Lecture 3

Concrete models and tight upper/lower bounds

3.1 Overview

In this lecture, we will examine some simple, concrete models of computation, each with a precise definition of what counts as a step, and try to get tight upper and lower bounds for a number of problems. Specific models and problems examined in this lecture include:

- The number of comparisons needed to sort an array.
- The number of exchanges needed to sort an array.
- The number of comparisons needed to find the largest and second-largest elements in an array.
- The number of probes into a graph needed to determine if the graph is connected (the evasiveness of connectivity).

3.2 Terminology and setup

In this lecture, we will look at (worst-case) upper and lower bounds for a number of problems in several different concrete models. Each model will specify exactly what operations may be performed on the input, and how much they 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.

By an *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 *lower bound* of $g(n)$, we mean that for any algorithm there exists an input on which it takes at least $g(n)$ steps. 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 $f(n)$.

3.3 Sorting in the comparison model

One natural model for examining problems like sorting is what is known as the comparison model.

Definition 3.1 *In the* comparison model*, we have an input consisting of n items (typically in some initial order). An algorithm may compare two items (asking is $a_i > a_j$?) at a cost of 1. Moving the items around is free. No other operations on the items are allowed (such as using them as indices, XORing them, etc).*

For the problem of *sorting* in the comparison model, the input is an array $a = [a_1, a_2, \ldots, a_n]$ and the output is a permutation of the input $\pi(a) = [a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}]$ in which the elements are in increasing order. We begin this lecture by showing the following lower bound for comparison-based sorting.

Theorem 3.1 *Any deterministic comparison-based sorting algorithm must perform at least* $\lg(n!)$ *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* $\lg(n!) = \Omega(n \log n)$ *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. We now present the proof, which uses a very nice information-theoretic argument. (This proof is deceptively short: it's worth thinking through each line and each assertion.)

Proof: First of all, for a deterministic algorithm, the permutation it outputs (e.g., $[a_3, a_1, a_4, a_2]$) is solely a function of the sequence of answers it receives to its comparisons. In particular, any two different input arrays that yield the same sequence of answers will cause the same permutation to be produced as output. So, if an algorithm always made at most k comparisons, then there would be at most 2^k different permutations of the input array that it can possibly output, since each comparison has a YES or NO answer. This implies that if $k < \lg(n!)$, so $2^k < n!$, then there would be some permutation π of the input array that it *can't* output. Let's assume for contradiction that such a permutation exists. All that remains is to show that for any such permutation π, there is some input array for which π is the *only correct answer*. This is easy. For example, the permutation $[a_3, a_1, a_4, a_2]$ is the only correct answer for sorting the input $[2, 4, 1, 3]$, and more generally, permutation π is the only correct answer for the input $[\pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n)]$. Thus we have our desired contradiction. ■

The above is often called an "information theoretic" argument because we are in essence saying that we need at least $\lg_2(n!)$ bits of information about the input before we can correctly decide what output we need to produce. More generally, if we have some problem with M different outputs the algorithm might be required to produce, then we have a worst-case lower bound of $\lg M$.

What does $\lg(n!)$ look like? We have: $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + ... + \lg(1) < n \lg(n) = O(n \log n)$ and $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + ... + \lg(1) > (n/2) \lg(n/2) = \Omega(n \log n)$. So, $\lg(n!) = \Theta(n \log n)$.

[1] As is common in CS, we will use "lg" to mean "$\log_2$".

However, since today's theme is tight bounds, let's be a little more precise. We can in particular use the fact that $n! \in [(n/e)^n, n^n]$ to get:

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

Since $1.433n$ is a low-order term, sometimes people will write this fact this 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.

Assume n is a power of 2 — in fact, let's assume this for the entire rest of today's lecture. 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 $(n-1) + 2(n/2-1) + 4(n/4-1) + \ldots + n/2(2-1) = n\lg n - (n-1) < n\lg n$.

3.4 Sorting in the exchange model

Consider a shelf containing n unordered books to be arranged alphabetically. In each step, we can swap any pair of books we like. How many swaps do we need to sort all the books? Formally, we are considering the problem of *sorting* in the *exchange model*.

Definition 3.2 *In the* **exchange model**, *an input consists of an array of n items, and the only operation allowed on the items is to swap a pair of them at a cost of 1 step. All other (planning) work is free: in particular, the items can be examined and compared to each other at no cost.*

Question: how many exchanges are necessary (lower bound) and sufficient (upper bound) in the exchange model to sort an array of n items in the worst case?

Claim 3.2 (Upper bound) *$n-1$ exchanges is sufficient.*

Proof: For this we just need to give an algorithm. For instance, consider the algorithm that in step 1 puts the smallest item in location 1, swapping it with whatever was originally there. Then in step 2 it swaps the second-smallest item with whatever is currently in location 2, and so on (if in step k, the kth-smallest item is already in the correct position then we just do a no-op). No step ever undoes any of the previous work, so after $n-1$ steps, the first $n-1$ items are in the correct position. This means the nth item must be in the correct position too. ■

But are $n-1$ exchanges necessary in the worst-case? If n is even, and no book is in its correct location, then $n/2$ exchanges are clearly necessary to "touch" all books. But can we show a better lower bound than that?

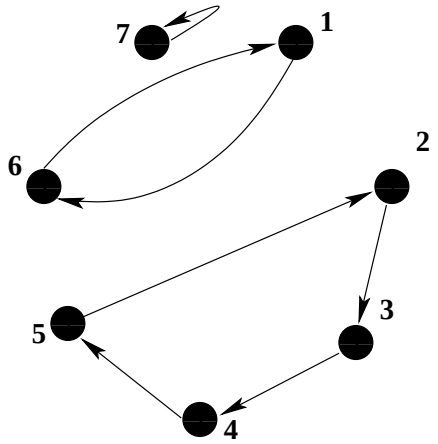


Figure 3.1: Graph for input `[f c d e b a g]`

Claim 3.3 (Lower bound) *In fact, $n-1$ exchanges are necessary, in the worst case.*

Proof: Here is how we can see it. Create a graph in which a directed edge (i, j) means that that the book in location i must end up at location j. For instance, consider the example in Figure 3.1. Note that this is a special kind of directed graph: it is a permutation — a set of cycles. In particular, every book points to *some* location, perhaps its own location, and every location is pointed to by exactly one book. Now consider the following points:

1. What is the effect of exchanging any two elements (books) that are in the same cycle?

   Answer: Suppose the graph had edges (i_1, j_1) and (i_2, j_2) and we swap the elements in locations i_1 and i_2. Then this causes those two edges to be replaced by edges (i_2, j_1) and (i_1, j_2) because now it is the element in location i_2 that needs to go to j_1 and the element in i_1 that needs to go to j_2. This means that if i_1 and i_2 were in the same cycle, that cycle now becomes two disjoint cycles.

2. What is the effect of exchanging any two elements that are in different cycles?

   Answer: If we swap elements i_1 and i_2 that are in different cycles, then the same argument as above shows that this merges those two cycles into one cycle.

3. How many cycles are in the final sorted array?

   Answer: The final sorted array has n cycles.

Putting the above 3 points together, suppose we begin with an array consisting of a single cycle, such as $[n, 1, 2, 3, 4, \ldots, n-1]$. Each operation at best increases the number of cycles by 1 and in the end we need to have n cycles. So, this input requires $n-1$ operations. ■

3.5 The comparison model revisited

3.5.1 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 3.4 (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. ■

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 maximum 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 3.5 (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.

3.5.2 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 3.6 (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.

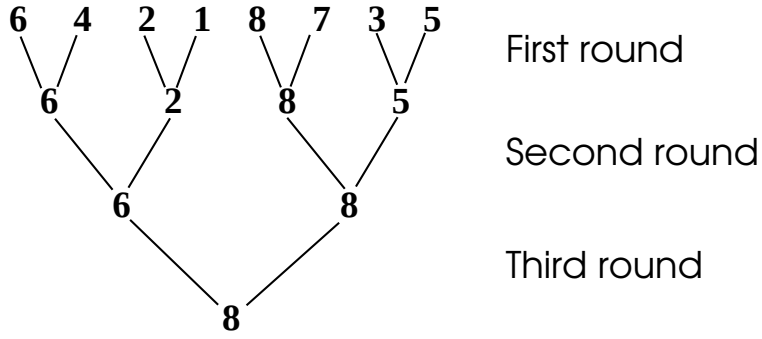
Claim 3.7 (Upper bound #1) *$2n-3$ comparisons are sufficient to find the second-largest of n elements.*

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 3.8 (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.[2] 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.[3]

3.6 Query models, and the evasiveness of connectivity

To finish with something totally different, let's look at the query complexity of determining if a graph is connected. Assume we are given the adjacency matrix G for some n-node graph. That is, $G[i,j]=1$ if there is an edge between i and j, and $G[i,j]=0$ otherwise. We consider a model in which we can *query* any element of the matrix G in 1 step. All other computation is free. That is, imagine the graph matrix has values written on little slips of paper, face down. In one step we can turn over any slip of paper. How many slips of paper do we need to turn over to tell if G is connected?

[2] Apparently the 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.

[3] First shown by Kislitsyn (1964).

Claim 3.9 (Easy upper bound) *$n(n-1)/2$ queries are sufficient to determine if G is connected.*

Proof: This just corresponds to querying every pair (i, j). Once we have done that, we know the entire graph and can just compute for free to see if it is connected. ■

Interestingly, it turns out the simple upper-bound of querying every edge is a lower bound too. Because of this, connectivity is called an "evasive" property of graphs.

Theorem 3.10 (Lower bound) *$n(n-1)/2$ queries are necessary to determine connectivity in the worst case.*

Proof: Here is the strategy for the adversary: when the algorithm asks us to flip over a slip of paper, we return the answer 0 *unless* that would force the graph to be disconnected, in which case we answer 1. (It is not important to the argument, but we can figure this out by imagining that all un-turned slips of paper are 1 and seeing if that graph is connected.) Now, here is the key claim:

> *Claim:* we maintain the invariant that for any un-asked pair (u, v), the graph revealed so far has no path from u to v.
>
> *Proof of claim:* If there was, consider the last edge (u', v') revealed on that path. We could have answered 0 for that and kept the same connectivity in the graph by having an edge (u, v). So, that contradicts the definition of our adversary strategy.

Now, to finish the proof: Suppose an algorithm halts without examining every pair. Consider some unasked pair (u, v). If the algorithm says "connected," we reveal all-zeros for the remaining unasked edges and then there is no path from u to v (by the key claim) so the algorithm is wrong. If the algorithm says "disconnected," we reveal all-ones for the remaining edges, and the algorithm is wrong by definition of our adversary strategy. So, the algorithm must ask for all edges. ■