Lecture 18 — Quicksort and Sorting Lower Bounds

Parallel and Sequential Data Structures and Algorithms, 15-210 (Qatar-Spring 2014)

Lectured by Kemal Oflazer — 25 March 2014

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, in the way we present it, 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. 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.

Consider the following implementation of quicksort. In this implementation, we intentionally leave the pivot-choosing step unspecified because the property we are discussing holds regardless of the choice of the pivot.

Pseudo Code 1.1 (Quicksort 1/2).

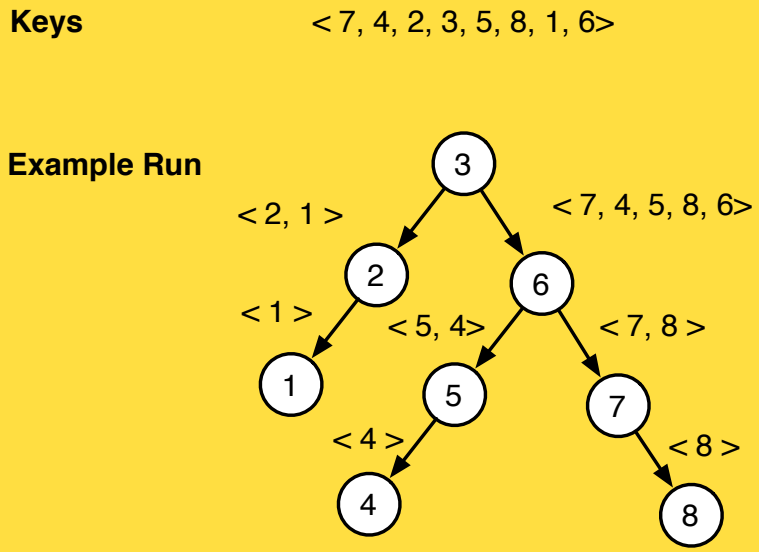
```
1  fun quicksort(S) =
2    if |S| = 0 then S
3    else let
4           p = pick a pivot from S
5           S1 = ⟨s ∈ S | s < p⟩
6           S2 = ⟨s ∈ S | s = p⟩
7           S3 = ⟨s ∈ S | s > p⟩
8           (R1,R3) = (quicksort(S1) || quicksort(S3))
9         in
10          append(R1, append(S2,R3))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

Question 1.2. *Is there parallelism in quicksort?*

†Lecture notes by Umut Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, with additional edits by Kemal Oflazer

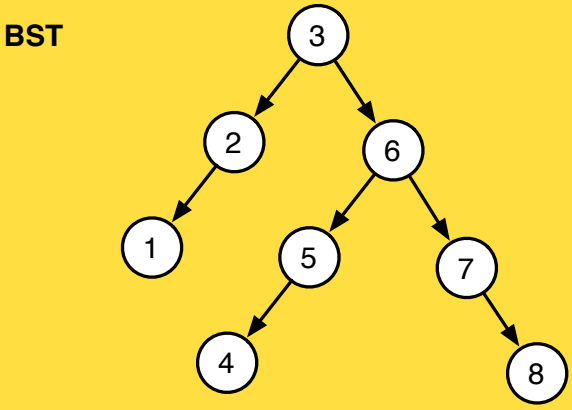
There is clearly plenty of parallelism in this version quicksort.[1] There is both parallelism due to the two recursive calls and in the fact that the filters for selecting elements greater, equal, and less than the pivot can be parallel.

Example 1.3. *An example run of quicksort.*



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) represeting the recursive calls along with the pivots chosen. We will sometimes refer to this tree as the *call tree* or *pivot tree*. We will use this call-tree representation to reason about the properties of quicksort, e.g., the comparisons performed, its span.

Example 1.4. *An example call tree, which is a binary-search-tree, illustrated.*



[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.

Question 1.5. *How does the pivot choice effect the costs of quicksort?*

Let's consider some strategies for picking a pivot:

- *Always pick the first element:* If the sequence is sorted in increasing order, then picking the first element is the same as picking the smallest element. We end up with a lopsided recursion tree of depth n. The total work is $O(n^2)$ since $n-i$ keys will remain at level i and hence we will do $n-i-1$ comparisons at that level for a total of $\sum_{i=0}^{n-1}(n-i-1)$. Similarly, if the sequence is sorted in decreasing order, we will end up with a recursion tree that is lopsided in the other direction. In practice, it is not uncommon for a sort function input to be a sequence that is already sorted or nearly sorted.
- *Pick the median of three elements:* Another strategy is to take the first, middle, and the last elements and pick the median of them. For sorted lists the split is even, so each side contains half of the original size and the depth of the tree is $O(\log n)$. Although this strategy avoids the pitfall with sorted sequences, it is still possible to be unlucky, and in the worst-case the costs and tree depth are the same as the first strategy. This is the strategy used by many library implementations of quicksort. Can you think of a way to slow down a quicksort implementation that uses this strategy by picking an adversarial input?
- *Pick an element randomly:* It is not immediately clear what the depth of this is, but intuitively, when we choose a random pivot, the size of each side is not far from $n/2$ in expectation. This doesn't give us a proof but it gives us hope that this strategy will result in a tree of depth $O(\log n)$ in expectation. Indeed, picking a random pivot gives us expected $O(n \log n)$ work and $O(\log^2 n)$ span for quicksort and an expected $O(\log n)$-depth tree, as we will show.

2 Analysis of Quicksort, the Basics

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

Pseudo Code 2.1 (Quicksort 2/2).

```
 1  fun quicksort(S) =
 2    if |S| = 0 then S
 3    else let
 4           p = pick a pivot from S
 5           C = ⟨ s ∈ S | (s, compare(p, s)) ⟩
 6           S1 = ⟨ s | (s, LESS) ∈ C ⟩
 7           S2 = ⟨ s | (s, EQUAL) ∈ C ⟩
 8           S3 = ⟨ s | (s, GREATER) ∈ C ⟩
 9           (R1, R3) = ( quicksort(S1) || quicksort(S3) )
10         in
11           append(R1, append(S2, R3))
12         end
```

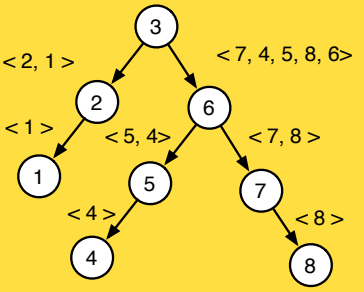
In the analysis, for picking the pivots, we are going to use a priority-based selection technique. Before the start of the algorithm, we'll pick for each key a random priority uniformly at random from the real interval $[0, 1]$ such that each key has a unique priority. We then pick in Line 4 the key with the highest priority. Notice that once the priorities are decided, the algorithm is completely deterministic.

Example 2.2. *Quicksort with priorities and its call tree, which is a binary-search-tree, illustrated.*

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

Priorities < 0.3, 0.2, 0.7, 0.8, 0.4, 0.1, 0.5 , 0.6 >

Example Run



Exercise 1. *Convince yourself that the two presentations of randomized quicksort are fully equivalent (modulo the technical details about how we might store the priority values).*

Before we get to the analysis, let's observe some properties of quicksort. For these observations, it might be helpful to consider the example shown above.

Question 2.3. *How many time any two keys x and y in the input are compared in a run of the algorithm?*

In quicksort, a comparison always involves a pivot and another key. Since, the pivot is never sent to a recursive call, a key is selected as a pivot exactly once, and is not involved in further comparisons (after is becomes a pivot). Before a key is selected a pivot, it may be compared to other pivots, once per pivot, and thus two keys are never compared more than once.

Question 2.4. *Can you tell which keys are compared by looking at just the call tree?*

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

Question 2.5. *Let x and z two keys such that $x < z$. Suppose that a key y is selected as a pivot before either x or z is selected. Are x and z compared?*

When the algorithm selects a key (y) in between two keys (x, z) as a pivot, it sends the two keys to two separate subtrees. The two keys (x and z) seperated in this way are never compared again.

Question 2.6. *Suppose that the keys x and y are adjacent in the sorted order, how many times are they compared?*

Adjacent keys are compared exactly once. This is the case because there is no key that will separate them.

Question 2.7. *Would it be possible to modify quicksort so that it never compares adjacent keys?*

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

Question 2.8. *Based on these, can you bound the number of comparisons performed by quicksort in terms of the comparisons between keys.*

Obviously the total number of comparisons can be written by summing over all comparisons involving each key. Since each key is involved in a comparison with another key at most once, the total number of comparisons can be expressed as the sum over all pairs of keys. That is, we can just consider each pair of key once and check whether they are ancestors/descendants or not and add one of if so. We never have to consider each pair more than once.

3 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

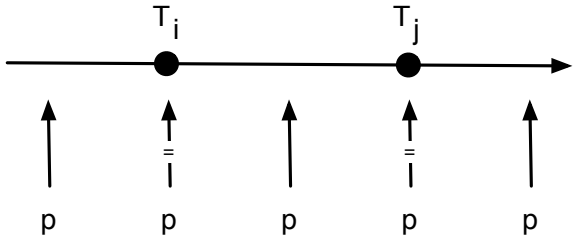


Figure 1: The possible relationships between the selected pivot p, T_i and T_j illustrated.

input this naive version of quicksort works well on average, but can be slow on some (common) inputs.

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

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

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

our goal is to find an upper bound on $\mathbf{E}[X_n]$ 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} $(i < j)$ 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.[3]

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

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

By linearity of expectation, we have

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

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.

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

[3] Note that we index the element positions starting at 1, but this has no bearing on the following analysis.

To compute this probability, let's take a closer look at the quicksort algorithm to gather some intuitions. Notice that the top level takes as its pivot p the element with highest priority. Then, it splits the sequence into two parts, one with keys larger than p and the other with keys smaller than p. For each of these parts, we run `quicksort` recursively; therefore, inside it, the algorithm will pick the highest priority element as the pivot, which is then used to split the sequence further.

For any one call to `quicksort` there are three possibilities (illustrated in Figure 1) 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`.

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 3.1. *For $i < j$, T_i and T_j are compared if and only if p_i or p_j has the highest priority among $\{p_i, p_{i+1}, \ldots, p_j\}$.*

Proof. Assume first that T_i (T_j) has the highest priority. In this case, all the elements in the subsequence $T_i \ldots T_j$ will move together in the call tree until T_i (T_j) is selected as pivot. When it is, T_i and T_j will be compared. This proves the first half of the theorem.

For the second half, assume that T_i and T_j are compared. For the purposes of contradiction, assume that there is a key T_k, $i < k < j$ with a higher priority between them. In any collection of keys that include T_i and T_j, T_k will become a pivot before either of them. Since $T_i \leq T_k \leq T_j$ it will separate T_i and T_j into different buckets, so they are never compared. This is a contradiction; thus we conclude there is no such T_k.

□

Therefore, for T_i and T_j to be compared, p_i or p_j has to be bigger than all the priorities in between. Since there are $j - i + 1$ possible keys in between (including both i and j) and each has equal probability of being the highest, the probability that either i or j is the greatest is $2/(j - i + 1)$. Therefore,

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

Question 3.2. *What does this bound tell us about the likelihood of keys being compared?*

The bound says that the closer two keys in the sorted order, the more likely it is that they are compared. For example, the keys T_i is compared to T_{i+1} with probability 1. It is easy to understand why if we consider the corresponding BST. One of T_i and T_{i+1} must be an ancestor of the other in the BST: There is no element that could be the root of a subtree that has T_i in its left subtree and T_{i+1} in its right subtree.

If we consider T_i and T_{i+2} there could be such an element, namely T_{i+1}, which could have T_i in its left subtree and T_{i+2} in its right subtree. That is, with probability $1/3$, T_{i+1} has the highest probability of the three and T_i is not compared to T_{i+2}, and with probability $2/3$ one of T_i and T_{i+2} has the highest probability and, the two are compared.

In general, the probability of two elements being compared is inversely proportional to the number of elements between them when sorted. The further apart the less likely they will be compared. Analogously, the further apart the less likely one will be the ancestor of the other in the related BST.

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

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

(Recall: $H_n = \ln n + O(1)$.)

Indirectly, we have also shown that the average work for the basic deterministic quicksort (always pick the first element) is also $O(n \log n)$. Just shuffle the data randomly and then apply the basic quicksort algorithm. Since shuffling the input randomly results in the same input as picking random priorities and then reordering the data so that the priorities are in decreasing order, the basic quicksort on that shuffled input does the same operations as randomized quicksort on the input in the original order. Thus, if we averaged over all permutations of the input the work for the basic quicksort is $O(n \log n)$ on average.

3.1 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:

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

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 $X(n)$.

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

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.

4 Expected Span of Quicksort

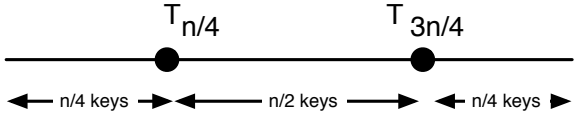
Recall that in randomized quicksort, at each recursive call, we partition the input sequence S of length n into three subsequences L, E, and R, such that elements in the subsequences are less than, equal, and greater than the pivot, respectfully. Let $M_n = \max\{|L|, |R|\}$, which is the size of larger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

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

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

Question 4.1. *What is the probabily that M_n is at most $3n/4$?*

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



Question 4.2. *What does this say about what the span of quicksort may be?*

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

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

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

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

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

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

Now, we can write this trivially as

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

The rest is algebra

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

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

5 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.

5.1 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 5.1. *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 5.2. *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.

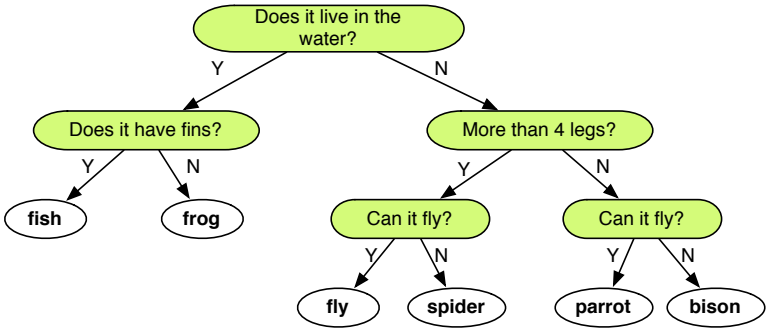
5.2 Decision Trees or The 20 Questions Game

Let's play game. Suppose I think of an animal but don't tell you what it is. You want to find out the animal by asking yes/no questions only?

Question 5.3. *What strategy would you follow? Can you argue for why that strategy is the best?*

Since you don't know the animal on my mind, your best strategy is to ask a question that divides the set of possibilities (the animal kingdom) into two equal halves. This is the best strategy because with any other division, you risk the possibility that my choice is in the larger set.

For example, suppose that I picked one of the following (and you know this): a fish, a frog, a fly, a spider, a parrot, or a bison. 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 5.4 (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.

5.3 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.

5.4 A Sorting Lower Bound

Let's turn back to the classical sorting problem. We will prove Theorem 5.1.

Question 5.5. *Can you see how many comparisons would a sorting algorithm has to perform?*

A sorting algorithm must distinguish between all permutations of the input, because in order to sort each, it has to perform something different. Connecting back to our decision tree diagram, the algorithm needs $\log(n!)$ queries to distinguish the $n!$ possible permutations (outcomes).

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

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

5.5 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 5.6 (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 5.2