Chapter 35

The Quick Sort Algorithm

This chapter presents an analysis of the randomized Quick sort algorithm. The first section presents the quicksort algorithm and its variants. The second section analyzes the randomized variant.

1 Quicksort

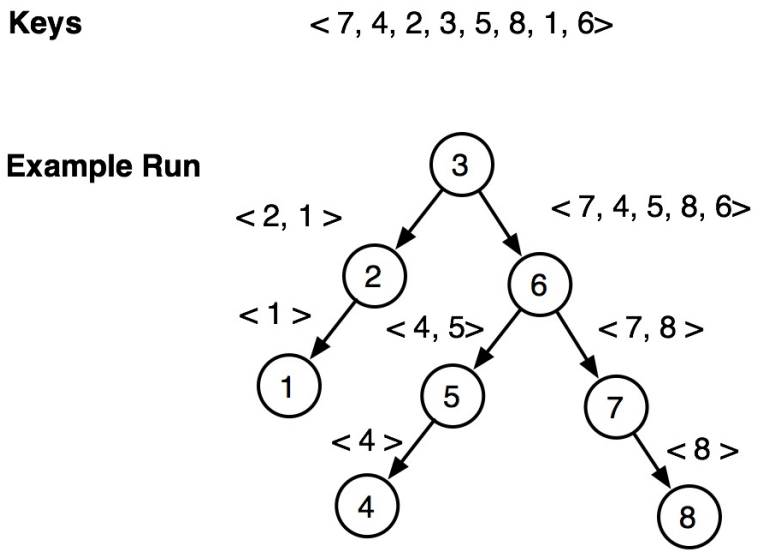
Algorithm 35.1 (Generic Quicksort)**.** The code below show the quicksort algorithm, where the pivot-choosing step is intentionally left under-specified. Quicksort exposes plenty of parallelism:

- the two recursive recursive calls are parallel, and
- the filters for selecting elements greater, equal, and less than the pivot are also internally parallel.

```
1   quicksort a =
2     if |a| = 0 then a
3     else
4       let
5         p = pick a pivot from a
6         a1 = ⟨ x ∈ a | x < p ⟩
7         a2 = ⟨ x ∈ a | x = p ⟩
8         a3 = ⟨ x ∈ a | x > p ⟩
9         (s1, s3) = (quicksort a1) || (quicksort a3)
10      in
11        s1++a2++s3
12      end
```

Pivot Tree. Each call to *quicksort* either makes no recursive calls (the base case) or two recursive calls. We can therefore represent an execution of *quicksort* as a binary search tree, where the nodes—tagged with the pivots chosen—represent the recursive calls and the edges represent the caller-callee relationships. We refer to such a tree as a ***pivot tree*** and use pivot trees to reason about the properties of *quicksort* such as its work and span.

Example 35.1 (Pivot Tree)**.** An example run of *quicksort* along with its pivot tree. In this run, the first call choses 3 as the pivot.



Pivot Tree, Work, and Span. Given a pivot tree for a run of quicksort, we can bound the work and the span of that run by inspecting the pivot tree. Note that for an sequence of length n, the work at each call to *quicksort* is $\Theta(n)$ excluding the calls to the recursive calls, because the *filter* calls each take $\Theta(n)$ work. Similarly, the span is $\Theta(\lg n)$, because *filter* requires $\Theta(\lg n)$ span. Thus we can bound work by adding up the instance size of each node in the pivot tree; the instance size is exactly the number of nodes in that subtree. Likewise for span, we can compute the sum of the logarithm of the instance sizes along each path and take their maximum. Or more simply we can bound the span by multiplying the length of the longest path by $\Theta(\lg n)$.

Impact of Pivot Selection on Work and Span. Because the chosen pivots determine exactly how balanced the pivot tree is, they can have a significant impact on the work and span of the *quicksort* algorithm. Let's consider several alternatives and their impact on work and span.

- **Always pick the first element as pivot:** In this case, inputs that are sorted or nearly sorted can lead to high work and span. For example, if the input sequence is sorted in increasing order, then the smallest element in the input is chosen as the pivot.

This leads to an unbalanced, lopsided pivot tree of depth n. The total work is $\Theta(n^2)$, because $n-i$ keys will remain at level i and the total work is $\sum_{i=0}^{n-1}(n-i-1)$. Similarly, if the sequence is sorted in decreasing order, we will end up with a pivot tree that is lopsided in the other direction. In practice, it is not uncommon for inputs to $quicksort$, or any sort, to be sorted or nearly sorted.

- **Pick the median of three elements as the pivot:** Another strategy is to take the first, middle, and the last elements and pick the median of them. For sorted inputs, this strategy leads to an even partition of the input, and the depth of the tree is $\Theta(\lg n)$. In the worst case, however, this strategy is no better than the first strategy. Nevertheless, this is the strategy used quite broadly in practice.

- **Pick a random element as the pivot:** In this strategy, the algorithm selects one of the elements, uniformly randomly from its input as the pivot. It is not immediately clear what the work and span of this strategy 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 some reason to expect that this strategy could result in a tree of depth $\Theta(\lg n)$ in expectation or with high probability. Indeed, in the analysis section , we prove that selecting a random pivot gives us expected $\Theta(n \lg n)$ work and a recursion tree of depth $\Theta(\lg n)$ leading to $\Theta(\lg^2 n)$ span since each recursive call has $\Theta(\lg n)$ span.

Exercise 35.1. Describe the worst-case input for the version of $quicksort$ that uses the second strategy above.

2 Analysis of Quicksort

This section presents an analysis of the randomized quicksort algorithm which selects its pivots randomly.

Assumptions. We assume that the algorithm is as shown previously and that the pivot is chosen as a uniformly random element of the input sequence.

To streamline the analysis, we make the following two assumptions.

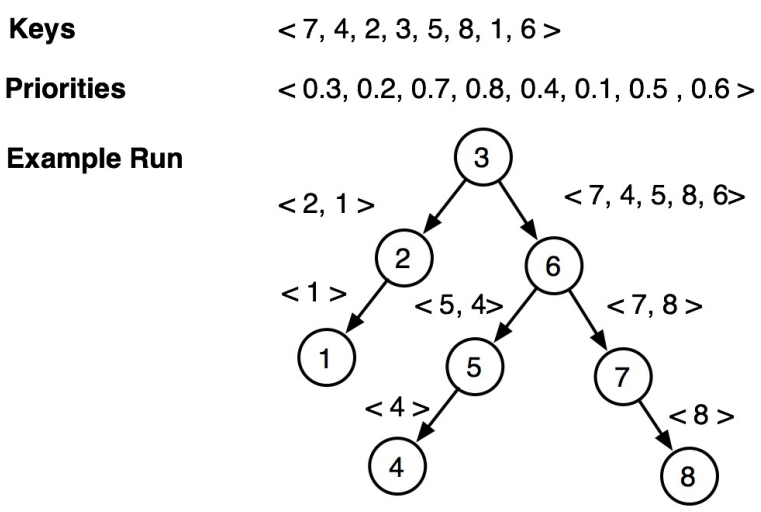
- We "simulate" randomness with priorities: before the start of the algorithm, we assign each key a priority uniformly at random from the real interval $[0, 1]$ such that each key has a unique priority. The algorithm then picks in Line 5 the key with the highest priority.

- We assume a version of $quicksort$ that compares the pivot p to each key in the input sequence once (instead of 3 times).

Notice that once the priorities are decided at the beginning, the algorithm is completely deterministic.

Exercise 35.2. Rewrite the quicksort algorithm to use the comparison once when comparing the pivot with each key at a recursive call.

Example 35.2 (Randomness and Priorities)**.** An execution of *quicksort* with priorities and its pivot tree, which is a binary-search-tree, illustrated.



Exercise 35.3. Convince yourself that the two presentations of randomized *quicksort* are fully equivalent (modulo the technical details about how we might store the priority values).

Roadmap. The rest of this section is organized as follows. First, we present an intuitive analysis. Then, we present a mathematical analysis. And finally, we outline an alternative analysis , which uses recurrence relations.

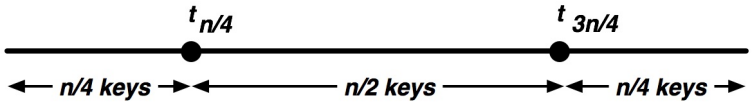
2.1 Intuition

We can reason about the work and span of quicksort in a way similar to the randomized *select* algorithm that we considered in the previous chapter .

Observations. Recall that rank of an element in a sequence is the position of the element in the sorted sequence. Now consider the rank of the pivot selected at a step. If the rank of the pivot is greater $n/4$ and less than $3n/4$ then we say that we are ***lucky***, because the instance size at the next recursive call is at most $3n/4$. Because all elements are equally likely to be selected as a pivot, the probability of being lucky is

$$\frac{3n/4 - n/4}{n} = 1/2.$$

The figure below illustrates this.



Multiple Tries for Better Luck. By the above observation, we know that, if we are lucky then the instance size decreases by a constant fraction ($3/4$ or less).

But what if we are unlucky? Consider two successive recursive calls, the probability that the input size decreases by $3/4$ after two calls is the probability that it decreases at either call, which is at least $1 - \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$. More generally, after $c > 1$ such successive calls, the probability that the input size decreases by a factor of $\frac{3}{4}$ is at least $1 - \frac{1}{2^c}$, which quickly approaches 1 as c increases. For example if $c = 10$ then this probability is 0.999.

In other words, chances of getting unlucky at any given step might be reasonably high $(1/2)$ but chances of getting unlucky over and over again is low, and we only need to get lucky once.

This means that with quite high probability, a constant number c of recursive calls is almost guaranteed to decrease the input size by a constant fraction. Thus we expect $quicksort$ to complete after $k \lg n$ levels for some constant k.

Therefore, intuitively, the total work is $\Theta(n \lg n)$ and the span is $\Theta(\lg^2 n)$ in expectation. But, this is not a proof.

2.2 The Analysis

Observations. 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.
- Because the pivot is not part of the instance solved recursively, a key can become a pivot at most once.
- Each key eventually becomes a pivot, or is a singleton base case.

Based on the first two observations, we conclude that each pair of keys is compared at most once.

2.2.1 Expected Work for Quicksort

Work and Number of Comparisons. We are now ready to analyze the expected work of randomized $quicksort$. Instead of bounding the work directly, we will use a surrogate metric—the number of comparisons—because it is easier to reason about it mathematically. Observe that the total work of $quicksort$ is bounded by the number of comparisons because the work required to partition the input at each recursive calls is asymptotically

bounded by the number of comparisons between the keys and the pivot. To bound the work, asymptotically, it therefore suffices to bound the total number of comparisons.

Random Variables. We define the random variable $Y(n)$ as the number of comparisons *quicksort* makes on input of size n. For the analysis, we will find an upper bound on $\mathbf{E}[Y(n)]$. In particular we will show a bound of $O(n \lg n)$ irrespective of the the order keys in the input sequence.

In addition to $Y(n)$, we define another random variable X_{ij} that indicates whether keys with rank i and j are compared. More precisely, consider the final sort of the keys $t = sort(a)$ and for any element element t_i. Consider two positions $i, j \in \{0, \ldots, n-1\}$ in the sequence t and define following random variable

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

Total Number of Comparisons. By the observations stated above , we know that in any run of *quicksort*, each pair of keys is compared at most once. Thus $Y(n)$ is equal to the sum of all X_{ij}'s, i.e.,

$$Y(n) \leq \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}$$

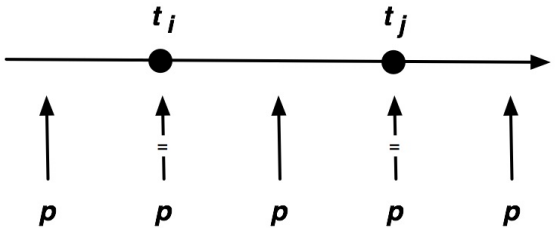
In the sum above, we only consider the case of $i < j$, because we only want to count each comparison once.

By linearity of expectation, we have

$$\mathbf{E}[Y(n)] \leq \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \mathbf{E}[X_{ij}]$$

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

Calculating $\mathbf{P}[X_{ij}]$. To compute the probability that t_i and t_j are compared (i.e., $\mathbf{P}[X_{ij} = 1]$), let's take a closer look at the *quicksort* algorithm and consider the first pivot p selected by the algorithm. The drawing below illustrates the possible relationships between the pivot p and the elements t_i and t_j.



Notice first first that the pivot p is the element with highest priority. For X_{ij}, where $i < j$, we distinguish between three possible scenarios as illustrated in the drawing above:

- $p = t_i$ or $p = t_j$; in this case t_i and t_j are compared and $X_{ij} = 1$.
- $t_i < p < t_j$; in this 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 $X_{ij} = 0$.
- $p < t_i$ or $p > t_j$; in this case 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 $quicksort$.

Note now that the argument above applies to any recursive call of $quicksort$.

Lemma 35.1 (Comparisons and Priorities)**.** For $i < j$, let t_i and t_j be the keys with rank i and j, and p_i or p_j be their priorities. The keys t_i and t_j are compared if and only if either p_i or p_j has the highest priority among the priorities of keys with ranks $i \dots j$.

Proof. For the proof, let p_i be the priority of the key t_i with rank i.

Assume first that t_i (t_j) has the highest priority. In this case, all the elements in the subsequence $t_i \dots t_j$ will move together in the pivot 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.

□

Bounding $\mathbf{E}[Y(n)]$. 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,

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

We can write the expected number of comparisons made in randomized *quicksort* is

$$\begin{aligned}
\mathbf{E}[Y(n)] &\leq \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \mathbf{E}[X_{ij}] \\
&= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1} \\
&= \sum_{i=0}^{n-1} \sum_{k=2}^{n-i} \frac{2}{k} \\
&\leq 2 \sum_{i=0}^{n-1} H_n \\
&= 2nH_n \in O(n \lg n).
\end{aligned}$$

Note that in the derivation of the asymptotic bound, we used the fact that $H_n = \ln n + O(1)$. Recall that $H_n = \sum_{k=1}^{n} \frac{1}{k}$ is the "harmonic number" for n.

Note. Indirectly by Lemma 35.1, we have also shown that the average work for the "basic" deterministic *quicksort*, which always pick the first element as pivot, is also $\Theta(n \lg n)$. Just shuffle the input sequence 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 $O(n \lg n)$ on average.

Remark (Ranks and Comparisons). The bound

$$\mathbf{E}[X_{ij}] = \frac{2}{j-i+1}$$

indicates that the closer two keys are in the sorted order (t) the more likely it is that they are compared. It is instructive to interpret this in the context of pivot tree.

For example, the keys t_i is compared to t_{i+1} with probability 1. Indeed, one of t_i and t_{i+1} must be an ancestor of the other in the pivot tree, because 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.

2.2.2 Expected Analysis of Quicksort: Span

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 $O(\lg n)$—needed for the filter. We argue that the depth of the pivot tree is $O(\lg n)$ by relating it to the number of contraction steps of the randomized order statistics algorithm *select* that we considered earlier. For the discussion, we define the i^{th} node of the pivot tree as the node corresponding to the i^{th} smallest key. This is also the i^{th} node in an in-order traversal.

Lemma 35.2 (Quicksort and Order Statistics)**.** The path from the root to the i^{th} 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.

Proof. Note that that *select* is the same as *quicksort* except that it only goes down one of the two recursive branches—the branch that contains the k^{th} 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

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

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

The overall span of randomized *quicksort* is therefore $O(\lg^2 n)$ with high probability. As in *select*, we can establish an expected bound by using Theorem 32.1.

Exercise 35.4. Complete the span analysis of proof by showing how to apply the Total Expectation Theorem.

2.3 Alternative Analysis of Quicksort

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:

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

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

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

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

Span Analysis. 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 $X(n) = \max\{|a_1|, |a_3|\}$, 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

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

For the analysis, we shall condition the span on the random variable denoting the size of the maximum half and apply Theorem 32.1.

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

The rest is algebra

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

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

3 Concluding Remarks

History of quick sort. Quick sort 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 founders of the field of probability theory.

Our presentation of the quick sort algorithm differs from Hoare's original, which partitions the input sequentially by using two fingers that move 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. The analysis covered in this chapter is different from Hoare's original analysis.

It is interesting that while Quick sort is a quintessential example of a recursive algorithm, at the time of its invention, no programming language supported recursion and Hoare went into some depth explaining how recursion can be simulated with a stack.

Remark. In Chapter 38, we will see that the analysis of quick sort 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.