Lecture 5

Comparison-based Lower Bounds for Sorting

5.1 Overview

In this lecture we discuss the notion of *lower bounds*, in particular for the problem of sorting. We show that any deterministic comparison-based sorting algorithm must take $\Omega(n \log n)$ time to sort an array of n elements in the worst case. We then extend this result to average case performance, and to randomized algorithms. In the process, we introduce the 2-player game view of algorithm design and analysis.

5.2 Sorting lower bounds

So far we have been focusing on the question: "given some problem X, can we construct an algorithm that runs in time $O(f(n))$ on inputs of size n?" This is often called an upper bound problem because we are determining an upper bound on the inherent difficulty of problem X, and our goal here is to make $f(n)$ as small as possible. In this lecture we examine the "lower bound problem." Here, the goal is to prove that *any* algorithm must take time $\Omega(g(n))$ time to solve the problem, where now our goal is to do this for $g(n)$ as large as possible. Lower bounds help us understand how close we are to the best possible solution to some problem: e.g., if we have an algorithm that runs in time $O(n \log^2 n)$ and a lower bound of $\Omega(n \log n)$, then we have a $\log(n)$ "gap": the maximum possible savings we could hope to achieve by improving our algorithm.

Often, we will prove lower bounds in restricted models of computation, that specify what types of operations may be performed on the input and at what cost. So, a lower bound in such a model means that if we want to do better, we would need somehow to do something outside the model.

Today we consider the class of comparison-based sorting algorithms. These are sorting algorithms that only operate on the input array by comparing pairs of elements and moving elements around based on the results of these comparisons. In particular, let us make the following definition.

Definition 5.1 *A* comparison-based sorting algorithm *takes as input an array $[a_1, a_2, \ldots, a_n]$ of n items, and can only gain information about the items by comparing pairs of them. Each comparison ("is $a_i > a_j$?") returns YES or NO and counts a 1 time-step. The algorithm may also for free* *reorder items based on the results of comparisons made. In the end, the algorithm must output a permutation of the input in which all items are in sorted order.*

For instance, Quicksort, Mergesort, and Insertion-sort are all comparison-based sorting algorithms. What we will show is the following theorem.

Theorem 5.1 *Any deterministic comparison-based sorting algorithm must perform $\Omega(n \log n)$ comparisons to sort n elements in the worst case. Specifically, for any deterministic comparison-based sorting algorithm $\mathcal{A}$, for all $n \geq 2$ there exists an input I of size n such that $\mathcal{A}$ makes at least $\log_2(n!) = \Omega(n \log n)$ comparisons to sort I.*

To prove this theorem, we cannot assume the sorting algorithm is going to necessarily choose a pivot as in Quicksort, or split the input as in Mergesort — we need to somehow analyze *any possible* (comparison-based) algorithm that might exist. The way we will do this is by showing that in order to sort its input, the sorting algorithm is implicitly playing a game of "20 questions" with the input, trying to figure out in what the order its elements are being given.

Proof: Since the algorithm must output a permutation of its input, we can assume the input elements are $\{1, 2, \ldots, n\}$ but in some unknown order. The key to the argument is that (a) two different input orders cannot both be correctly sorted by the same permutation, and (b) there are $n!$ different orders the input elements could be in. Now, suppose that two different initial orderings of these numbers I_1, I_2, are consistent with all the comparisons the sorting algorithm has made so far. Then, the sorting algorithm cannot yet be done since any permutation it outputs at this point cannot be correct for both I_1 and I_2 (by observation (a) above). So, the sorting algorithm needs at least implicitly to have pinned down which ordering of $\{1, \ldots, n\}$ was given in the input.

Let S be the set of input orderings consistent with all answers to comparisons made so far (so, initially, S is the set of all $n!$ possible orderings of the input). We can think of a new comparison as splitting S into two groups: those input orderings for which the answer is YES and those for which the answer is NO. Now, suppose the answer to each comparison is always the one corresponding to the larger group. Then, each comparison cuts down the size of S by at most a factor of 2. Since S initially has size $n!$, and at the end the algorithm must have reduced $|S|$ down to 1, in this case the algorithm will need to make at least $\log_2(n!)$ comparisons before it can halt. We can then solve:

$$\begin{aligned} \log_2(n!) &= \log_2(n) + \log_2(n-1) + \ldots + \log_2(2) \\ &= \Omega(n \log n). \quad \blacksquare \end{aligned}$$

Let's do an example with $n = 3$. In this case, there are six possible input orderings:

$$\{123\}, \{132\}, \{213\}, \{231\}, \{312\}, \{321\}.$$

Suppose the sorting algorithm first compares $A[0]$ with $A[1]$. If the answer is that $A[1] > A[0]$ then we have narrowed down the input to the three possibilities:

$$\{123\}, \{132\}, \{231\}.$$

Suppose the next comparison is between $A[1]$ and $A[2]$. In this case, the most popular answer is that $A[1] > A[2]$, which removes just one ordering, leaving us with:

$$\{132\}, \{231\}.$$

It now takes one more comparison to finally isolate the input ordering.

Notice that our proof is like a game of 20 Questions in which the responder doesn't actually decide what he is thinking of until there is only one option left. This is legitimate because we just need to show that there is *some* input that would cause the algorithm to take a long time. In other words, since the sorting algorithm is deterministic, we can take that final remaining option and then re-run the algorithm on that specific input, and the algorithm will make the same exact sequence of operations.

You can also think of the above proof in terms of the number of possible *outputs* of the sorting algorithm. Any comparison-based sorting algorithm can be thought of as producing a permutation as its output (the permutation that, when applied to the input, produces a sorted array). There are $n!$ permutations and only one of them can be correct for any given input. Each comparison breaks the set of possible outputs into two classes, and the response to the question says which class the correct output is in. By always giving the answer corresponding to the larger class, an adversary can force the algorithm to make $\log_2(n!)$ comparisons.

5.3 Average-case lower bounds

In fact, we can generalize the above theorem to show that any comparison-based sorting algorithm must take $\Omega(n \log n)$ time *on average*, not just in the worst case.

Theorem 5.2 *For any deterministic comparison-based sorting algorithm, the Average-Case number of comparisons (the number of comparisons on average on a randomly chosen input permutation) is at least* $\lfloor \log_2(n!) \rfloor$.

Proof: Let's build out the entire decision tree: the tree we get by looking at all possible series of answers that one might get from some ordering of the input. By the previous argument, each leaf of this tree corresponds to a single input permutation (we can't have two permutations at the same leaf, else the algorithm would not be finished). The depth of the leaf is the number of comparisons performed by the sorting algorithm on that input.

If the tree is completely balanced, then each leaf is at depth $\lceil \log_2(n!) \rceil$ or $\lfloor \log_2(n!) \rfloor$ and we are done.[1] To prove the theorem, we just need to show that out of all binary trees on a given number of leaves, the one that minimizes their average depth is a completely balanced tree. This is not too hard to see: given some unbalanced tree, we take two sibling leaves at largest depth and move them to be children of the leaf of smallest depth. Since the difference between the largest depth and the smallest depth is at least 2 (otherwise the tree would be balanced), this operation reduces the average depth of the leaves. Specifically, if the smaller depth is d and the larger depth is D, we have removed two leaves of depth D and one of depth d, and we have added two leaves of depth $d+1$ and one of depth $D-1$. Since any unbalanced tree can be modified to have a smaller average depth, such a tree cannot be one that *minimizes* average depth, and therefore the tree of smallest average depth must in fact be balanced. ■

In fact, if we are a bit more clever in the proof, we can get rid of the floor in the bound.

[1] Let us define a tree to be completely balanced if the deepest leaf is at most one level deeper than the shallowest leaf. Everything would be easier if we could somehow assume $n!$ was a power of 2....

5.4 Lower bounds for randomized algorithms

Theorem 5.3 *The above bound holds for randomized algorithms too.*

Proof: The argument here is a bit subtle. The first step is to argue that with respect to counting comparisons, we can think of a randomized algorithm $\mathcal{A}$ as a probability distribution over deterministic algorithms. To make things easier, let us only consider algorithms that have some finite upper bound B (like n^2) on the number of random coin-flips they make. This means we can think of $\mathcal{A}$ as having access to a special "random bit tape" with B bits on it, and every time $\mathcal{A}$ wants to flip a coin, it just pulls the next bit off that tape. In that case, for any *given* string s on that tape, the resulting algorithm $\mathcal{A}_s$ is deterministic, and we can think of $\mathcal{A}$ as just the uniform distribution over all those deterministic algorithms $\mathcal{A}_s$.

This means that the expected number of comparisons made by randomized algorithm $\mathcal{A}$ on some input I is just

$$\sum_s \Pr(s)(\text{Running time of } \mathcal{A}_s \text{ on } I).$$

If you recall the definition of expectation, the running time of the randomized algorithm is a random variable and the sequences s correspond to the elementary events.

So, the expected running time of the randomized algorithm is just an average over deterministic algorithms. Since each deterministic algorithm has average-case running time at least $\lfloor \log_2(n!) \rfloor$, any average over them must too. Formally, the average-case running time of the randomized algorithm is

$$\begin{aligned}
\operatorname*{avg}_{\text{inputs } I} \sum_s [\Pr(s)(\text{Running time of } \mathcal{A}_s \text{ on } I)] &= \sum_s \operatorname*{avg}_I [\Pr(s)(\text{Running time of } \mathcal{A}_s \text{ on } I)] \\
&= \sum_s \Pr(s) \operatorname*{avg}_I(\text{Running time of } \mathcal{A}_s \text{ on } I) \\
&\geq \sum_s \Pr(s) \lfloor \log_2(n!) \rfloor \\
&= \lfloor \log_2(n!) \rfloor. \quad \blacksquare
\end{aligned}$$

One way to think of the kinds of bounds we have been proving is to think of a matrix with one row for every possible deterministic comparison-based sorting algorithm (there could be a lot of rows!) and one column for every possible permutation of the n inputs (there are a lot of columns too). Entry (i, j) in this matrix contains the running time of algorithm i on input j. The worst-case deterministic lower bound tells us that for each row i there exists a column j_i such that the entry (i, j_i) is large. The average-case deterministic lower bound tells us that for each row i, the average of the elements in the row is large. The randomized lower bound says "well, since the above statement holds for every row, it must also hold for any weighted average of the rows." In the language of game-theory, one could think of this as a two-player game (much like rock-paper-scissors) between an "algorithm player" who gets to pick a row and an adversarial "input player" who gets to pick a column. Each player makes their choice and the entry in the matrix is the cost to the algorithm-player which we can think of as how much money the algorithm-player has to pay the input player. We have shown that there is a randomized strategy for the input player (namely, pick a column at random) that guarantees it an expected gain of $\Omega(n \log n)$ no matter what strategy the algorithm-player chooses.