Lecture 7

Concrete models and tight bounds II

7.1 Overview

In this lecture we continue our study of upper and lower bounds for simple concrete models of computation. In each model we give 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:

- Sorting in the exchange model.
- The number of probes into a graph needed to determine if the graph is connected (the evasiveness of connectivity).
- Lower bounds for randomized algorithms, using the 2-player game view of algorithm design and analysis.

7.2 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 7.1 *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 7.1 (Upper bound) $n - 1$ *exchanges is sufficient.*

Proof: To prove an upper bound of $n-1$ 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

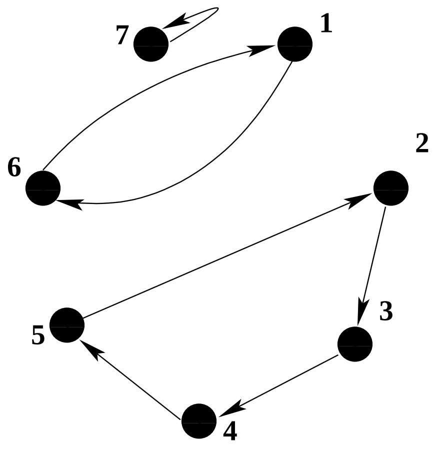

Figure 7.1: Graph in lower bound proof for input `[6 3 4 5 2 1 7]`

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?

Claim 7.2 (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 7.1. 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. ■

7.3 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?

Claim 7.3 (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 7.4 (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 algorthm 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. ■

We'll see more arguments like this when we talk about spanning trees later on in the course.

7.4 Lower bounds for randomized algorithms

The lower bound proofs we have given so far have assumed the algorithm is *deterministic*. In particular, we used arguments in which the adversary did not commit to the actual instance until the end: e.g., in our lower bound for connectivity, the graph wasn't fully determined until after the algorithm finished; in our lower bound for sorting, we played "20 questions" without having a specific ordering of the input in mind until the end. This is fine for deterministic algorithms because the algorithm would behave in exactly the same way if we were to re-run it on the input that we produced. However, this argument is not legitimate for *randomized* algorithms. For randomized algorithms, an input that caused one sequence of random choices to behave badly might not necessarily cause the algorithm to behave badly in expectation. Instead, the way we will prove lower bounds for randomized algorithms is by giving a fixed randomized strategy for the *adversary*, such that no deterministic algorithm can do well in expectation. This will then imply that no randomized algorithm can do well in expectation either.

Let us see how this plays out with comparison-based sorting. In particular, the randomized strategy we will examine for the adversary is just to present the n items in a random order. We then prove a stronger version of Theorem 5.1 from last time, showing:

Theorem 7.5 *For any deterministic comparison-based sorting algorithm, the* average-case number of comparisons *(the number of comparisons needed on average to sort 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 argument from last time, each leaf of this tree must correspond 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 one is a bit more clever in the proof, one can get rid of the floor in the bound.

Theorem 7.6 *The above bound holds for randomized algorithms too.*

[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....

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.