Lecture 18 — Quicksort and Sorting Lower Bounds

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Margaret Reid-Miller — 26 March 2013

Today:

- Continued Borůvka algorithm from last lecture
- Quicksort
- Work analysis of Randomized Quicksort
- Lower Bounds (not sure we will get to this)

1 Quicksort

You have surely seen quicksort before. The purpose of this lecture is to analyze quicksort in terms of both its work and its span. As we will seen in upcoming lectures, the analysis of quicksort 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.

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.

```
1  function quicksort(S) =
2    if |S| = 0 then S
3    else let
4           p = pick a pivot from S
5           S1 = ⟨ s ∈ S | s < p ⟩
6           S2 = ⟨ s ∈ S | s = p ⟩
7           S3 = ⟨ s ∈ S | s > p ⟩
8           (R1, R3) = ( quicksort(S1) || quicksort(S3) )
9      in
10       append(R1, append(S2, R2))
11     end
```

†Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

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.

The question to ask is: *How does the pivot choice effect the costs of quicksort*? It will be useful to consider the function call tree generated by quicksort. Each call to quicksort either makes no recursive calls (the base case) or two recursive calls. The call tree is therefore binary. We will be interested in analyzing the depth of this tree since this will help us derive the span of the quicksort algorithm.

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.
- *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.

We are interested in picking elements at random.

2 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.

[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.

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 $O(n^2)$.

For the analysis of randomized quicksort, we'll consider a completely equivalent algorithm that will be slightly easier to analyze. Before the start of the algorithm, we'll pick for each element a random priority uniformly at random from the real interval $[0, 1]$—and in Line 4 in the above algorithm, we'll instead pick the key with the highest priority. Notice that once the priorities are decided, the algorithm is completely deterministic; you should convince yourself that the two presentations of randomized quicksort are fully equivalent (modulo the technical details about how we might store the priority values).

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

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

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 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.

The crux of the matter is in describing the event $A_{ij} = 1$ in terms of a simple event that we have a handle on. Before we prove any concrete result, 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 for A_{ij}, where $i < j$:

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

Observation 2.1. *If two elements are compared if and only if one of them they will never be compared again in other call.*

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

In the first case above, when two elements are compared, the non-pivot element is part of S_1, S_2, or S_3—but the pivot element is part of S_2, on which we don't recurse. This gives the following observation:

Observation 2.2. *If two elements are compared in a* `quicksort` *call, they will never be compared again in other call.*

Also notice in the corresponding BST, when two elements are compared, the pivot element become the root of two subtrees, one of which contains the other element.

Observation 2.3. *In the quicksort algorithm, two elements are compared in a* `quicksort` *call if and only if one element is an ancestor of the other in the corresponding BST.*

Therefore, with these random variables, we can express the total comparison count X_n as follows:

$$X_n \;\leq\; 3\sum_{i=1}^{n}\sum_{j=i+1}^{n} A_{ij}$$

The constant 3 is because our not-so-optimized quicksort compares each element to a pivot 3 times. By linearity of expectation, we have $\mathbf{E}\left[X_n\right] \leq 3\sum_{i=1}^{n}\sum_{j=i+1}^{n}\mathbf{E}\left[A_{ij}\right]$. 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.

Computing the probability $\mathbf{Pr}\left[A_{ij}=1\right]$. 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 2.4. *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. We'll show this by contradiction. Assume 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_k "sits" between T_i and T_k (i.e., $T_i \leq T_k \leq T_j$) , it will separate T_i and T_j into different buckets, so they are never compared. □

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,

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

Notice that element 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. On the other hand, 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

$$\begin{aligned}
\mathbf{E}\left[X_n\right] &\leq 3\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathbf{E}\left[A_{ij}\right] \\
&= 3\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\frac{2}{j-i+1} \\
&= 3\sum_{i=1}^{n-1} n\sum_{k=2}^{n-i+1}\frac{2}{k} \\
&\leq 6\sum_{i=1}^{n-1} H_n \\
&= 6nH_n \in O(n\log n)
\end{aligned}$$

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.

2.1 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 $X(n)$.

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

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

3 Expected Span of Quicksort

We will postpone the discussion on the expected span of quicksort to a later lecture on randomized binary search trees, as the analysis is closely related, and provides more intuition. It will show that the expected span of quicksort is $O(log^2 n)$. Not only is this the span in expectation, it is the span with high probability. That is, it is highly unlikely that a particular run of quicksort will have a span worse than $O(log^2 n)$.

4 Lower Bounds

After spending time formulating a concrete problem, we might wonder how hard the problem actually is. In this course thus far, our focus has been on obtaining efficient algorithms for certain problems. For a problem P, we try to design efficient algorithms to solve it. The existence of an algorithm gives an upper bound on the complexity of the problem P. In particular, an algorithm A with work (either expected or worst-case) $O(f(n))$ is a constructive proof that P can be solved provided $O(f(n))$ work. This is essentially the upper bound part of the question.

In this lecture, we'll turn the tables, showing that certain problems cannot be solved more efficiently than a given bound. This is the lower bound part of the question. In general, this is a harder task: To establish a lower bound, we have to argue that *no algorithm, however smart, can possibly do better than what we claim*; it is no longer sufficient to exhibit an algorithm A and analyze its performance.

4.1 Sorting and Merging Lower Bounds

Before we look at lower bounds for sorting and merging, let us review the (upper) bounds we have for various sorting algorithms we've covered:

Algorithm	Work	Span
Quick Sort	$O(n \log n)$	$O(\log^2 n)$
Merge Sort	$O(n \log n)$	$O(\log^2 n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$
Balanced BST Sort	$O(n \log n)$	$O(\log^2 n)$

Notice that in this table, all algorithms have $O(n \log n)$ work—and except for heap sort, every algorithm is very parallel ($\log^2 n$ span). Can we sort in less than $O(n \log n)$ work? Probably. But we'll show that *any* deterministic *comparison-based* sorting algorithm must use $\Omega(n \log n)$ comparisons to sort n entries in the worst case. In the comparison-based model, we have no domain knowledge about the entries and the only operation we have to determine the relative order of a pair of entries x and y is a comparison operation, which returns whether $x < y$. More precisely, we'll prove the following theorem:

Theorem 4.1. *For a sequence $\langle x_1, \ldots, x_n \rangle$ of n distinct entries, finding the permutation π on $[n]$ such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$ requires, in the worst case, at least $\frac{n}{2} \log(\frac{n}{2})$ queries to the $<$ operator.*

Since each comparison takes at least constant work, this implies an $\Omega(n \log n)$ lower bound on the work required to sort a sequence of length n in the comparison model.

What about merging? Can we merge sorted sequences faster than resorting them? As seen in previous lectures, we can actually merge two sorted sequences in $O(m \log(1 + n/m))$ work, where m is the length of the shorter of the two sequences, and n the length of the longer one. We'll show, however, that in the comparison-based model, we cannot hope to do better:

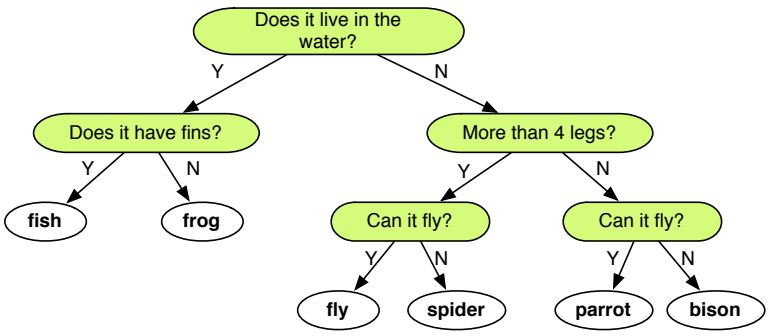
Theorem 4.2. *Merging two sorted sequences of lengths m and n ($m \leq n$) requires at least*

$$m \log_2(1 + \tfrac{n}{m})$$

comparison queries in the worst case.

4.2 Decision Trees or The 20 Questions Game

Let's play game. Suppose I think of an animal but you know for fact it's one of the following: a fish, a frog, a fly, a spider, a parrot, or a bison. You want to find out what animal that is by answering the fewest number of Yes/No questions (you're only allowed to ask Yes/No questions). What strategy would you use? Perhaps, you might try the following reasoning process:



Interestingly, this strategy is optimal: There is no way you could have asked any 2 Yes/No questions to tell apart the 6 possible answers. If we can ask only 2 questions, any strategy that is deterministic and computes the output using only the answers to these Yes/No questions can distinguish between only $2^2 = 4$ possibilities. Thus, using 3 questions is the best one can do.

Determining the minimum number of questions necessary in the worst case in at the crux of many lower-bound arguments. For starters, we describe a way to represent a deterministic strategy for playing such a game in the definition below.

Definition 4.3 (Binary Decision Trees)**.** A *decision tree* is a tree in which

- each leaf node is an answer (i.e. what the algorithm outputs);
- each internal node represents a query—some question about the input instance—and has k children, corresponding to one of the k possible responses $\{0, \ldots, k-1\}$;
- and the answer is computed as follows: we start from the root and follow a path down to a leaf where at each node, we choose which child to follow based on the query response.

The crucial observation is the following: If we're allowed to make at most q queries (i.e., ask at most q Yes/No questions), the number of possible answers we can distinguish is the number of leaves in a binary tree with depth at most q; this is at most 2^q. Taking logs on both sides, we have

If there are N possible outcomes, the number of questions needed is at least $\log_2 N$.

That is, there is *some* outcome, that requires answering at least $\log_2 N$ questions to determine that outcome.

4.3 Warm-up: Guess a Number

As a warm-up question, if I pick a number a between 1 and 2^{20}, how many Yes/No questions you need to ask before you can zero in on a? By the calculation above, since there are $N = 2^{20}$ possible outcomes, you will need *at least*

$$\log_2 N = 20$$

questions in the worst case.

Another way to look at the problem is to suppose I am devious and I don't actually pick a number in advance. Each time you ask a question of the form "is the number greater than x", in effect you are splitting the set of possible numbers into two groups. I always answer so the set of remaining possible numbers has the greater cardinality. That is, each question you ask eliminates at most half of the numbers. Since there are $N = 2^{20}$ possible values, I can force you ask $\log_2 N = 20$ questions before I must concede and pick the last remaining number as my a. This variation of the games shows that no matter what strategy you use to ask questions, there is always *some* a that would cause you to ask a lot of questions.

4.4 A Sorting Lower Bound

Let's turn back to the classical sorting problem. We will prove Theorem 4.1. This theorem follows almost immediately from our observation about k-ary decision trees. There are $n!$ possible permutations, and to narrow it down to one permutation which orders this sequence correctly, we'll need $\log(n!)$ queries, so the number of comparison queries is at least

$$\begin{aligned}\log(n!) &= \log n + \log(n-1) + \cdots + \log(n/2) + \cdots + \log 1 \\ &\geq \log n + \log(n-1) + \cdots + \log(n/2) \\ &\geq \tfrac{n}{2} \cdot \log(n/2).\end{aligned}$$

We can further improve the constants by applying Stirling's formula instead of this crude approximation. Remember that *Stirling's formula* gives the following approximation:

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta(n^{-1})\right) > \left(\frac{n}{e}\right)^n,$$

so $\log_2(n!) > n\log_2(n/e)$.

4.5 A Merging Lower Bound

Closely related to the sorting problem is the merging problem: Given two sorted sequences A and B, the merging problem is to combine these sequences into a sorted one. To apply the argument we used for sorting, we'll need to count how many possible outcomes the comparison operation can produce.

Suppose $n = |A|$, $m = |B|$, and $m \leq n$. We'll also assume that these sequences are made up of unique elements. Now observe that we have not made any comparison between elements of A and B. This means any interleaving sequence A's and B's elements is possible. Therefore, the number of possible merged outcomes is the number of ways to choose n positions out from $n+m$ positions to put A's elements; this is simply $\binom{n+m}{n}$. Hence, we'll need, in the worst case, at least $\log_2 \binom{n+m}{n}$ comparison queries to merge these sequences.

The following lemma gives a simple lower bound for $\binom{n}{r}$, so that we can simplify $\log_2 \binom{n+m}{n}$ to an expression that we recognize.

Lemma 4.4 (Binomial Lower Bound)**.**

$$\binom{n}{r} \geq \left(\frac{n}{r}\right)^r.$$

Proof. First, we recall that

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{n(n-1)(n-2)\ldots(n-r+1)}{r(r-1)(r-2)\ldots 1} = \prod_{i=0}^{r-1} \frac{n-i}{r-i}.$$

We'll argue that for $0 \leq i < \min(r, n)$, $\frac{n-i}{r-i} \geq \frac{n}{r}$. Notice that

$$\frac{n-i}{r-i} \geq \frac{n}{r} \iff r(n-i) \geq n(r-i) \iff rn - ri \geq nr - ni \iff (n-r)i \geq 0.$$

Therefore, we have $\binom{n}{r} \geq \prod_{i=0}^{r-1} \frac{n-i}{r-i} \geq (n/r)^r$. □

With this lemma, we conclude that the number of comparison queries needed to merge sequences of lengths m and n $(m \leq n)$ is at least

$$\log_2 \binom{n+m}{m} \geq m \log_2 \left(1 + \tfrac{n}{m}\right),$$

proving Theorem 4.2