down to computing $\mathbf{E}[X_i]$ for $i = 3, \ldots, n$. To compute this expectation, we ask ourselves: *What is the probability that $S_i > m_2$?* A moment's thought shows that the condition $S_i > m_2$ holds exactly when S_i is either the largest element or the second largest element in $\{S_1, \ldots, S_i\}$. So ultimately we're asking: what is the probability that S_i is the largest or the second largest element in randomly-permuted sequence of length i?

To compute this probability, we note that each element in the sequence is equally likely to be anywhere in the permuted sequence (we chose a random permutation. In particular, if we look at the k-th largest element, it has $1/i$ chance of being at S_i. (You should also try to work it out using a counting argument.) Therefore, the probability that S_i is the largest or the second largest element in $\{S_1, \ldots, S_i\}$ is $\frac{1}{i} + \frac{1}{i} = \frac{2}{i}$, so

$$\mathbf{E}[X_i] = 1 \cdot \tfrac{2}{i} = 2/i.$$

Plugging this into the expression for $\mathbf{E}[Y]$, we get

$$\begin{aligned}
\mathbf{E}[Y] &= 1 + (n-2) + \sum_{i=3}^{n} \mathbf{E}[X_i] \\
&= 1 + (n-2) + \sum_{i=3}^{n} \frac{2}{i} \\
&= 1 + (n-2) + 2\Big(\tfrac{1}{3} + \tfrac{1}{4} + \dots \tfrac{1}{n}\Big) \\
&= n - 4 + 2\Big(1 + \tfrac{1}{2} + \tfrac{1}{3} + \tfrac{1}{4} + \dots \tfrac{1}{n}\Big) \\
&= n - 4 + 2H_n,
\end{aligned}$$

where H_n is the n-th Harmonic number. But we know that $H_n \leq 1 + \log_2 n$, so we get $\mathbf{E}[Y] \leq n - 2 + 2\log_2 n$. We could also use the following sledgehammer:

As an aside, the Harmonic sum has the following nice property:

$$H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n + \gamma + \varepsilon_n,$$

where γ is the Euler-Mascheroni constant, which is approximately $0.57721\cdots$, and $\varepsilon_n \sim \frac{1}{2n}$, which tends to 0 as n approaches ∞. This shows that the summation and integral of $1/i$ are almost identical (up to a small adative constant and a low-order vanishing term).

9.3 Finding The k^{th} Smallest Element

Consider the following problem:

Problem 9.15 (The k^{th} Smallest Element (KS))**.** *Given an α `sequence`, S, an integer $k, 0 \leq k < |S|$, and a comparison $<$ defining a total ordering over α, return $(\texttt{sort}_<(S))_k$, where `sort` is the standard sorting problem.*

This problem can obviously be implemented using a sort, but this would require $O(n \log n)$ work (we assume the comparison takes constant work). Our goal is to do better. In particular we would like to achieve linear work, while still achieving $O(\log^2 n)$ span. Here's where the power of randomization gives the following simple algorithm.

Algorithm 9.16 (contracting k^th smallest)**.**

```
function kthSmallest($(k, S)$) = let
    val $p = S_0$
    val $L = \langle\, x \in S \mid x < p \,\rangle$
    val $R = \langle\, x \in S \mid x > p \,\rangle$
in
    if $(k < |L|)$ then kthSmallest($(k, L)$)
    else if $(k < |S| - |R|)$ then $p$
    else kthSmallest($(k - (|S| - |R|), R)$)
```

This algorithm is similar to quicksort but instead of recursing on both sides, it only recurses on one side. Basically it figures out in which side the k^{th} smallest must be in, and just explores that side. When exploring the right side, R, the k needs to be adjusted by since all elements less or equal to the pivot p are being thrown out: there are $|S| - |R|$ such elements. The algorithm is based on contraction.

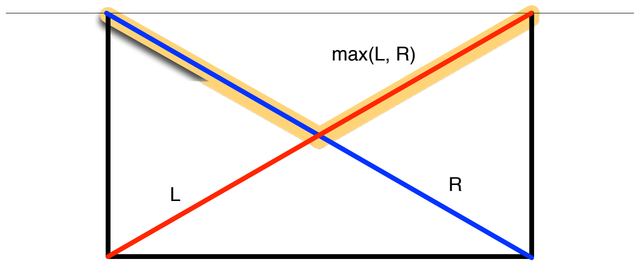
As written the algorithm picks the first key instead of a random key. As with the two-max problem, we can add randomness by first randomly permuting a sequence T to generate S and then applying `kthSmallest` on S. You should convince yourself that this is equivalent to randomly picking a pivot at each step of contraction.

We now analyze the work and span of this algorithm. Let $X = \max\{|L|, |R|\}/|S|$, which is the fractional size of the larger side. Notice that X is an upper bound on the fractional size of the side the algorithm actually recurses into. Now since lines 3 and 4 are simply two `filter` calls, we have the following recurrences:

$$\begin{array}{lcl} W(n) & \leq & W(X \cdot n) + O(n) \\ S(n) & \leq & S(X \cdot n) + O(\log n) \end{array}$$

Let's first look at the work recurrence. Specifically, we are interested in $\mathbf{E}\left[W(n)\right]$. First, let's try to get a sense of what happens in expectation.

What is the value of $\mathbf{E}\left[X\right]$? To understand this we note that all pivots are equally likely. We can then draw the following plot of the size of L and size of R as a function of where the pivot belongs in the sorted order of S.



If the pivot is at the start then L is empty and $|R| = |S| - 1$, and if the pivot is at the end then R is empty and $|L| = |S| - 1$. The probability that we land on a point on the x axis is $1/n$, so

$$\mathbf{E}[X] = \frac{1}{n}\sum_{i=0}^{n-1} \max\{i, n-i-1\}/n \leq \frac{1}{n}\sum_{j=n/2}^{n-1} \frac{2}{n}\cdot j \leq \frac{3}{4}$$

(Recall that $\sum_{i=a}^{b} i = \frac{1}{2}(a+b)(b-a+1)$.)

This computation tells us that in expectation, X is a constant fraction smaller than 1, so intuitively in calculating the work we should have a nice geometrically decreasing sum that adds up to $O(n)$. It is not quite so simple, however, since the constant fraction is only in expectation. It could be we are unlucky for a few contraction steps and the sequences size hardly goes down at all. How do we deal with analyzing this. We will cover other algorithms on graphs that have the same property, i.e. that the size goes down by an expected constant factor on each contraction step. The following theorem shows that even if we are unlucky on some steps, the expected size will indeed go down geometrically. Together with the linearity of expectations this will allow us to bound the work.

Theorem 9.17. *Starting with size n, the expected size of S in algorithm `kthSmallest` after i recursive calls is $\left(\frac{3}{4}\right)^i n$.*

Proof. Let Y_i be the random variable representing the size of the result after step (recursive call) i, and let X_i be the random variable representing the fraction of elements that are kept on the i^{th} step, giving $Y_i = n\prod_{j=1}^{i} X_j$. Since we pick the pivot independently on each step, the X_j are independent, allowing us to take advantage of the fact that with independence the product of expectations of random variables is equal to the expectation of their products. This gives:

$$\mathbf{E}[Y_i] = \mathbf{E}\left[n\prod_{j=1}^{i} X_j\right] = n\prod_{j=1}^{i}\mathbf{E}[X_j] \leq \left(\frac{3}{4}\right)^i n$$

□

The work at each step is linear, which we can write as $W_{\mathtt{contract}}(n) \leq k_1 n + k_2$, so we can now bound the work by summing the work across levels, giving

$$\begin{aligned}
\mathbf{E}[W_{\mathtt{kthSmallest}}(n)] &\leq \sum_{i=0}^{n}(k_1 n\left(\frac{3}{4}\right)^i + k_2) \\
&\leq k_1 n\left(\sum_{i=0}^{n}\left(\frac{3}{4}\right)^i\right) + k_2 n \\
&\leq 4k_1 n + k_2 n \\
&\in O(n)
\end{aligned}$$

Note that we summed across n steps. This is because we know the algorithm cannot run for more than n steps since each step removes at least one element, the pivot.

We now use Theorem 9.17 to bound the number of steps taken by `kthSmallest` to much better than n using high probability bounds. This will allow us to bound the span of the algorithm. Consider step $i = 10\log_2 n$. In this case we have the expected size upper bounded by $n\left(\frac{3}{4}\right)^{10\log_2 n}$, which with a little math is the same as $n \times n^{-10\log_2(4/3)} \approx n^{-3.15}$. We can now use Markov's inequality and observe that if the expected size is at most $n^{-3.15}$ then the probability of having size at least 1 (if less than 1 then the algorithm is done) is bounded by:

$$\mathbf{Pr}\left[Y_{10\log_2 n} \geq 1\right] \leq E[Y_{10\log_2 n}]/1 = n^{-3.15}$$

This is a high probability bound as discussed at the start of Section 9.1. By increasing 10 to 20 we can decrease the probability to $n^{-7.15}$, which is extremely unlikely: for $n = 10^6$ this is 10^{-42}. We have therefore show that the number of steps is $O(\log n)$ with high probability. Each step has span $O(\log n)$ so the overall span is $O(\log^2 n)$ with high probability.

In summary, we have shown than the `kthSmallest` algorithm on input of size n does $O(n)$ work in expectation and has $O(\log^2 n)$ span with high probability. As mentioned at the start of Section 9.1 we will typically be analyzing work using expectation and span using high probability.

Exercise 9.18. *Show that the high probability bounds on span for* `kthSmallest` *imply equivalent bounds in expectation.*

9.4 Quicksort

You have surely seen quicksort before. The purpose of this section is to analyze the work and span of quicksort. In later chapters we will see that the analysis of quicksort presented here is is effectively identical to the analysis of a certain type of balanced tree called Treaps. It is also the same as the analysis of "unbalanced" binary search trees under random insertion.

Quicksort is one of the earliest and most famous algorithms. It was invented and analyzed by Tony Hoare around 1960. This was before the big-O notation was used to analyze algorithms. Hoare invented the algorithm while an exchange student at Moscow State University while studying probability under Kolmogorov—one of the most famous researchers in probability theory. The analysis we will cover is different from what Hoare used in his original paper, although we will mention how he did the analysis. It is interesting that while Quicksort is often used as an quintessential example of a recursive algorithm, at the time, no programming language supported recursion and Hoare spent significant space in his paper explaining how to simulate recursion with a stack.

Consider the quicksort algorithm given in Algorithm 9.19. In this algorithm, we intentionally leave the pivot-choosing step unspecified because the property we are discussing holds regardless of the choice of the pivot.

Algorithm 9.19 (Quicksort)**.**

```
function sort(S) =
if |S| = 0 then S
else let
        val p = pick a pivot from S
        val S_1 = ⟨ s ∈ S | s < p ⟩
        val S_2 = ⟨ s ∈ S | s = p ⟩
        val S_3 = ⟨ s ∈ S | s > p ⟩
        val (R_1, R_3) = (sort(S_1) || sort(S_3))
     in
        append(R_1, append(S_2, R_3))
     end
```

Question 9.20. *Is there parallelism in quicksort?*

There is plenty of parallelism in this version quicksort.[1] There is both parallelism due to the two recursive calls and in the fact that the filters for selecting elements greater, equal, and less than the pivot can be parallel.

Note that each call to quicksort either makes no recursive calls (the base case) or two recursive calls. The call tree is therefore binary. We will often find it convenient to map the run of a quicksort to a binary-search tree (BST) representing the recursive calls along with the pivots chosen. We will sometimes refer to this tree as the *call tree* or *pivot tree*. We will use this call-tree representation to reason about the properties of quicksort, e.g., the comparisons performed, its span. An example is shown in Example 9.21.

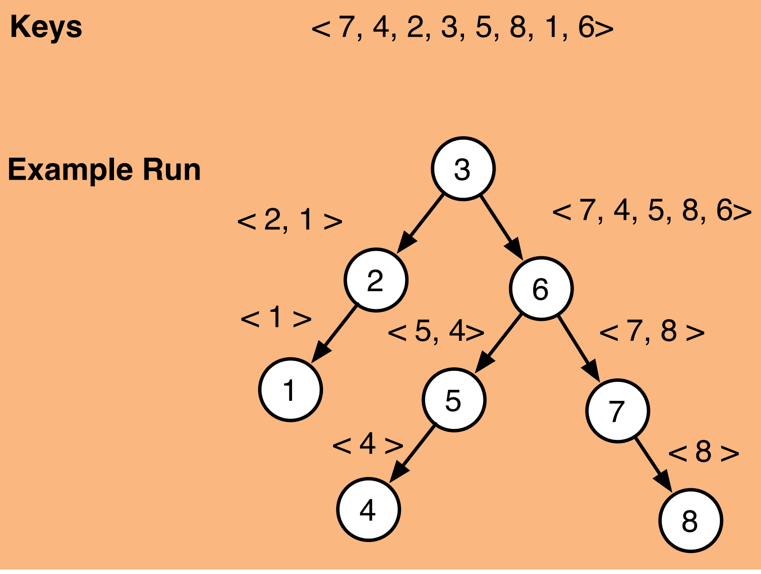
Let's consider some strategies for picking a pivot.

- *Always pick the first element:* If the sequence is sorted in increasing order, then picking the first element is the same as picking the smallest element. We end up with a lopsided recursion tree of depth n. The total work is $O(n^2)$ since $n - i$ keys will remain at level i and hence we will do $n - i - 1$ comparisons at that level for a total of $\sum_{i=0}^{n-1}(n - i - 1)$. Similarly, if the sequence is sorted in decreasing order, we will end up with a recursion tree that is lopsided in the other direction. In practice, it is not uncommon for a sort function input to be a sequence that is already sorted or nearly sorted.

- *Pick the median of three elements:* Another strategy is to take the first, middle, and the last elements and pick the median of them. For sorted lists the split is even, so each side contains half of the original size and the depth of the tree is $O(\log n)$. Although this

[1]This differs from Hoare's original version which sequentially partitioned the input by the pivot using two fingers that moved from each end and swapping two keys whenever a key was found on the left greater than the pivot and on the right less than the pivot.

Example 9.21. *An example run of quicksort along with its pivot tree.*



strategy avoids the pitfall with sorted sequences, it is still possible to be unlucky, and in the worst-case the costs and tree depth are the same as the first strategy. This is the strategy used by many library implementations of quicksort. Can you think of a way to slow down a quicksort implementation that uses this strategy by picking an adversarial input?

- *Pick an element randomly:* It is not immediately clear what the depth of this is, but intuitively, when we choose a random pivot, the size of each side is not far from $n/2$ in expectation. This doesn't give us a proof but it gives us hope that this strategy will result in a tree of depth $O(\log n)$ in expectation or with high probability. Indeed, picking a random pivot gives us expected $O(n \log n)$ work and $O(\log^2 n)$ span for quicksort and an expected $O(\log n)$-depth tree, as we will show.

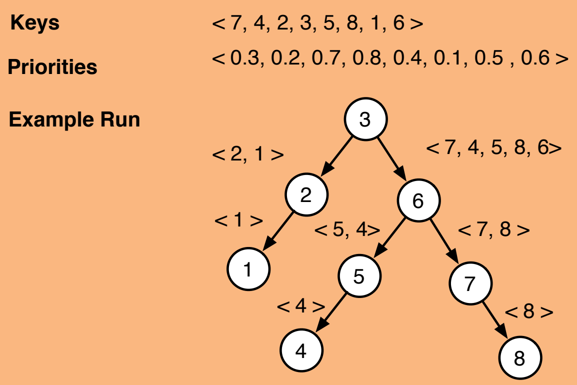
Analysis of Quicksort

We now analyze the algorithm that selects a uniformly randomly chosen key as the pivot. For the analysis, we assume a version of quicksort that compares the pivot p to each key in S once (instead of 3 times, once to generate each of S_1, S_2, and S_3).

Exercise 9.22. *Rewrite the quicksort algorithm so that it takes a comparison function* `cmp` : $\alpha \times \alpha \rightarrow$ `order` *and a sequence of type* α *seq, and only uses the comparison once when comparing the pivot with each key. The type* `order` *is the set* $\{$`Less`, `Equal`, `Less`$\}$.

In the analysis, for picking the pivots, we are going to use a priority-based selection technique. Before the start of the algorithm, we'll pick for each key a random priority uniformly at random from the real interval $[0, 1]$ such that each key has a unique priority. We then pick in Line **??** the key with the highest priority. Notice that once the priorities are decided, the algorithm is completely deterministic.

Example 9.23. *Quicksort with priorities and its call tree, which is a binary-search-tree, illustrated.*



Exercise 9.24. *Convince yourself that the two presentations of randomized quicksort are fully equivalent (modulo the technical details about how we might store the priority values).*

Before we get to the analysis, let's observe some properties of quicksort. For these observations, it might be helpful to consider the example shown above. In quicksort, a comparison always involves a pivot and another key. Since, the pivot is never sent to a recursive call, a key is selected as a pivot exactly once, and is not involved in further comparisons (after is becomes a pivot). Before a key is selected as a pivot, it may be compared to other pivots, once per pivot, and thus two keys are never compared more than once.

Question 9.25. *Can you tell which keys are compared by looking at just the call tree?*

Following the discussion above, we note that a key is compared with all its ancestors in the call tree, and all its descendants in the call tree.

Question 9.26. *Let x and z two keys such that $x < z$. Suppose that a key y is selected as a pivot before either x or z is selected. Are x and z compared?*

When the algorithm selects a key (y) in between two keys (x, z) as a pivot, it sends the two keys to two separate subtrees. The two keys (x and z) separated in this way are never compared again.

Question 9.27. *Suppose that the keys x and y are adjacent in the sorted order, how many times are they compared?*

Adjacent keys are compared exactly once. This is the case because there is no key that will ever separate them.

Question 9.28. *Would it be possible to modify quicksort so that it never compares adjacent keys?*

Indeed adjacent keys must be compared by any sorting algorithm since otherwise it would be impossible to tell which one appears first in the output.

Question 9.29. *Based on these observations, can you bound the number of comparisons performed by quicksort in terms of the comparisons between keys.*

The total number of comparisons can be written by summing over all comparisons involving each key. Since each key is involved in a comparison with another key at most once, the total number of comparisons can be expressed as the sum over all pairs of keys. That is, we can just consider each pair of key once and check whether they are ancestors/descendants or not and add one to our sum if so.

Expected work for randomized quicksort. We are now ready to analyze the expected work of randomized quicksort by counting how many comparisons `quicksort` it makes in expectation. We introduce a random variable

$$X_n = \# \text{ of comparisons } \texttt{quicksort} \text{ makes on input of size } n,$$

and we are interested in finding an upper bound on $\mathbf{E}[X_n]$. In particular we will show it is in $O(n \log n)$. $\mathbf{E}[X_n]$ will not depend on the order of the input sequence.

For our analysis we will consider the final sorted order of the keys $T = \texttt{sort}(S)$. In this terminology, we'll also denote by p_i the priority we chose for the element T_i. We'll derive an expression for X_n by breaking it up into a bunch of random variables and bound them. Consider two positions $i, j \in \{1, \ldots, n\}$ in the sequence T (the output ordering). We define following random variable:

$$A_{ij} = \begin{cases} 1 & \text{if } T_i \text{ and } T_j \text{ are compared by quicksort} \\ 0 & \text{otherwise} \end{cases}$$

Based on the discussion in the previous section, we can write X_n by summing over all A_{ij}'s:

$$X_n \leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} A_{ij}$$

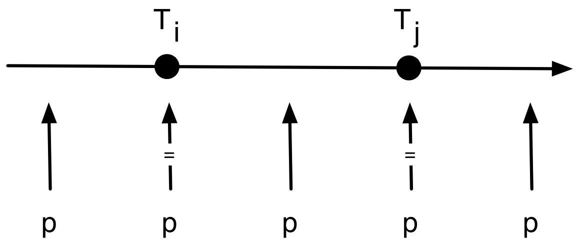


Figure 9.2: The possible relationships between the selected pivot p, T_i and T_j illustrated.

Note that we only need to consider the case that $i < j$ since we only want to count each comparison once. By linearity of expectation, we have

$$\mathbf{E}[X_n] \leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}[A_{ij}]$$

Furthermore, since each A_{ij} is an indicator random variable, $\mathbf{E}[A_{ij}] = \mathbf{Pr}[A_{ij} = 1]$. Our task therefore comes down to computing the probability that T_i and T_j are compared (i.e., $\mathbf{Pr}[A_{ij} = 1]$) and working out the sum.

To compute this probability, let's take a closer look at the quicksort algorithm to gather some intuitions. Notice that the top level takes as its pivot p the element with highest priority. Then, it splits the sequence into two parts, one with keys larger than p and the other with keys smaller than p. For each of these parts, we run `quicksort` recursively; therefore, inside it, the algorithm will pick the highest priority element as the pivot, which is then used to split the sequence further.

For any one call to `quicksort` there are three possibilities (illustrated in Figure 9.2) for A_{ij}, where $i < j$:

- The pivot (highest priority element) is either T_i or T_j, in which case T_i and T_j are compared and $A_{ij} = 1$.
- The pivot is element between T_i and T_j, in which case T_i is in S_1 and T_j is in S_3 and T_i and T_j will never be compared and $A_{ij} = 0$.
- The pivot is less than T_i or greater than T_j. Then T_i and T_j are either both in S_1 or both in S_3, respectively. Whether T_i and T_j are compared will be determined in some later recursive call to `quicksort`.

Let us first consider the first two cases when the pivot is one of $T_i, T_{i+1}, ..., T_j$. With this view, the following observation is not hard to see:

Claim 9.30. *For $i < j$, T_i and T_j are compared if and only if p_i or p_j has the highest priority among $\{p_i, p_{i+1}, \ldots, p_j\}$.*

Proof. Assume first that T_i (T_j) has the highest priority. In this case, all the elements in the subsequence $T_i \ldots T_j$ will move together in the call tree until T_i (T_j) is selected as pivot. When it is, T_i and T_j will be compared. This proves the first half of the theorem.

For the second half, assume that T_i and T_j are compared. For the purposes of contradiction, assume that there is a key T_k, $i < k < j$ with a higher priority between them. In any collection of keys that include T_i and T_j, T_k will become a pivot before either of them. Since $T_i \leq T_k \leq T_j$ it will separate T_i and T_j into different buckets, so they are never compared. This is a contradiction; thus we conclude there is no such T_k. □

Therefore, for T_i and T_j to be compared, p_i or p_j has to be bigger than all the priorities in between. Since there are $j - i + 1$ possible keys in between (including both i and j) and each has equal probability of being the highest, the probability that either i or j is the greatest is $2/(j - i + 1)$. Therefore,

$$\begin{aligned}\mathbf{E}\left[A_{ij}\right] &= \mathbf{Pr}\left[A_{ij} = 1\right] \\ &= \mathbf{Pr}\left[p_i \text{ or } p_j \text{ is the maximum among } \{p_i, \ldots, p_j\}\right] \\ &= \frac{2}{j - i + 1}.\end{aligned}$$

Question 9.31. *What does this bound tell us about the likelihood of keys being compared?*

The bound indicates that the closer two keys are in the sorted order (T) the more likely it is that they are compared. For example, the keys T_i is compared to T_{i+1} with probability 1. It is easy to understand why if we consider the corresponding BST. One of T_i and T_{i+1} must be an ancestor of the other in the BST: There is no element that could be the root of a subtree that has T_i in its left subtree and T_{i+1} in its right subtree.

If we consider T_i and T_{i+2} there could be such an element, namely T_{i+1}, which could have T_i in its left subtree and T_{i+2} in its right subtree. That is, with probability $1/3$, T_{i+1} has the highest probability of the three and T_i is not compared to T_{i+2}, and with probability $2/3$ one of T_i and T_{i+2} has the highest probability and, the two are compared. In general, the probability of two elements being compared is inversely proportional to the number of elements between them when sorted. The further apart the less likely they will be compared. Analogously, the further apart the less likely one will be the ancestor of the other in the related BST.

Hence, the expected number of comparisons made in randomized `quicksort` is

$$\begin{aligned}
\mathbf{E}[X_n] &\le \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}[A_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} n \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&\le 2 \sum_{i=1}^{n-1} H_n \\
&= 2nH_n \in O(n \log n)
\end{aligned}$$

(Recall: $H_n = \ln n + O(1)$.)

Indirectly, we have also shown that the average work for the basic deterministic quicksort (always pick the first element) is also $O(n \log n)$. Just shuffle the data randomly and then apply the basic quicksort algorithm. Since shuffling the input randomly results in the same input as picking random priorities and then reordering the data so that the priorities are in decreasing order, the basic quicksort on that shuffled input does the same operations as randomized quicksort on the input in the original order. Thus, if we averaged over all permutations of the input the work for the basic quicksort is $O(n \log n)$ on average.

An alternative analysis of work. Another way to analyze the work of quicksort is to write a recurrence for the expected work (number of comparisons) directly. This is the approach taken by Tony Hoare in his original paper. For simplicity we assume there are no equal keys (equal keys just reduce the cost). The recurrence for the number of comparisons $X(n)$ done by quicksort is then:

$$X(n) \quad = \quad X(Y_n) + X(n - Y_n - 1) + n - 1$$

where the random variable Y_n is the size of the set S_1 (we use $X(n)$ instead of X_n to avoid double subscrips). We can now write an equation for the expectation of $X(n)$.

$$\begin{aligned}
\mathbf{E}[X(n)] &= \mathbf{E}[X(Y_n) + X(n - Y_n - 1) + n - 1] \\
&= \mathbf{E}[X(Y_n)] + \mathbf{E}[X(n - Y_n - 1)] + n - 1 \\
&= \frac{1}{n} \sum_{i=0}^{n-1} (\mathbf{E}[X(i)] + \mathbf{E}[X(n-i-1)]) + n - 1
\end{aligned}$$

where the last equality arises since all positions of the pivot are equally likely, so we can just take the average over them. This can be by guessing the answer and using substitution. It gives the same result as our previous method. We leave this as exercise.

Span of Quicksort. We now analyze the span of quicksort. All we really need to calculate is the depth of the pivot tree, since each level of the tree has span $O(\log n)$—needed for the filter. We argue that the depth of the pivot tree is $O(\log n)$ by relating it to the number of contraction steps of the randomized `kthSmallest` we considered in Section 9.3. The refer to the i^{th} node of the pivot tree as the node corresponding to the i^{th} smallest key. This is also the i^{th} node in an in-order traversal.

Claim 9.32. *The path from the root to the i^{th} node of the pivot tree is the same as the steps of `kthSmallest` on $k = i$. That is to the say that the distribution of pivots selected and the sizes of each problem is identical. [NEEDS A BIT MORE EXPLANATION.]*

The reason this is true, is that `kthSmallest` is the same as quicksort except we only go down one of the two recursive branches—the branch that contains the k^{th} key. We are almost done now since we showed that the number of steps of `kthSmallest` will be more than $10 \lg n$ with probability at most $1/n^{3.15}$. We might be tempted to say immediately that therefore the probability that quicksort has any node of depth $10 \lg n$ is also $1/n^{3.15}$. This is wrong.

Question 9.33. *Why is it wrong?*

It is wrong because all we have analyzed is that each node had depth greater than $10 \lg n$ with probability at most $1/n^{3.15}$. Since we have multiple nodes the probably increases that at least one will go above the bound. Here is where we get to apply the union bound. Recall the union bound says the probability of the union of a bunch of events is at most the sum of the probabilities of the events. In our case the individual events are the depths of each node being larger $10 \lg n$, and the union is the probability that any of the nodes has depth larger than $10 \lg n$. There are n events each with probability $1/n^{3.15}$, so the union bound states that:

$$\mathbf{Pr}\left[\text{depth of quicksort pivot tree} > 10 \lg n\right] \leq \frac{n}{n^{3.15}} = \frac{1}{n^{2.15}}$$

We thus have our high probability bounds on the depth.

The overall span of randomized quicksort is therefore $O(\log^2 n)$ with high probability.