

Randomized Algorithms: Quicksort

1 Quicksort

You've certainly seen the **Quicksort** algorithm, possibly the most famous algorithm in all of computer science. Quicksort sorts a sequence by selecting a *pivot* element, and then *partitioning* the sequence into the elements that are (a) less than the pivot, (b) equal to the pivot, and (c) greater than the pivot. After recursively sorting the less and greater parts, the input is sorted.

While standard Quicksort is usually described as sequential, and introductory coverage of it typically focuses on deterministic pivot selection rules, we can make this simple algorithm substantially more interesting by making it both **randomized and parallel!**

Algorithm: Parallel Randomized Quicksort

```
fun quicksort(S : sequence<T>) -> sequence<T>:
  if length(S) <= 1:
    return S

  p = uniform_random_element(S)    // pivot is selected randomly
  L = filter(fn x => (x < p), S)
  R = filter(fn x => (x > p), S)
  M = filter(fn x => (x == p), S)

  sortedL, sortedR = parallel(quicksort(L), quicksort(R))
  return sortedL + M + sortedR
```

The three calls to `filter` make this a *functional* implementation of Quicksort rather than the common (sequential) in-place implementation, since doing things in place in parallel is very difficult. Each filter has linear work and logarithmic span.

Quicksort is closely related to Quickselect which we covered last lecture. Indeed, just about every algorithms courses teach Quicksort *first*, and then Quickselect as a modification of Quicksort. The reason that we do it in this unconventional order will become apparent soon. It turns out that we can *reuse* the analysis of Quickselect's span when analyzing Quicksort's span!

In this class, we are of course interested in analyzing the work and span of Quicksort. Since the cost depends on the random pivots, the costs are *random variables*, not just functions of n , the input size. Therefore, we must analyze some measure of those random variables. As usual, our tools are *expected cost* and *high probability bounds*.

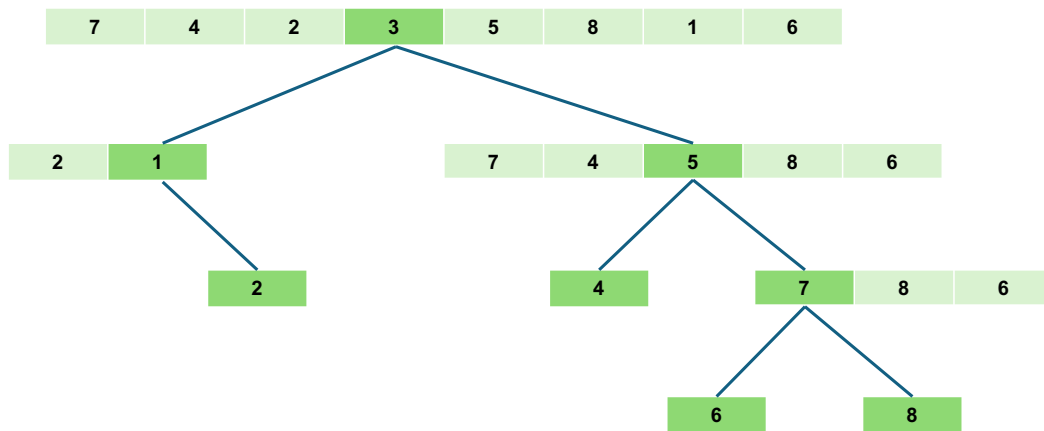
Throughout this lecture we will assume that the comparison function among elements of S has $O(1)$ work and span. Under this assumption, we will show that the work is $\Theta(n)$ in expectation, and that the span is $O(\log^2 n)$ w.h.p.

1.1 Pivot Trees

A useful way to visualize a run of Quicksort is with its *pivot tree*. The pivot tree records the recursive structure of the algorithm. Consider a call to quicksort on a sequence S .

- If $|S| = 0$, its pivot tree is empty.
- If $|S| = 1$, its pivot tree consists of a single node containing the only element of S .
- Otherwise, let p be the pivot selected by quicksort, and let L and R be the elements smaller and larger than p . The root of the pivot tree is p , its left subtree is the pivot tree for L , and its right subtree is the pivot tree for R .

Thus, assuming distinct keys, every element of the input appears exactly once in the pivot tree. The tree captures which recursive subproblem contains each element and how many recursive calls are made along the path leading to it.



The figure above shows one run of Quicksort on

$$S = [7, 4, 2, 3, 5, 8, 1, 6].$$

The first pivot is 3, so 3 becomes the root of the pivot tree. The recursive calls operate on

$$L = [2, 1] \quad \text{and} \quad R = [7, 4, 5, 8, 6].$$

Their recursive choices form the left and right subtrees, respectively. The *depth* of the pivot tree is the maximum number of nodes on a root-to-leaf path. In the example above, the depth is 4.

The pivot tree will later be useful for analyzing the span of Quicksort: a root-to-leaf path corresponds to a chain of recursive calls that must execute sequentially. The span is therefore bounded by the depth of the pivot tree (which is a random variable) multiplied by the span of one level, which is bounded by $O(\log n)$ from the filter calls.

2 Expected Work Analysis

2.1 Cost Setup

We begin by simplifying the analysis.

Assume all keys are distinct. Write the elements in sorted order as

$$k_0 < k_1 < \dots < k_{n-1}.$$

This assumption is without loss of generality for an upper bound. If several elements are equal to the pivot, they are all placed into the middle group M and Quicksort does not recurse on them. Thus duplicate keys can only reduce the cost of the algorithm. Therefore, analyzing the no-duplicates case is the *worst-case*, which is what we are interested in.

Count comparisons. In one call to quicksort, each element of the current subproblem is compared to the pivot a constant number of times by the three filter operations. Therefore, up to a constant factor, the work of Quicksort is the total number of elements that each pivot is compared to at any point during the algorithm.

So, to compute the expected work of Quicksort, it suffices to compute *the expected number of pairs of elements that are ever compared*. Counting the number of comparisons is also independently interesting as a cost model, and is frequently used in algorithm analysis to compute *lower bounds*, i.e., proofs that for a particular problem, there can not exist an algorithm that solves it more efficiently than some particular cost bound. You will see this if you take 15-451.

Comparison indicators. Since our goal is to count the number of pairs of elements that are ever compared, we use our favourite tool for counting events inside a randomized algorithm: **indicator random variables**. We define, for each pair $0 \leq i < j < n$:

$$X_{i,j} = \begin{cases} 1 & \text{if } k_i \text{ and } k_j \text{ are ever compared,} \\ 0 & \text{otherwise.} \end{cases}$$

If X denotes the number of pairs of keys that are compare, then

$$X = \sum_{0 \leq i < j < n} X_{i,j}.$$

By linearity of expectation,

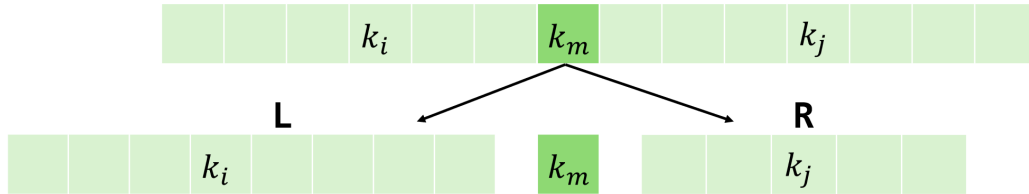
$$\mathbb{E}[X] = \sum_{0 \leq i < j < n} \mathbb{E}[X_{i,j}] = \sum_{0 \leq i < j < n} \Pr[X_{i,j} = 1].$$

Thus, our goal now reduces to computing this probability $\Pr[X_{i,j} = 1]$.

2.2 When Are Two Keys Compared?

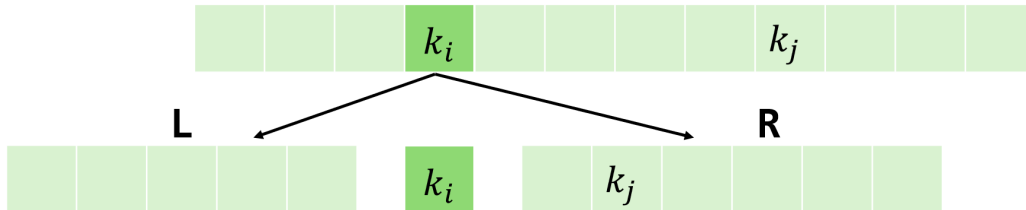
Consider an arbitrary pair of keys $k_i, k_j \in S$, where $i < j$. In what circumstances are k_i and k_j compared? It all depends on the chosen pivot. There are **four scenarios** that can happen each time a pivot is chosen, each of which gives us some of the answer to our question.

Case: k_i and k_j get separated. Suppose that Quicksort chooses a pivot k_m with $i < m < j$, i.e., the pivot is *in between* k_i and k_j , so k_i goes in the left part and k_j goes in the right part.



Then k_i and k_j are sent to different recursive subproblems, so they will **never** be compared.

Case: k_i is chosen as the pivot. If k_i is chosen as the pivot, it will be compared to k_j since Quicksort compares the pivot to *every element*.



Case: k_j is chosen as the pivot. The same argument holds symmetrically if k_j is chosen as the pivot. It will be compared to k_i because it is compared to every element.

Case: None of the above. If the pivot has rank $m < i$ or $m > j$, then k_i and k_j are not compared *in this recursive call*, but both will end up together in the *next recursive call* because they are both in the left part or both in the right part. So, the problem hasn't changed, and we simply consider the future recursive calls until eventually the pivot with rank $i \leq m \leq j$ is chosen.

Thus we have the following lemma.

Lemma: What triggers a comparison?

The keys k_i and k_j are compared if and only if the first pivot chosen from the range

$$k_i, k_{i+1}, \dots, k_j$$

is either k_i or k_j .

Remark: Why can we ignore pivots outside $k_i, \dots, k_j$?

We argued above that all that matters is the *first* pivot chosen from $k_i, k_{i+1}, \dots, k_j$. Why don't pivots chosen outside this range affect the probability?

Suppose the next pivot has rank $m < i$. Then both k_i and k_j belong to the right recursive subproblem. Similarly, if $m > j$, both belong to the left recursive subproblem. In either case, all of $k_i, k_{i+1}, \dots, k_j$ remain together in the same recursive subproblem. We have discarded some *irrelevant keys*, but nothing has happened yet that affects whether k_i and k_j will eventually be compared.

We can therefore ignore pivots outside the range and simply wait until the first pivot whose rank lies between i and j . At that point, all of $k_i, \dots, k_j$ are still present in the same recursive subproblem. Conditioned on choosing a pivot from this range, each of these keys is equally likely to be chosen. Therefore, the first pivot chosen from the range is uniformly distributed among $k_i, k_{i+1}, \dots, k_j$.

2.3 Calculating the Comparison Probability

We now use the lemma from the previous section to compute the probability that a fixed pair k_i and k_j , where $i < j$, is compared. We argued that k_i and k_j are compared if and only if the first pivot chosen from $k_i, k_{i+1}, \dots, k_j$ is either k_i or k_j .

There are $j - i + 1$ keys in this range, and as discussed above, each is equally likely to be the first pivot chosen from the range. Exactly two of these choices, k_i and k_j , cause the pair to be compared. Therefore,

$$\Pr[X_{i,j} = 1] = \frac{2}{j - i + 1}.$$

Thus we have the following corollary.

Corollary: Comparison Probability

For any $0 \leq i < j < n$,

$$\Pr[X_{i,j} = 1] = \frac{2}{j - i + 1}.$$

Intuitively, *nearby keys are more likely to be compared*. In particular, if $j = i + 1$, then k_i and k_j are compared with probability 1. Since no key lies between them, their relative order cannot be inferred transitively from comparisons with other keys; they must be compared directly.

2.4 Expected Number of Comparisons

Let X be the number of pairs of keys that are compared during Quicksort. From the indicator random variables defined above, this was equal to

$$X = \sum_{0 \leq i < j < n} X_{i,j}.$$

By linearity of expectation,

$$\begin{aligned}\mathbb{E}[X] &= \sum_{0 \leq i < j < n} \mathbb{E}[X_{i,j}] \\ &= \sum_{0 \leq i < j < n} \Pr[X_{i,j} = 1] \\ &= \sum_{0 \leq i < j < n} \frac{2}{j-i+1}.\end{aligned}$$

The random variables $X_{i,j}$ are not independent, but linearity of expectation does not require independence! We can evaluate this sum by fixing i and summing over all $j > i$:

$$\mathbb{E}[X] = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1}.$$

Now make the change of variables $k = j - i + 1$. As j ranges from $i + 1$ to $n - 1$, k ranges from 2 to $n - i$. Therefore,

$$\begin{aligned}\mathbb{E}[X] &= 2 \sum_{i=0}^{n-1} \sum_{k=2}^{n-i} \frac{1}{k} \\ &= 2 \sum_{i=0}^{n-1} (H_{n-i} - 1),\end{aligned}$$

where H_n is the n^{th} *harmonic number* (recall last lecture, and recall the important fact that $\ln n < H_n < \ln n + 1$). For an upper bound, $H_{n-i} \leq H_n$ for every i , and hence

$$\mathbb{E}[X] \leq 2nH_n \in O(n \log n).$$

To show that this is tight (i.e., we'd prefer a big- Θ rather than just big- O), it is enough to observe that for the first half of the values of i , we have $n - i \geq n/2$. Therefore each of the corresponding terms $H_{n-i} - 1$ is at least $H_{n/2} - 1 > \ln(n/2) - 1 \in \Theta(\log n)$, and there are $n/2 \in \Theta(n)$ such terms.

Hence $\mathbb{E}[X] \in \Omega(n \log n)$, and combining the upper and lower bounds, $\mathbb{E}[X] \in \Theta(n \log n)$. Since, as argued in Section 2.1, the work of Quicksort is within a constant factor of its number of comparisons, we can conclude:

Theorem: Cost of Parallel Randomized Quicksort

Parallel Randomized Quicksort costs $\Theta(n \log n)$ worst-case expected work.

3 High Probability Span Analysis

We have now shown that Parallel Randomized Quicksort costs $\Theta(n \log n)$ expected work. To analyze the span, we need very different techniques. Specifically, we need to understand the *recursion depth*, since that is the main factor influencing the span.

3.1 Why Expectation Is Not Enough

At each level of recursion, the span is dominated by the filter calls, which have $\Theta(\log |S|)$ span. Although $|S|$ *does* decrease in expectation at each level, it doesn't decrease fast enough for the span to be root dominated, since $\log(n/c)$ for any constant c is $\log n - \log c$, which is asymptotically *not* a constant fraction of $\log n$. Therefore, if D is the recursion depth, then the total span of Quicksort is $O(D \log n)$.

At first glance, it might seem natural to analyze the *expected depth* of each node in the recursion tree. But this is not enough. For example, consider n random variables $D_0, \dots, D_{n-1}$ such that exactly one of them has value n , while all others have value $\log_2 n$, and the unlucky one is chosen uniformly at random. Then, for each fixed index i ,

$$\mathbf{E}[D_i] = \frac{1}{n} \cdot n + \frac{n-1}{n} \log_2 n \leq \log_2 n + 1 = \Theta(\log n).$$

Thus, every individual random variable has low expected value, but the expected maximum value is still n . More generally,

$$\mathbf{E}\left[\max_{0 \leq i < n} D_i\right] \neq \max_{0 \leq i < n} \mathbf{E}[D_i].$$

So, even if every individual node has low expected depth, one unlucky node can still make the whole recursion tree much deeper. To analyze the span, we therefore need a stronger guarantee on each individual path, one that remains true when we consider *all* of the paths together. This motivates our use of *high probability* bounds.

3.2 Why High Probability

Recall that we say a random variable $D(n)$ is at most $f(n)$ *with high probability* (w.h.p.) if there exists a constant n_0 such that for every integer $k \geq 1$ and every $n \geq n_0$,

$$\Pr[D(n) \leq k \cdot f(n)] \geq 1 - \frac{1}{n^k}.$$

Equivalently, in the positive sense, $D(n)$ exceeds $f(n)$ with low probability:

$$\Pr[D(n) > k \cdot f(n)] \leq \frac{1}{n^k}.$$

The important feature for us is that this guarantee holds for *every* k . Thus, if we need a smaller failure probability, we can increase k by a constant amount and obtain more “slack.”

This will be critical when combined with our final tool, the *union bound*, which says for any events $A_1, \dots, A_m$,

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

Just like linearity of expectation, the union bound does not require the events to be independent.

Suppose every fixed node has depth at most $f(n)$ w.h.p. Then, using the definition with $k + 1$,

$$\Pr[D_i > (k + 1)f(n)] \leq \frac{1}{n^{k+1}}.$$

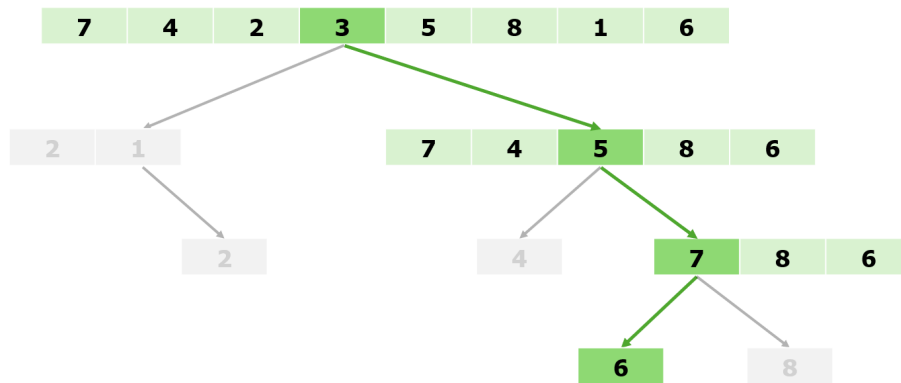
There are at most n nodes, so by the union bound,

$$\begin{aligned} \Pr\left[\max_i D_i > (k + 1)f(n)\right] &\leq \sum_i \Pr[D_i > (k + 1)f(n)] \\ &\leq n \cdot \frac{1}{n^{k+1}} \\ &= \frac{1}{n^k}. \end{aligned}$$

Then, since $(k + 1)f(n) \leq 2kf(n) = k(2f(n))$, the definition of high probability implies that if every fixed node has depth $f(n)$ w.h.p., then the maximum depth over all nodes is $2f(n)$ w.h.p.

3.3 Quicksort Recursion Depth

We now return to the pivot tree, which corresponds to the sequence of recursive calls made by Quicksort. For each fixed key k_i , let D_i be the depth of its node in the pivot tree, i.e., the chain of recursive calls leading to either the base case $S = [k_i]$, or a recursive call in which k_i is the pivot.



This path is *exactly* the same sequence of recursive subproblems that *Quickselect* would visit when searching for k_i , assuming the same random pivot choices. At each level, both algorithms choose the same pivot and recurse into the unique side containing k_i .

Recall from the previous lecture that *quickselect* terminates after at most $5 \log_2 n$ recursive calls with high probability. Therefore, for every fixed key k_i , the corresponding node of the Quicksort pivot tree has depth at most $5 \log_2 n$ w.h.p.

From the previous discussion, if every fixed node has depth at most $f(n)$ w.h.p., then the maximum depth over all nodes is at most $2f(n)$ w.h.p. Taking $f(n) = 5 \log_2 n$, we conclude that the maximum recursion depth of Quicksort is at most $10 \log_2 n$ w.h.p.

Theorem: Quicksort Recursion Depth

The recursion depth of Parallel Randomized Quicksort is $O(\log n)$ with high probability.

3.4 Span of Quicksort

Recall that each level of the recursion costs $\Theta(\log n)$ span, dominated by the calls to `filter`. If D is the maximum recursion depth, then $S(n) \in O(D \log n)$.

We have just shown that $D \in O(\log n)$ w.h.p. Therefore, $S(n) \in O(\log^2 n)$ w.h.p.

Theorem: Span of Parallel Randomized Quicksort

Parallel Randomized Quicksort costs $O(\log^2 n)$ span with high probability.

And that concludes our analysis of Quicksort, an algorithm you've surely seen, possibly many times, but probably never at this level of detail! To analyze Parallel Randomized Quicksort, we used two very different probabilistic techniques.

Remark: Summary

- To analyze the *expected work*, we counted comparisons using **indicator random variables** and **linearity of expectation**. This gave

$$\mathbb{E}[W(n)] \in \Theta(n \log n).$$

- To analyze the *span with high probability*, we observed that the recursion chain for each fixed key behaves like `quickselect`. The depth of each fixed path is $O(\log n)$ w.h.p., and a **union bound** lets us extend this to the maximum depth over all nodes. Therefore,

$$S(n) \in O(\log^2 n) \quad \text{w.h.p.}$$

Theorem: Cost of Parallel Randomized Quicksort

Parallel Randomized Quicksort costs $\Theta(n \log n)$ worst-case expected work and $O(\log^2 n)$ span with high probability.