Lecture 5

Concrete models and tight bounds I

5.1 Overview

In this lecture and the next, we discuss the notion of *lower bounds*: proving that any algorithm for some problem must take *at least* a certain amount of time to solve it. We will examine some simple concrete models of computation, each with a precise definition of what counts as a step, and will attempt to get tight upper and lower bounds for a number of problems. Unlike many of the other lectures, we will try to avoid using O, Θ, and Ω, and instead will examine exact quantities as much as possible.

In this lecture we focus on the *comparison model*. We will show that any deterministic comparison-based sorting algorithm must use at least $log_2(n!) \in \Omega(n \log n)$ comparisons to sort an array of n elements in the worst case. We also consider the problem of finding the largest and second-largest element in an array, which has an interesting connection to tennis tournaments and Lewis Carroll.

5.2 The idea of lower bounds

So far we have been focusing on designing good algorithms for various problems like sorting and median-finding. A natural question that arises this context is: are these algorithms best possible? If not, how much better could one hope to get? Addressing these questions requires proving that any algorithm must take *at least* a certain amount of time to solve the problem at hand. Statements of this form are called *lower bounds* because they give a lower bound to the question: "how fast an algorithm can one hope to get for the given problem?" In this context, an algorithm with a performance guarantee would be considered an *upper bound*. 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.

Lower bounds are often difficult to show: you cannot necessarily assume, for instance, that the sorting algorithm is going to choose a pivot as in Quicksort, or that it will split the array into pieces as in Mergesort; you need to consider any possible algorithm. To make this task a bit more tractable (as well as more precise) we will look at *concrete computational models*, where we explicitly specify what operations are allowed, and how much they each cost. Typically, each model will have some operations that cost 1 step (like performing a comparison, or swapping a pair of

elements), some that are free, and some that are not allowed at all. Then, within these models, we will see how close we can bring the upper and lower bounds together. In these next two lectures, we will consider doing this for several models and problems.

5.2.1 Terminology and setup

We will be focusing in this lecture on deterministic algorithms only (in the next lecture we will look at lower bounds for randomized algorithms). By a *worst-case upper bound* of $f(n)$ for some problem, we mean that there exists an algorithm that takes at most $f(n)$ steps on any input of size n. By a *worst-case lower bound* of $g(n)$, we mean that for any algorithm there exists an input on which it takes at least $g(n)$ steps. As mentioned above, the reason for this terminology is that if we think of our goal as being to understand the "true complexity" of each problem, measured in terms of the best possible worst-case guarantee achievable by any algorithm, then an upper bound of $f(n)$ and lower bound of $g(n)$ means that the true complexity is somewhere between $g(n)$ and $f(n)$.

5.3 Sorting by comparisons

We begin by considering 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 definitions.

Definition 5.1 *In the* **comparison model**, *an input consists of an array $[a_1, a_2, \ldots, a_n]$ of n items. Items can be examined only by comparing pairs of them. Each comparison ("is $a_i > a_j$?") returns YES or NO and counts a 1 time-step. All other computation, such as reordering items based on comparisons made, or incrementing counters, is free.*

Definition 5.2 *A* comparison-based sorting algorithm *operates in the comparison model. It takes as input an array $[a_1, a_2, \ldots, a_n]$ and 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 $\lg(n!) \in \Omega(n \log n)$ comparisons to sort n elements in the worst case.*[1] *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 $\lg(n!)$ comparisons to sort I.*

(Note: we will examine the quantity $\lg(n!)$ more carefully after proving the theorem.)

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*

[1] We use "lg" to mean "$\log_2$".

(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, if 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, the algorithm will need to make at least $\log_2(n!)$ comparisons before it can halt. ■

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 perform the above proof by considering the possible *outputs* of the sorting algorithm. From this perspective, the two key facts we need are: (a) there are $n!$ possible outputs, and (b) for any permutation, there exists an input for which it is the only correct answer. Now we can consider the same 20-questions game where S is the set of *outputs* consistent with all comparisons made so far. Again, 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 forces the algorithm to make at least $\lg(n!)$ comparisons.

Finally, let's take a look at the quantity $\lg(n!)$. We can expand this as: $\lg(n) + \lg(n-1) + \lg(n-2) + \ldots + 1$. The first $n/2$ terms are each at least $\lg(n/2)$, so this quantity is $\Omega(n \log n)$.

Question: Suppose we consider the problem: "order the input array so that the smallest $n/2$ come before the largest $n/2$"? Does our lower bound still hold for that problem, or where does it break down? How fast can you solve that problem?

Answer: No, the proof does not still hold. It breaks down because two different input orderings can have the same correct answer. E.g., the identity permutation is a correct answer for both the inputs `[1 2 3 4]` and `[2 1 4 3]`. In fact, not only does the lower bound break down, but we can actually solve this problem in linear time: just run the linear-time median-finding algorithm and then make a second pass putting elements into the first half or second half based on how they compare to the median.

5.3.1 How tight are our upper and lower bounds?

Let's consider how close our upper and lower bounds are for comparison-based sorting. First, to get a better handle on what exactly $\lg(n!)$ looks like, since today's theme is tight bounds, we can use the fact that $n! \in [(n/e)^n, n^n]$. So this means that:

$$\begin{array}{rcccl} n\lg n - n\lg e & < & \lg(n!) & < & n\lg n \\ n\lg n - 1.443n & < & \lg(n!) & < & n\lg n. \end{array}$$

In particular, no algorithm can sort with fewer than $n\lg n - 1.433n$ comparisons.[2]

Let's now consider our upper bounds. Assume n is a power of 2 for simplicity. Can you think of an algorithm that makes at most $n\lg n$ comparisons, and so is tight in the leading term? In fact, there are several algorithms, including:

Binary insertion sort: If we perform insertion-sort, using binary search to insert each new element, then the number of comparisons made is at most $\sum_{k=2}^{n}\lceil\lg k\rceil \leq n\lg n$. Note that insertion-sort spends a lot in moving items in the array to make room for each new element, and so is not especially efficient if we count movement cost as well, but it does well in terms of comparisons.

Mergesort: Merging two lists of $n/2$ elements each requires at most $n-1$ comparisons. So, unrolling the recurrence we get $(n-1) + 2(n/2-1) + 4(n/4-1) + \ldots + n/2(2-1) = n\lg n - (n-1) < n\lg n$.

5.3.2 Finding the maximum of n elements

How many comparisons are necessary and sufficient to find the maximum of n elements, in the comparison model of computation?

Claim 5.2 (Upper bound) *$n-1$ comparisons are sufficient to find the maximum of n elements.*

Proof: Just scan left to right, keeping track of the largest element so far. This makes at most $n-1$ comparisons. ■

[2]Since $1.433n$ is a low-order term, sometimes people will write this these bounds on $\lg(n!)$ as: $\lg(n!) = (n\lg n)(1 - o(1))$, meaning that the ratio between $\lg(n!)$ and $n\lg n$ goes to 1 as n goes to infinity.

Now, let's try for a lower bound. One simple lower bound is that since there are n possible answers for the location of the minimum element, our previous argument gives a lower bound of $\lg n$. But clearly this is not at all tight. In fact, we can give a better lower bound of $n-1$.

Claim 5.3 (Lower bound) *$n-1$ comparisons are needed in the worst-case to find the maximum of n elements.*

Proof: Suppose some algorithm $\mathcal{A}$ claims to find the maximum of n elements using less than $n-1$ comparisons. Consider an arbitrary input of n distinct elements, and construct a graph in which we join two elements by an edge if they are compared by $\mathcal{A}$. If fewer than $n-1$ comparisons are made, then this graph must have at least two components. Suppose now that algorithm $\mathcal{A}$ outputs some element u as the maximum, where u is in some component C_1. In that case, pick a different component C_2 and add a large positive number (e.g., the value of u) to every element in C_2. This process does not change the result of any comparison made by $\mathcal{A}$, so on this new set of elements, algorithm $\mathcal{A}$ would still output u. Yet this now ensures that u is not the maximum, so $\mathcal{A}$ must be incorrect. ■

Since the upper and lower bounds are equal, these bounds are tight.

5.3.3 Finding the second-largest of n elements

How many comparisons are necessary (lower bound) and sufficient (upper bound) to find the second largest of n elements? Again, let us assume that all elements are distinct.

Claim 5.4 (Lower bound) *$n-1$ comparisons are needed in the worst-case to find the second-largest of n elements.*

Proof: The same argument used in the lower bound for finding the maximum still holds. ■

Let us now work on finding an upper bound. Here is a simple one to start with.

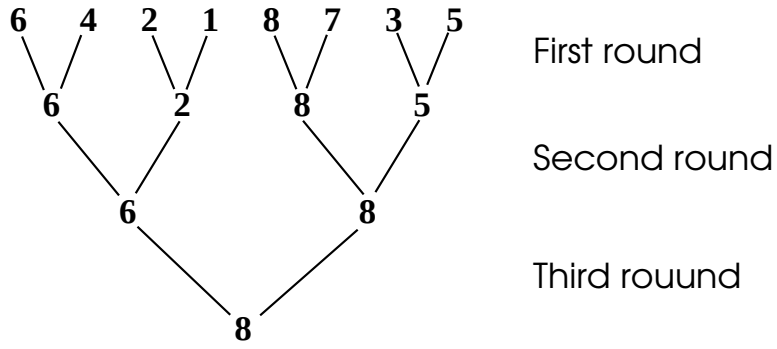
Claim 5.5 (Upper bound #1) *$2n-3$ comparisons are sufficient to find the second-largest of n elements.*

Proof: Just find the largest using $n-1$ comparisons, and then the largest of the remainder using $n-2$ comparisons, for a total of $2n-3$ comparisons. ■

We now have a gap: $n-1$ versus $2n-3$. It is not a huge gap: both are $\Theta(n)$, but remember today's theme is tight bounds. So, which do you think is closer to the truth? It turns out, we can reduce the upper bound quite a bit:

Claim 5.6 (Upper bound #2) *$n+\lg n-2$ comparisons are sufficient to find the second-largest of n elements.*

Proof: As a first step, let's find the maximum element using $n-1$ comparisons, but in a tennis-tournament or playoff structure. That is, we group elements into pairs, finding the maximum in each pair, and recurse on the maxima. E.g.,



Now, given just what we know from comparisons so far, what can we say about possible locations for the second-highest number (i.e., the second-best player)? The answer is that the second-best must have been directly compared to the best, and lost.[3] This means there are only $\lg n$ possibilities for the second-highest number, and we can find the maximum of them making only $\lg(n)-1$ more comparisons. ■

At this point, we have a lower bound of $n-1$ and an upper bound of $n+\lg(n)-2$, so they are nearly tight. It turns out that, in fact, the lower bound can be improved to exactly meet the upper bound.[4]

[3] Apparently first person to have pointed this out was Charles Dodgson (better known as Lewis Carroll!), writing about the proper way to award prizes in lawn tennis tournaments.

[4] First shown by S.S. Kislitsyn "On the selection of the kth element of an ordered set by pairwise comparison," 1964 (in Russian).