Chapter 11

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.

Even though analyzing randomized algorithms can be difficult, randomization turns out to be a crucial technique for algorithm design, making the extra effort well worth. For example, for some problems randomized algorithms are simpler or faster than non-randomized algorithms. 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 particularly 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 kth order statistics of a

sequences, and the other is quicksort. In future chapters we will cover many other randomized algorithms.

In this book we 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 correct answer, depending on random choices, are called ***Monte Carlo*** algorithms.

11.1 Expectation versus High Probability

In analyzing costs for a randomized algorithms there are two types of bounds that are useful: expected bounds, and high-probability bounds.

Expected bounds tell us about the average cost across all random choices made by the algorithm. For example if an algorithm has $\Theta(n)$ expected work, it means that on averaged over all random choices it makes in all runs, the algorithm performs $\Theta(n)$ work. Since expected bounds are averaged over all random choices in all possible runs, there can be runs that require more or less work. For example once in every $1/n$ tries the algorithm might require $\Theta(n^2)$ work, and (or) once in every $\sqrt{n}$ tries the algorithm might require $\Theta(n^{3/2})$ work.

High-probability bounds on the other hand tell us that it is very unlikely that the cost will be above some bound. 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 $1/n^k$. 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. High-probability bounds are typically stronger than expectation bounds.

Expected bounds are quite convenient when analyzing work (or running time in traditional sequential algorithms). This is because the linearity of expectations (Chapter 10) allows adding expectations across the components of an algorithm to get the overall expected work. For example, if the algorithm performs n tasks each of which take on average 2 units of work, then the total work on average across all tasks will be $n \times 2 = 2n$ units. Unfortunately this kind of composition does not work when analyzing the span of an algorithm, because this requires taking the maximum of random variables, rather than their sum. For example, if we had n tasks each of which has expected span of 2 units of time, we cannot say that the expected span across all tasks is 2 units. It could be that most of the time each task has a span of 2 units, but that once with probability $1/n$, the task requires n units. The expected span for each task is still close to 2 units but if we have n tasks chances are high that one task will take n units and the expected maximum will be close to n rather than 2. We therefore cannot compose the expected span from each task by taking a maximum.

Unlike expected bounds, high-probability bounds can allow us to bound span. For example, lets say we know that every task finishes in 2 units of time with probability $1 - 1/n^5$, or equivalently that each task takes more than 2 units of time with probability $1/n^5$ and takes at most n units of time otherwise. Now with n tasks the probability that there will be at least one

Figure 11.1: 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.

that requires more than 2 units of time is at most $1/n^4$ by union bound. Furthermore, when it does, the contribution to the expectation is $1/n^3$. 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.

11.2 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 iterative algorithm for the problem.

Algorithm 11.1. [Iterative Max-Two]

```
1  max2 a =
2  let
3    update ((m1, m2), v) =
4      if v ≤ m2 then (m1, m2)
5      else if v ≤ m1 then (m1, v)
6      else (v, m1)
7    init = if a[0] ≥ a[1] then (a[0], a[1]) else (a[1], a[0])
8  in
9    iter update init a[2, ..., n − 1]
10 end
```

In the following analysis, we will be meticulous about constants. This iterative algorithm requires up to $1 + 2(n - 2) = 2n - 3$ comparisons since there is one comparison in `init` and since each of the $n - 2$ `update`'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?

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 **??**; therefore, only some of the elements will need the comparison in Line **??**. 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. It is not difficult to convince yourself that an increasing sequence of length n, e.g., $\langle 1, 2, 3, \ldots, n \rangle$ leads to $2n - 3$ comparisons. As we iterate from left to right, we find a new maximum for each new element—this new element gets compared in both Lines 4 and 5.

But perhaps it's unlikely to get such a deliberately structured sequence if we consider the elements in random order. With only 1 in $n!$ chance, a 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. Thus we can reasonably hope to save a lot of comparisons in Line **??** by considering elements in random order.

Let's thus analyze the following algorithm: on input a sequence t of n elements:

1. Let $a = \texttt{permute}(t, \pi)$, where π is a random permutation (i.e., we choose one of the $n!$ permutations).

2. Run algorithm `max2` on a.

Note that we don't need to explicitly construct a. All we need instead is to pick a random element that hasn't been considered and consider that element next. For the analysis, it is convenient to describe the process in terms of a randomly permuted sequence.

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 $\textbf{Pr}[x] = 1/n!, x \in \Omega$. 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 a (indexed from 1 to n). Now let X_i be an indicator random variable denoting whether Line **??** 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 comparisons the algorithm takes on

that permutation. This can be defined as

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

We are interested in computing the expected value of Y. 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 $a_i > m_2$?* A moment's thought shows that the condition $a_i > m_2$ holds exactly when a_i is either the largest element or the second largest element in $\{a_1, \ldots, a_i\}$. So ultimately we're asking: what is the probability that a_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 a_i. (You should also try to work it out using a counting argument.) Therefore, the probability that a_i is the largest or the second largest element in $\{a_1, \ldots, a_i\}$ 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 obtain

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

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

$$H(n) = O(\lg n + 1),$$

or more precisely

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

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).

Remark 11.2. Reducing the number of comparisons by approximately a factor of two might not lead to a significant gain in performance in practice. For example, if the comparison function is a constant-time, simple comparison function the $2n-3$ algorithm and the $n-1+2\log n$ algorithm are unlikely to be significant. For most cases, the $2n-3$ algorithm might in fact be faster to due better 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).

11.3 Order statistics

In statistics, computing the order statistics of sample, which we may represent as a sequence, has many important applications. We can precisely state the problem as follows.

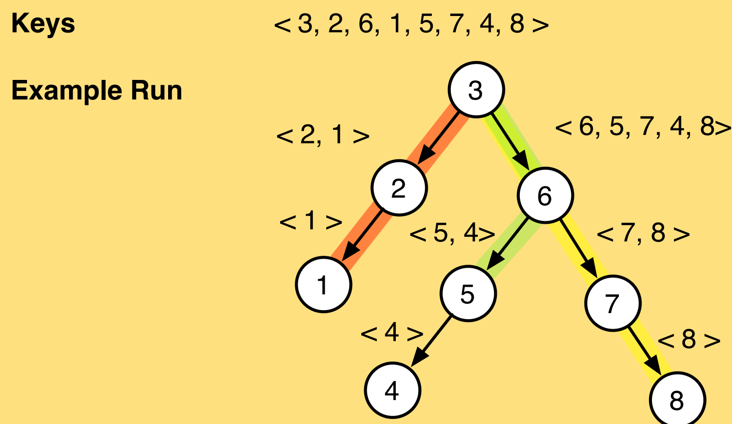
Problem 11.3. [Order statistics] Given an a sequence and an integer k where $0 \leq k < |a|$, and a comparison $<$ defining a total ordering over the elements of the sequence, find the kth order statistics, i.e., k^{th} smallest element, in the sequences.

We can solve this problem by sorting first and selecting the kth element but this would require $O(n \log n)$ work, assuming that comparisons require constant work. We wish to do better; in particular we would like to achieve linear work and still achieve $O(\log^2 n)$ span. For the purposes of simplicity, let's assume that sequences consist of unique elements and consider the following simple algorithm. Based on the contraction design technique, the algorithm uses randomization to contract the problem to a smaller instance.

Algorithm 11.4. [contracting k^{th} smallest]

select a k =
let
  $p = a[0]$
  $\ell = \langle x \in a \mid x < p \rangle$
  $r = \langle x \in b \mid x > p \rangle$
in
  if $(k < |\ell|)$ **then** select ℓ k
  else if $(k < |a| - |r|)$ **then** p
  else select r $(k - (|a| - |r|))$

Example 11.5. Example runs of `select` illustrated by a "pivot tree." For illustrative purposes, we show all possible recursive calls being explored down to singleton sequences. In reality, the algorithm explores only one path. The path highlighted with red is the path of recursive calls taken by `select` when searching for the first-order statistics, $k = 0$. The path highlighted with brown is the path of recursive calls taken by `select` when searching for the fifth-order statistics, $k = 4$. The path highlighted with green is the path of recursive calls taken by `select` when searching for the eight-order statistics, $k = 7$.



The algorithm divides the input into left and right sequences, ℓ and r, and figures out the side k^{th} smallest must be in, and recursively explores that side. When exploring the right side, r, the parameter k needs to be adjusted by since all elements less or equal to the pivot p are being thrown out: there are $|a| - |r|$ such elements.

As written the algorithm picks as pivot the first key in the sequence instead of a random key. As with the two-largest problem, we can add randomness by first randomly permuting a sequence t to generate the input sequence a and then applying `select` on a. This is equivalent to randomly picking a pivot at each step of contraction.

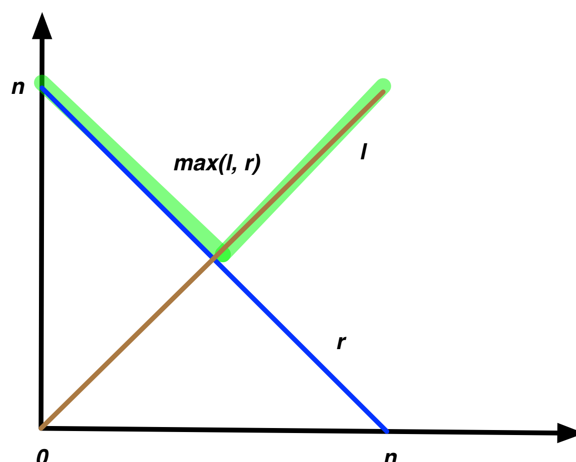
Let's analyze the work and span of the randomized algorithm where we pick pivots uniformly randomly. Let $n = |a|$ and define $X(n) = \max\{|\ell|, |r|\}/|a|$, which is the fractional size of the larger side. Notice that X is an upper bound on the fractional size of the side the algorithm actually recurs into. Now since lines 3 and 4 are simply two `filter` calls, we have the following recurrences:

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

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.

The key quantity in bounding th expectation is bounding $\mathbf{E}\left[X(n)\right]$. To this end, let's none first that all pivots are equally likely. We can thus 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 a.



If the pivot is at the start then ℓ is empty and $|r| = |a| - 1$, and if the pivot is at the end then r is empty and $|\ell| = |a| - 1$. Since the probability that we land on a point on the x axis is $1/n$, we can write the expectation for $X(n)$ as

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

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

This calculation tells us that in expectation, $X(n)$ 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 also be we are unlucky for a few contraction steps and the sequences size hardly goes down at all. 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. Note that the proof of this theorem would have been relatively easy if the successive choices made by the algorithm were independent but they are not, because the size to the algorithm at each recursive call depends on prior choices of pivots.

Theorem 11.6. *Starting with size n, the expected size of a in algorithm* `select` *after d recursive calls is $\left(\frac{3}{4}\right)^d n$.*

Proof. The proof is by induction on the depth of the recursion d. In the base case, $d = 0$ and the lemma holds trivially. For the inductive case assume that the lemma holds for some $d \geq 0$. Consider now the $(d+1)^{th}$ recursive call. Let Y_d be the random variable denoting the size of the input to the dth recursive call and let Z the pivot chosen at the dth call. For any value of y and z, let $f(y, z)$ be the fraction of the input reduced by the choice of the pivot at position z for

an input of size y. We can write the expectation for the input size at $(d+1)^{st}$ call as

$$\begin{array}{rcl} E[Y_{d+1}] & = & \sum_{y,z} y f(y,z) \mathbf{P}_{Y,Z}(y,z) \\ & = & \sum_y \sum_z y f(y,z) \mathbf{P}_Y(y) \mathbf{P}_{Z \mid Y}(z \mid y) \\ & = & \sum_y y \mathbf{P}_Y(y) \sum_z f(y,z) \mathbf{P}_{Z \mid Y}(z \mid y) \\ & \leq & \sum_y y \mathbf{P}_Y(y) \, \mathbf{E}\left[X(y)\right]. \\ & \leq & \frac{3}{4} \sum_y y \mathbf{P}_Y(y). \\ & \leq & \frac{3}{4} \, \mathbf{E}\left[Y_d\right]. \end{array}$$

Note that we have used the bound

$$\mathbf{E}\left[X(y)\right] = \sum_z f(y,z) \mathbf{P}_{Z \mid Y}(z \mid y) \leq \frac{3}{4},$$

which we established above.

We thus conclude that $\mathbf{E}\left[Y_{d+1}\right] = \frac{3}{4} \mathbf{E}\left[Y_d\right]$, which this trivially solves to the bound given in the theorem, since at $d = 0$ the input size is n. □

The work at each level of the recursive calls is linear in the size of the input and thus can be written as $W_{\texttt{select}}(n) \leq k_1 n + k_2$, where n is the input size. Since at least one element, the pivot, is taken out of the input for the recursive call at each level, there are at most n levels of recursion, and thus, we can bound the expected work as

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

Expected Span. We can bound the span of the algorithm by $O(n \lg n)$ trivially in the worst case, but we expect the average span to be a lot better because chances of picking a poor pivot over and over again, which would be required for the linear span is unlikely. To bound the span in the expected case, we shall use Theorem 11.6 to bound the number of levels taken by `select` more tightly using a high probability bound.

Consider depth $d = 10 \lg n$. At this depth, the expected size upper bounded by $n \left(\frac{3}{4}\right)^{10 \lg n}$. With a little math this is equal to $n \times n^{-10 \lg(4/3)} \approx n^{-3.15}$. Now, by Markov's inequality, if the expected size is at most $n^{-3.15}$ then the probability of having size at least 1 is bounded by

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

In applying Markov's inequality, we choose 1, because we know that the algorithm terminates for that input size. By increasing the constant factor from 10 to 20 would decrease the probability to $n^{-7.15}$, which is extremely unlikely: for $n = 10^6$ this is 10^{-42}. We have therefore shown 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.

Using the high probability bound, we can bound the expected span by using the total expectation theorem. For brevity let the random variable Y be defined as $Y = Y_{10 \lg n}$,

$$\begin{aligned}
\mathbf{E}[S] &= \textstyle\sum_y \mathbf{P}_Y(y)\, \mathbf{E}[S \mid Y = y]\,. \\
&= \textstyle\sum_{y \le 1} \mathbf{P}_Y(y)\, \mathbf{E}[S \mid Y = y] + \sum_{y > 1} \mathbf{P}_Y(y)\, \mathbf{E}[S \mid Y = y] \\
&\le (1 - n^{-3.5}) O(\lg^2 n) + n^{-3.5} O(n) \\
&= O(\lg^2 n).
\end{aligned}$$

The expected bound follows by the fact that with high probability the depth of the recursive calls is $O(\lg n)$ and that each recursive call has $O(\lg n)$ span, because it requires a sequences `filter`. The span for the case when the span is not greater that $10 \lg n$ contributes only a constant value to the expectation as long as it is a polynomial that is less that $n^{3.5}$.

In summary, we have shown than the `select` 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 the chapter, we will typically be analyzing work using expectation and span using high probability.

11.4 Quicksort

Moving on to a more complex algorithm, let's analyze the work and span of the randomized quicksort algorithm. 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.

Consider the quicksort algorithm given in Algorithm 11.7. 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.

There is plenty of parallelism in this version quicksort. 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 11.8.

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

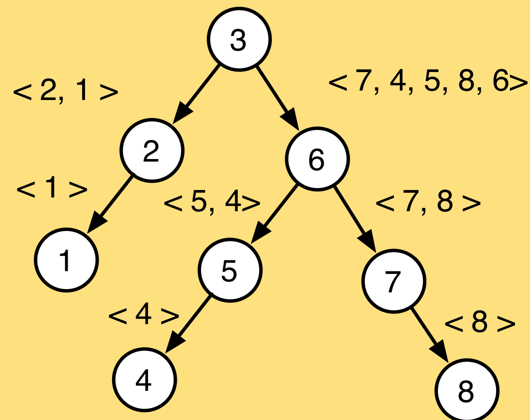
Algorithm 11.7. [Quicksort]

$\texttt{quicksort}\ a\ =$
if $|a| = 0$ **then** a
else
  let
    $p = \textit{pick a pivot from } a$
    $a_1 = \langle\, x \in a \mid x < p \,\rangle$
    $a_2 = \langle\, x \in a \mid x = p \,\rangle$
    $a_3 = \langle\, x \in a \mid x > p \,\rangle$
    $(s_1, s_3)\ =\ (\texttt{sort}\ a_1\ \|\ \texttt{sort}\ a_3)$
  in
    $s_1\ \texttt{++}\ a_2\ \texttt{++}\ s_3$
  end

Example 11.8. An example run of quicksort along with its pivot tree.

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

Example Run



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.

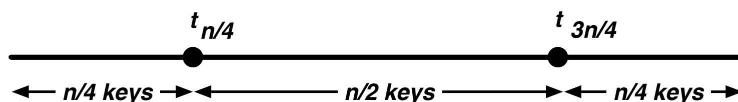
- **Pick the median of three elements:** Another strategy is to take the first, middle, and the last elements and pick the median of them. For sorted lists the split is even, so each side contains half of the original size and the depth of the tree is $O(\log n)$. Although this

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 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

To develop some intuition for the span analysis, let's consider the probability that we split the input sequence more or less evenly. If we select a pivot that is greater than $t_{n/4}$ and less than $t_{3n/4}$ then $X(n)$ 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.



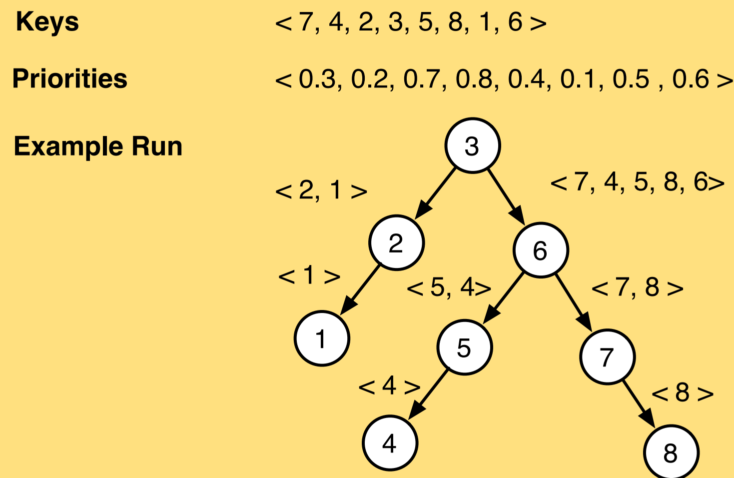
This observations implies that at each level of the call tree (every time a new pivot is selected), the size of the input to both calls decrease by a constant fraction (of $3/4$). At every two levels, the probability that the input size decreases by $3/4$ is the probability that it decreases at either step, which is at least $1 - \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$, etc. More generally, after m such steps, the probability that the input size decreases by a factor of $3/4$ is $1 - \frac{1}{2}^m$. Thus the probability that the input size decreases by a factor of $3/4$ approaches 1 quickly. For example if $m = 10$ then this probability is 0.999. Thus we can conclude that quicksort behaves like a balanced divide and conquer algorithm and it should complete after $c \log n$ levels for some constant c.

We now make this intuition more precise. There are many methods of analysis that we can use. In the rest of this section, we consider one in detail, which is based on counting, and outline another, which is based establishing a recurrence, which can then be solved.

For the analysis, we assume a priority-based selection technique for pivots. At the start of the algorithm, we assign 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 5 the key with the highest priority. Notice that once the priorities are decided, the algorithm is completely deterministic. In addition, 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 a_1, a_2, and a_3).

Exercise 11.9. Rewrite the quicksort algorithm so to use the comparison once when comparing the pivot with each key at a recursive call.

Example 11.10. Quicksort with priorities and its call tree, which is a binary-search-tree, illustrated.



Exercise 11.11. 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 not sent as part of the input to a recursive call, a key is selected to be a pivot at most once.
- Each key is selected to be pivot.

Based on these observations, we conclude that each pair of keys is compared at most once. [1]

Expected work for 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

$$Y(n) \quad = \quad \text{number of comparisons } \texttt{quicksort} \text{ makes on input of size } n,$$

[1] We need only the first two observations to establish this conclusion.

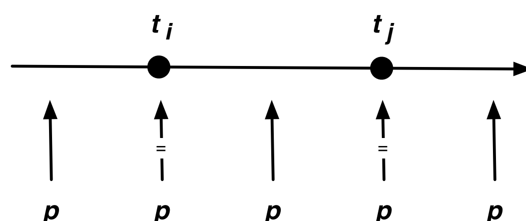


Figure 11.2: The possible relationships between the selected pivot p, t_i and t_j illustrated.

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

Consider the final sorted order of the keys $t = \texttt{sort}(a)$ and let p_i be the priority we chose for the element t_i. Consider two positions $i, j \in \{1, \ldots, n\}$ in the sequence t and define following random variable

$$X_{ij} = \begin{cases} 1 & \text{if } t_i \text{ and } t_j \text{ are compared by quicksort} \\ 0 & \text{otherwise} \end{cases}$$

Since in any run of quicksort, each pair of keys is compared at most once, $Y(n)$ is equal to the sum of all X_{ij}'s, i.e.,

$$Y(n) \leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

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

$$\mathbf{E}[Y(n)] \leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}[X_{ij}]$$

Since each X_{ij} is an indicator random variable, $\mathbf{E}[X_{ij}] = \mathbf{Pr}[X_{ij} = 1]$.

To compute the probability that t_i and t_j are compared (i.e., $\mathbf{Pr}[X_{ij} = 1]$), let's take a closer look at the quicksort algorithm to gather some intuitions. Notice that the first recursive call 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 11.2) for X_{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 $X_{ij} = 1$.
- The pivot is element between t_i and t_j, in which case t_i is in a_1 and t_j is in a_3 and t_i and t_j will never be compared and $X_{ij} = 0$.

- The pivot is less than t_i or greater than t_j. Then t_i and t_j are either both in a_1 or both in a_3, respectively. Whether t_i and t_j are compared will be determined in some later recursive call to `quicksort`.

With this intuition in mind, we can establish the following claim.

Claim 11.12. 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 selected as pivot, t_i and t_j will be compared. This proves the first half of the claim.

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 highest is $2/(j - i + 1)$. Therefore,

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

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 pivot tree. One of t_i and t_{i+1} must be an ancestor of the other in the pivot tree: there is no other 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. Regardless, t_i and t_{i+1} will be compared.

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 pivot tree.

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

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

Note that in the derivation of the asymptotic bound, we used the fact that $H_n = \ln n + O(1)$.

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.

Expected 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 `select` we considered in Section **??**. We refer to the ith node of the pivot tree as the node corresponding to the ith smallest key. This is also the ith node in an in-order traversal.

Claim 11.13. The path from the root to the ith node of the pivot tree is the same as the steps of `select` on $k = i$. That is to the say that the distribution of pivots selected along the path and the sizes of each problem is identical.

The reason this is true, is that `select` is the same as `quicksort` except that it only goes down one of the two recursive branches—the branch that contains the kth key. Recall that for `select`, we showed that the length of the path is more than $10 \lg n$ with probability at most $1/n^{3.15}$. This means that the length of any path being longer that $10 \lg n$ is tiny. This does not suffice to conclude, however, that there are no paths longer than $10 \lg n$, because there are many paths in the pivot tree, and because we only need one to be long to impact the span. Luckily, we don't have too many paths to begin with. We can take advantage of this property by using the union bound, which says that the probability of the union of a collection of events is at most the sum of the probabilities of the events. To apply the union bound, consider the event that

the depth of a node along a path is larger $10 \lg n$, which is $1/n^{3.5}$. The total probability that any of the n leaves have depth larger than $10 \lg n$ is

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

We thus have our high probability bound on the depth of the pivot tree.

The overall span of randomized `quicksort` is therefore $O(\log^2 n)$ with high probability. As in `select`, we can establish an expected bound by using the total expectation theorem. We leave this as an exercise to the reader.

Alternative Analysis. 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 $Y(n)$ done by `quicksort` is then:

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

where the random variable $Y(n)$ is the size of the set a_1 (we use $X(n)$ instead of Y_n to avoid double subscripts). We can now write an equation for the expectation of $X(n)$.

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

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.

We can use a similar strategy to analyze span. Recall that in randomized `quicksort`, at each recursive call, we partition the input sequence a of length n into three subsequences a_1, a_2, and a_3, such that elements in the subsequences are less than, equal, and greater than the pivot, respectfully. Let the random variable $X(n) = \max\{|a_1|, |a_2|\}$, 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 $|a_2| = 0$, as more equal elements will only decrease the span. As this partitioning uses `filter` we have the following recurrence for span for input size n

$$S(n) = S(X(n)) + O(\log n).$$

For the analysis, we shall condition the span on the random variable denoting the size of the maximum half and apply the total expectation theorem.

$$\mathbf{E}\left[S(n)\right] = \sum_{m=n/2}^{n} \mathbf{Pr}\left[X(n) = m\right] \cdot \mathbf{E}\left[S(n) \mid (X(n) = m)\right].$$

The rest is algebra

$$\begin{aligned}
\mathbf{E}\left[a_n\right] &= \sum_{m=n/2}^{n} \mathbf{Pr}\left[M(n) = m\right] \cdot \mathbf{E}\left[S(n) \mid (M(n) = m)\right] \\
&\leq \mathbf{Pr}\left[X(n) \leq \frac{3n}{4}\right] \cdot \mathbf{E}\left[S(\frac{3n}{4})\right] + \mathbf{Pr}\left[X(n) > \frac{3n}{4}\right] \cdot \mathbf{E}\left[S(n)\right] + c \cdot \log n \\
&\leq \frac{1}{2}\,\mathbf{E}\left[S(\frac{3n}{4})\right] + \frac{1}{2}\,\mathbf{E}\left[S(n)\right] \\
&\implies \mathbf{E}\left[S(n)\right] \leq \mathbf{E}\left[S(\frac{3n}{4})\right] + 2c\log n.
\end{aligned}$$

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

Remark 11.14. 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.

We note that our presentation of quicksort algorithm shown in Algorithm 11.7 differs from Hoare's original version which sequentially partitioned the input by using two fingers that moved from each end and by swapping two keys whenever a key was found on the left greater than the pivot and on the right less than the pivot.