Chapter 8

Randomized Algorithms

The theme of this chapter is *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 naïve algorithm for finding the top two largest numbers in a sequence that requires about $2n - 3$ comparisons and there is a divide-and-conquer solution that needs about $3n/2$ comparisons (can be implemented with `reduce`). With the aid of randomization, there is a simple randomized algorithm that uses only $n - 1 + 2\ln n$ comparisons on average, under some notion of averaging. In probability speak, this is $n - 1 + 2\ln n$ comparisons in expectation.

In this chapter we will then look at two other algorithms: one for finding the k^{th} smallest element in a sequences, and the other is quickSort. In future chapters we will cover many other algorithms. It turns out that randomized algorithms seem to play an even more important role in parallel algorithms than in sequential algorithms.

8.1 Discrete Probability: Let's Toss Some Dice

Not surprisingly analyzing randomized algorithms requires some understanding of probability. There are two types of bounds on costs (work and span) that we will find useful: expected bounds, and high probability bounds. Expected bounds basically tell us about the average case across all random numbers (or other source of randomness) used in the algorithm. For example if an algorithm does $O(n)$ expected work, it means that on average across all random number selections, the algorithm does $O(n)$ work. Once it a while, the work could be much larger, for example once in every n tries the algorithm might require $O(n^2)$ work.

Exercise 8.1. *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 $\Theta(n^3)$?*

High probability bounds on the other hand tell us that it is quite unlikely that the cost would be above the bounds. Typically the probability is stated in terms of n. For example me might determine that an algorithm does $O(n)$ work with probability at least $1 - 1/n^5$. This means that only once in about n^5 tries will it take more than $O(n)$ work. High probability bounds are stronger, especially in conjunction with expectation bounds, but they can be harder to prove.

The reason that it is often easy to work with expected bounds is that if there are a bunch of components of an algorithm or task which we are putting together we can add the expected cost of each together to get an overall expectation of the sum of costs. For example, if we have 100 students and they each take on average 2 hours to finish a task, then the total time spent on average across all students will be 200 hours. This makes analysis reasonably easy since it is compositional. Indeed many textbooks that cover randomized algorithms only cover expected bounds since it makes analysis easy. Unfortunately such composition does not work when taking a maximum. 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 is 2 hours. It could be that typically one student gets hung up and takes 100 hours and the rest 1 hour. We can assume it is a different student each time, so the average for each student is still at most 2 hours. In this case, however the expected maximum time is 100 hours, instead of 2 hours. Since in parallel algorithms we often care about maximum when calculating the span, using expectations is sometimes not as useful as it is for sequential algorithms. In this book we often alalyze 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. 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).

Probability is really about counting, or possibly weighted counting. In the dice example we had 36 possibilities and and a subset (4 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 averages over the outcomes of some property (e.g. the average product of the two dice is $12\frac{1}{4}$). If not all outcomes are equally likely, then the sum has to be weighted by the probability of each outcome. For this purpose we define the notion of a sample space (the set of all possible outcomes), primitive events (each possible outcome), events (subsets of outcomes that satisfy a property), probability functions (defining the probability for each primitive event), and random variables (functions from the sample space to the real numbers indicating some property of the outcomes).

A *sample space* Ω is an arbitrary and possibly infinite (but countable) set of possible outcomes of a probabilistic experiment. In our dice tossing example the Ω is the set of 36 possible outcomes of tossing the two dice: $\{(1,1),(1,2),\ldots,(2,1),\ldots,(6,6)\}$. Often in analyzing algorithms we are not tossing dice, but instead using random numbers. In this case the sample space is all possible outcomes of the random numbers used. In the next section we analyze finding the top two lagerst 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. For example it could be the dice toss $(1,4)$. An *event* is any subset of Ω and typically represents some property common to multiple primitive events. For example an event could be "the first dice is 3" which would correspond to the set $\{(3,1),(3,2),(3,3),(3,4),(3,5),(3,6)\}$, or it could be "the dice sum to 4", which would correspond to the set $\{(1,3),(2,2),(3,1)\}$.

A *probability function* $\mathbf{Pr} : \Omega \to [0,1]$ is associated with the sample space with the condition that $\sum_{e \in \Omega} \mathbf{Pr}\left[e\right] = 1$ (i.e. the probabilities of the outcomes of a probabilistic experiment sum to 1, as we would expect). The probability of an event A is simply the sum of the probabilities of its primitive events: $\mathbf{Pr}\left[A\right] = \sum_{e \in A} \mathbf{Pr}\left[e\right]$. For example in tossing two dice, the probability of the event "the first dice is 3" is $\frac{6}{36} = \frac{1}{6}$, and the probability of the event "the dice sum to 4" is $\frac{3}{36} = \frac{1}{12}$ (as above). With fair dice all probabilities are equal so all we have to do is figure out the size of each 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.

A *random variable* X on a sample space Ω 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. For example for the two dice example, we could have a random variable X representing the sum of the two rolls (**function** $X(d_1, d_2) = d_1 + d_2$) or a random variable Y that is 1 if the values on the dice are the same and 0 otherwise (**function** $Y(d_1, d_2) =$ **if** $(d_1 = d_2)$ **then** 1 **else** 0). This second random variable Y is called an *indicator random variable* since it takes on the value 1 when some condition is true and 0 otherwise. In particular, for a predicate $p : \Omega \rightarrow \texttt{bool}$ a random indicator variable is defined as **function** $Y(e) =$ **if** $p(e)$ **then** 1 **else** 0. For a random variable X and a number $a \in \mathbb{R}$, the event "$X = a$" is the set $\{\omega \in \Omega \mid X(\omega) = a\}$. We typically denote random variables by capital letters from the end of the alphabet, e.g. X, Y Z.

We note that the term random variable might seem counter intuitive based on our definition 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. The term seems to be historical from before probability theory was formalized using sample spaces. Without the notion of a sample space, the sum of two roles of a dice, for example, is a variable that takes on random values between 2 and 12. We could plot a probability density function with the variable on the x-axis, and the probability on the y-axis. It would represent the probability that the "variable" takes on different values.

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:

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

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

The expectation of an indicator random variable Y is the probability that the associated predicate p is true. This is because

$$\mathbf{E}[Y] = \sum_{e \in \Omega} (\textbf{if } p(e) \textbf{ then } 1 \textbf{ else } 0) \cdot \mathbf{Pr}[e] = \sum_{e \in \Omega, p(e) = \mathtt{true}} \mathbf{Pr}[e] = \mathbf{Pr}[\{e \in \Omega \mid p(e)\}] \ .$$

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 $\mathbf{Pr}[A \cap B] = \mathbf{Pr}[A] \cdot \mathbf{Pr}[B]$. For our dice throwing example, 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 $\mathbf{Pr}[A] = \mathbf{Pr}[B] = \frac{1}{6}$ and $\mathbf{Pr}[A \cap B] = \frac{1}{36}$.

However the event $C = \{(d_1, d_2) \in \Omega \mid d_1 + d_2 = 4\}$ (the dice add to 4) is not independent of A since $\mathbf{Pr}[C] = \frac{1}{12}$ and $\mathbf{Pr}[A \cap C] = \mathbf{Pr}[\{(1,3)\}] = \frac{1}{36} \neq \mathbf{Pr}[A] \cdot \mathbf{Pr}[C] = \frac{1}{6} \cdot \frac{1}{12} = \frac{1}{72}$. They are not independent since the fact that the first dice is 1 increases the probability they sum to 4 (from $\frac{1}{12}$ to $\frac{1}{6}$), or the other way around, the fact that they sum to 4 increases the probability the first dice is 1 (from $\frac{1}{6}$ to $\frac{1}{3}$).

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 $I \subseteq \{1, \ldots, k\}$,

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

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 $\{X \leq a\}$ and $\{Y \leq b\}$ 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.

Markov's Inequality. Consider a non-negative random variable X. We can ask how much bigger 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 add too much to the sum.

Question 8.2. *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 would contribute more than $\beta\,\mathbf{E}[X] \times \frac{1}{\beta} = \mathbf{E}[X]$ to the expectation. This gives us the inequality:

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

or equivalently

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

which is known as Markov's inequality.

Linearity of Expectations

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

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

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

To continue our running example, let's consider analyzing the expected sum of values when throwing two dice. One way to calculate this is to consider all 36 possibilities and take their average. What is this average? A much simpler way is to sum the expectation for each of the two dice. The expectation for either dice is the average of just the six possible values $1, 2, 3, 4, 5, 6$, which is 3.5. Therefore the sum of the expectations is 7.

Note that for a binary function f the equality $f(\mathbf{E}\left[X\right], \mathbf{E}\left[Y\right]) = \mathbf{E}\left[f(X, Y)\right]$ is **not** true in general. For example $\max(\mathbf{E}\left[X\right], \mathbf{E}\left[Y\right]) \neq \mathbf{E}\left[\max(X, Y)\right]$. If it the equality held, the expected maximum value of two rolled dice would be 3.5.

Exercise 8.3. *What is the expected maximum value of throwing two dice?*

We note that $\mathbf{E}\left[X\right] \times \mathbf{E}\left[Y\right] = \mathbf{E}\left[X \times Y\right]$ is true if X and Y are independent. The expected value of the product of the values on two dice is therefore $3.5 \times 3.5 = 12.25$.

8.2 Finding The Two Largest

The max-two problem is to find the two largest elements from a sequence of n (unique) numbers. For inspiration, we'll go back and look at the naïve algorithm for the problem:

```
function max2($S$) = let
  function replace(($m_1, m_2$), $v$) =
    if $v \le m_2$ then ($m_1, m_2$)
    else if $v \le m_1$ then ($m_1, v$)
    else ($v, m_1$)
  val start = if $S_1 \ge S_2$ then ($S_1, S_2$) else ($S_2, S_1$)
in iter replace start $S\langle 3, \ldots, n\rangle$
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

Example 8.4. *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'll 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.

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

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,

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

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

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

Example 8.5. *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.

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 3; therefore, only some of the elements will need the comparison in Line 4. 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 3 and 4.

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 4 by considering elements in random order.*

The algorithm we'll analyze is the following. On input a sequence S of n elements:

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

Remarks: We don't need to explicitly construct T. 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 T.

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).

8.2.1 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 $\mathbf{Pr}\left[e\right] = 1/n!, e \in \Omega$. 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.

Now let X_i be an indicator variable denoting whether Line 4 gets executed for this particular value of i (i.e., Did T_i get compared in Line 4?) That is, $X_i = 1$ if T_i is compared in Line 4 and 0 otherwise. 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 4 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:

$$\textbf{function } Y(e) = \underbrace{1}_{\text{Line } 6} + \underbrace{n-2}_{\text{Line } 3} + \underbrace{\sum_{i=3}^{n} X_i(e)}_{\text{Line } 4}$$

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 $\mathbf{E}\left[Y\right] = \sum_{e \in \Omega} \mathbf{Pr}\left[e\right] Y(e)$. By linearity of expectation, we have

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

Our tasks therefore boils down to computing $\mathbf{E}\left[X_i\right]$ for $i = 3, \ldots, n$. To compute this expectation, we ask ourselves: *What is the probability that* $T_i > m_2$? A moment's thought shows that the condition $T_i > m_2$ holds exactly when T_i is either the largest element or the second largest element in $\{T_1, \ldots, T_i\}$. So ultimately we're asking: what is the probability that T_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 T_i. (You should also try to work it out using a counting argument.) Therefore, the probability that T_i is the largest or the second largest element in $\{T_1, \ldots, T_i\}$ 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

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

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).

8.3 Finding The k^{th} Smallest Element

Consider the following problem:

Problem 8.6 (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.

```
function kthSmallest$(k, S)$ = let
        val $p$ = a value from $S$ picked uniformly at random
        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$(k - (|S| - |R|), R)$
```

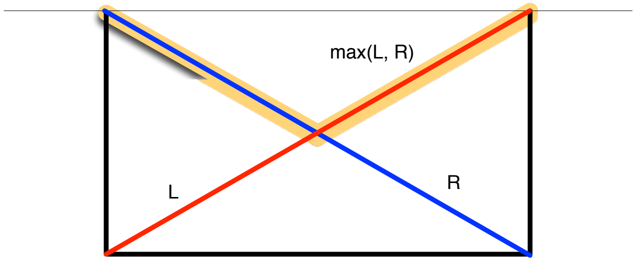
This algorithm is similar to quickSort but instead of recursing on both sides, it only recurses on one side. Basically it figures out in which side the k^{th} smallest must be in, and just explores that side. When exploring the right side, R, the k needs to be adjusted by since all elements less or equal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

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

$$\begin{array}{lclcl} W(n) & \leq & W(X_n) & + & O(n) \\ S(n) & \leq & S(X_n) & + & O(\log n) \end{array}$$

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.

How big is $\mathbf{E}\left[X_n\right]$*?* To understand this, let's take a look at a pictorial representation:



The probability that we land on a point on the curve is $1/n$, so

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

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

This computation tells us that in expectation, X_n is a constant fraction smaller than n ($\leq \frac{3n}{4}$), so intuitively we should have a nice geometrically decreasing sum, which works out 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. There will be other algorithms we cover when we talk about graphs that have the same property, i.e. that the expected size goes down by a consant factor. 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 8.7. *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 i, and let F_i be the random variable representing the fraction of elements that are kept on the i^{th} step, giving $Y_i = n\prod_{j=1}^{i} F_j$. We have that $F_i = \frac{X_n}{n}$, giving $\mathbf{E}\left[F_i\right] = \mathbf{E}\left[\frac{X_n}{n}\right] \leq \frac{3}{4}$. Now since we pick the pivot independently on each step, the F_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:

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

□

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

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

Note that in the sum over steps 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 8.7 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 probabilty bound as discussed at the start of Section 8.1. 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 overal 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 8.1 we will typically be analyzing work using expectation and span using high probability. This is partly because we cannot compose span using expectations. We note that the high probability bounds given here are strictly stronger than expected bounds since they imply expected bounds.

Exercise 8.8. *Show that the high probability bounds on span for `kthSmallest` imply equivalent bounds in expectation.*

8.4 Quicksort

You have surely seen quicksort before. The purpose of this section is to analyze quicksort in terms of both its work and its span. 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 following implementation of quicksort. In this implementation, we intentionally leave the pivot-choosing step unspecified because the property we are discussing holds regardless of the choice of the pivot.

Pseudo Code 8.9 (Quicksort 1/2)**.**

```
function $\mathtt{quicksort}(S) =$
  if $|S| = 0$ then $S$
  else let
      val $p = \textit{pick a pivot from } S$
      val $S_1 = \langle\, s \in S \mid s < p \,\rangle$
      val $S_2 = \langle\, s \in S \mid s = p \,\rangle$
      val $S_3 = \langle\, s \in S \mid s > p \,\rangle$
      val $(R_1, R_3) = (\, \mathtt{quicksort}(S_1) \;\|\; \mathtt{quicksort}(S_3) \,)$
    in
      $\mathtt{append}(R_1, \mathtt{append}(S_2, R_3))$
    end
```

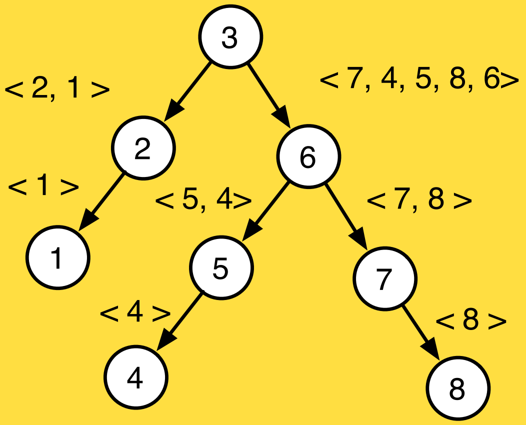
Question 8.10. *Is there parallelism in quicksort?*

There is clearly 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.

Example 8.11. *An example run of quicksort.*

Keys < 7, 4, 2, 3, 5, 8, 1, 6>

Example Run

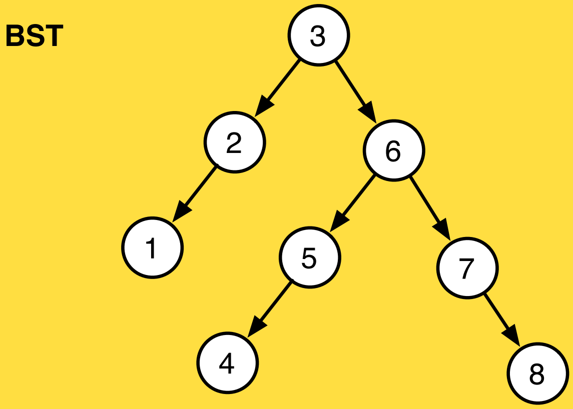


Note that each call to quicksort either makes no recursive calls (the base case) or two recursive

[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.

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.

Example 8.12. *An example call tree, which is a binary-search-tree, illustrated.*



Question 8.13. *How does the pivot choice effect the costs of quicksort?*

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 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. 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.

8.5 Analysis of Quicksort, the Basics

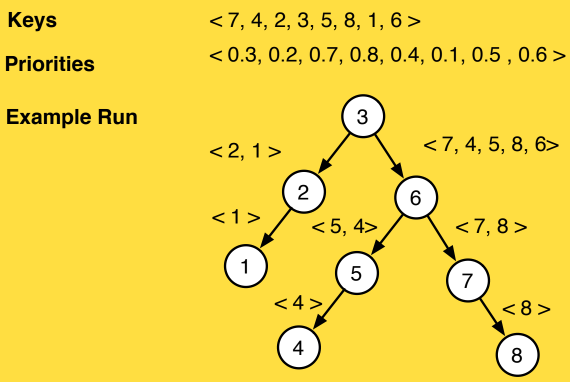
In this lecture, we will analyze the algorithm that selects a uniformly randomly chosen key as the pivot. For the analysis, we are going to rewrite the algorithm slightly so that we compare each key with the pivot only once.

Pseudo Code 8.14 (Quicksort 2/2)**.**

```
1   function $\texttt{quicksort}(S) =$
2     if $|S| = 0$ then $S$
3     else let
4           val $p = \textit{pick a pivot from } S$
5           val $C = \langle\, (s, compare(p, s)) : s \in S \,\rangle$
6           val $S_1 = \langle\, s \mid (s, LESS) \in C \,\rangle$
7           val $S_2 = \langle\, s \mid (s, EQUAL) \in C \,\rangle$
8           val $S_3 = \langle\, s \mid (s, GREATER) \in C \,\rangle$
9           val $(R_1, R_3) = (\, \texttt{quicksort}(S_1) \;\|\; \texttt{quicksort}(S_3)\,)$
10        in
11          $append(R_1, append(S_2, R_3))$
12        end
```

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 4 the key with the highest priority. Notice that once the priorities are decided, the algorithm is completely deterministic.

Example 8.15. *Quicksort with priorities and its call tree, which is a binary-search-tree, illustrated.*



Exercise 8.16. *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.

Question 8.17. *How many time any two keys x and y in the input are compared in a run of the algorithm?*

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 a pivot, it may be compared to other pivots, once per pivot, and thus two keys are never compared more than once.

Question 8.18. *Can you tell which keys are compared by looking at just the call tree?*

Following on the discussion above, a key is compared with all its ancestors in the call tree. Or alternatively, a key is compared with all the keys in its subtree.

Question 8.19. *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 8.20. *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 separate them.

Question 8.21. *Would it be possible to modify quicksort so that it never compares adjacent keys?*

Adjacent keys must be compared, because otherwise it would be impossible for quicksort to distinguish between two orders where adjacent keys are swapped.

Question 8.22. *Based on these, can you bound the number of comparisons performed by quicksort in terms of the comparisons between keys.*

Obviously 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 of if so. We never have to consider each pair more than once.

8.6 Expected work for randomized quicksort

As discussed above, if we always pick the first element then the worst-case work is $O(n^2)$, for example when the array is already sorted. The *expected work*, though, is $O(n \log n)$ as we will prove below. That is, the work averaged over all possible input ordering is $O(n \log n)$. In other words, on most input this naive version of quicksort works well on average, but can be slow on some (common) inputs.

On the other hand, if we choose an element randomly to be the pivot, the *expected worst-case work* is $O(n \log n)$. That is, for input in **any** order, the expected work is $O(n \log n)$: No input has expected $O(n^2)$ work. But with a very small probability we can be unlucky, and the random pivots result in unbalanced partitions and the work is $O(n^2)$.

We're interested in counting how many comparisons `quicksort` makes. This immediately bounds the work for the algorithm because this is where the bulk of work is done. That is, if we let

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

our goal is to find an upper bound on $\mathbf{E}\left[X_n\right]$ for any input sequence S. For this, we'll consider

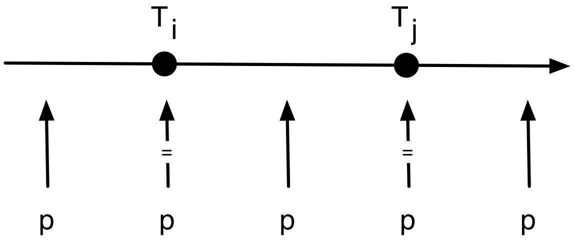


Figure 8.1: The possible relationships between the selected pivot p, T_i and T_j illustrated.

the final sorted order[2] 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. We use the random indicator variables A_{ij} to indicate whether we compare the elements T_i and T_j during the algorithm—i.e., the variable will take on the value 1 if they are compared and 0 otherwise.

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

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

By linearity of expectation, we have

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

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

[2] Formally, there's a permutation $\pi \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ between the positions of S and T.

- 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 8.23. *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 8.24. *What does this bound tell us about the likelihood of keys being compared?*

The bound says that the closer two keys in the sorted order, 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

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

(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.

8.6.1 An alternative method

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 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 $X(n)$.

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

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.

8.7 Expected Span of Quicksort

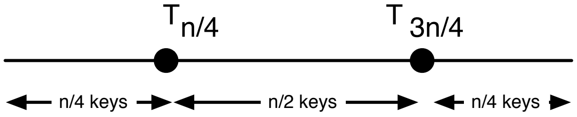
Recall that in randomized quicksort, at each recursive call, we partition the input sequence S of length n into three subsequences L, E, and R, such that elements in the subsequences are less than, equal, and greater than the pivot, respectfully. Let $M_n = \max\{|L|, |R|\}$, which is the size of larger subsequence. The span of quicksort is determined by the sizes of these larger subsequences. For ease of analysis, we will assume that $|E| = 0$, as more equal elements will only decrease the span. As this partitioning uses `filter` we have the following recurrence for span:

$$S(n) = S(M_n) + O(\log n)$$

Probability of splitting the input proportionately. To develop some intuition for the span analysis, let's consider the probability that we split the input sequence more or less evenly.

Question 8.25. *What is the probability that M_n is at most $3n/4$?*

If we select a pivot that is greater than $T_{n/4}$ and less than $T_{3n/4}$ then M_n is at most $3n/4$. Since all keys are equally likely to be selected as a pivot this probability is $\frac{3n/4 - n/4}{n} = 1/2$. The figure below illustrates this.



Question 8.26. *What does this say about what the span of quicksort may be?*

This observations implies that at each level of the call tree (every time a new pivot is selected), the size of the input to both calls decrease by a constant fraction (of $4/3$). At every two levels, the probability that the input size decreases by $4/3$ is the probability that it decreases at either step, which is at least $3/4$, etc. Thus at a small constant number of steps, the probability that we observe a $4/3$ factor decrease in the size of the input approches 1 quickly. This suggest that at some after $c \log n$ levels quicksort should complete. We now make this intuition more precise.

Conditioning and the total expectation theorem. For the analysis, we are going to use the conditioning technique for computing expectations. Specifically, we will use the total expectation theorem. Let X be a random variable and let A_i be disjoint events that form a a partition of the sample space such that $P(A_i) > 0$. The total expectation theorem states that

$$E[X] = \sum_{i=1}^{n} P(A_i) \cdot E[X|A_i].$$

Bounding Span. As we did for `SmallestK`, we will bound $E[S_n]$ by considering the $\textbf{Pr}\left[X_n \leq 3n/4\right]$ and $\textbf{Pr}\left[X_n > 3n/4\right]$ and use the maximum sizes in the recurrence to upper bound $\textbf{E}\left[S_n\right]$. Now, the $\textbf{Pr}\left[X_n \leq 3n/4\right] = 1/2$, since half of the randomly chosen pivots results in the larger partition to be at most $3n/4$ elements: Any pivot in the range $T_{n/4}$ to $T_{3n/4}$ will do, where T is the sorted input sequence.

By conditioning S_n on the random variable M_n, we write,

$$E[S_n] = \sum_{m=n/2}^{n} \textbf{Pr}\left[M_n = m\right] \cdot E[S_n|(M_n = m).$$

Now, we can write this trivially as

$$E[S_n] = \sum_{m=n/2}^{n} \textbf{Pr}\left[M_n = m\right] \cdot E[S_m].$$

The rest is algebra

$$\begin{aligned} E[S_n] &= \sum_{m=n/2}^{n} \textbf{Pr}\left[M_n = m\right] \cdot E[S_m] \\ &\leq \textbf{Pr}\left[M_n \leq \frac{3n}{4}\right] \cdot E[S_{\frac{3n}{4}}] + \textbf{Pr}\left[M_n > \frac{3n}{4}\right] \cdot E[S_n] + c \cdot \log n \\ &\leq \frac{1}{2} E[S_{\frac{3n}{4}}] + \frac{1}{2} E[S_n] \\ &\implies E[S_n] \leq E[S_{\frac{3n}{4}}] + 2c \log n. \end{aligned}$$

This is a recursion in $E[S(\cdot)]$ and solves easily to $E[S(n)] = O(\log^2 n)$.