In today’s class, after formally defining what “problems” and “algorithms” are, we talked about how to calculate lower bounds on the running time of all possible algorithms for one problem. More specifically, we analyzed the lower bounds running time of all comparison-based sorting algorithms.

[Announcements]

1. The graded midterm papers were handed back at today’s class. The minimum score was 6, the maximum was 31 and the average was 16.7 (with a standard deviation of 5.7).

2. Please have a look at the grading scale posted on the class website (also attached below). Since the final letter grades will not be based on curve, your final grades will not depend on whether other students in the same class are doing well or not. So a wise strategy is just to try your best to improve your total score.
<table>
<thead>
<tr>
<th>%</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>A+</td>
</tr>
<tr>
<td>70</td>
<td>A</td>
</tr>
<tr>
<td>63</td>
<td>A-</td>
</tr>
<tr>
<td>57</td>
<td>B+</td>
</tr>
<tr>
<td>51</td>
<td>B</td>
</tr>
<tr>
<td>46</td>
<td>B-</td>
</tr>
<tr>
<td>42</td>
<td>C+</td>
</tr>
<tr>
<td>38</td>
<td>C</td>
</tr>
<tr>
<td>34</td>
<td>C-</td>
</tr>
<tr>
<td>30</td>
<td>D</td>
</tr>
</tbody>
</table>
[During the Lecture]

1. Based on some common problems observed during midterm grading, two concepts are clarified:

(1) The general form of recurrence relation is: \( f(n) = g(f(n-1), f(n-2) \ldots) \). It can be viewed as the equation of relating the value of a function at input \( n \) to values of the same function for smaller values \( n' < n \). Two specific forms we used in this class were provided as examples. When giving a recurrence, one usually also needs to specify the base case.

(2) When big-O notation is asked, you are not just supposed to give the upper bounds form: \( f = O(g) \), but to give the asymptotic notation in detail. The term “big-O notation” refers to all of \( O \), Omega, and Theta.
2. Afterwards, the class moved to a new topic, investigating lower bounds for every algorithm for a problem.

So far in the algorithm course, what we did can be described as:

(1) pick up a problem;

(2) design an algorithm for it;

(3) prove the correctness and analyze the running time;

To analyze the running time of one particular algorithm, we calculate the worst case running time, which gives us the lower bounds.

Now the new challenge we are facing is to prove no algorithm (not just for all our, or all known algorithms, but for all possible algorithms) can beat a certain bound in the worst case for one problem. Take sorting problem as an example. From our past study we all know that Merge Sort can fulfill the task with a worst case running time $\Theta(n \log(n))$. What we’d like to prove is that no algorithm can be faster than that in the worst case. Notice the difference between giving a lower bound for one particular algorithm, and a lower bound for all algorithms.
3. In order to approach the problem, the first, maybe also the most important step is define what an algorithm is, or say, what an algorithm is/isn’t allowed to do.

So far in this course, we took the “know it when we see it” approach. All the algorithms are given explicitly, so we could agree that it is “fair game”. Now we are trying to reason about “all algorithms”. In order to do that, we need to say what “all algorithms” are.
Generally, we consider a problem as a “function”, $f: I \rightarrow \text{Powerset}(O)$ where $I$ is some set of inputs and $O$ is some set of outputs. For each input $i \in I$, there is a subset $f(i) \in O$, which is legal.

Solving a problem can be viewed as finding an algorithm $A: I \rightarrow O$ with $A(i) \in f(i)$ for all $i$. So what we are saying is really that an algorithm is nothing but a function, and the function has to compute a legal output.

Notice: since here we are talking about functions, there is no user interface or input/output during the execution time of the function calculation. One thus assumes that we are writing our algorithms using a
programming language that only allows data exchange at the beginning and end of the program.

5. Given the above assumption, we allow algorithms to contain:

(1) Loops (while, for);

(2) Test and branches (if);

(3) Variables, arrays, infinite memory;

(4) Basic arithmetic operations.

A programming language using only the above categories of commands is in fact as powerful as C++ and JAVA, but maybe not very comfortable to work with. In other words, most of the other commands in “real” programming languages add a lot of comfort in programming, but no
power that we didn’t have before.

Now we have properly defined problems and algorithms, the next statement we should know is that in reality, very few good lower bounds are known. They tend to be very hard to prove. Most of the time, the only lower bound we can prove is that a correct algorithm needs at least linear time, because to make sure to get the answer right, it has to look at the entire input. (Sometimes, one cannot even prove that.)

6. The specific problem we studied in today’s lecture is comparison based sorting. In a comparison-based algorithm, there is no access to individual bits, or even the numbers, of the input data. Instead, the only way the algorithm can query the array is by giving two indices i, j, and learning if a[i] < a[j], a[i] = a[j], or a[i] > a[j]. The algorithm can then branch, and make other decisions, based on the outcome of that test, but
it cannot learn anything else about the array entries. While this is somewhat restrictive, all the sorting algorithms we have seen so far (MergeSort, QuickSort, Insertion Sort, HeapSort, ...) are comparison-based.

One of the few success instances of obtaining lower bounds is a lower bound of $\Omega(n \log n)$ for comparison based sorting.

For comparison based sorting, each time we make a comparison of two array entries, we can change the content of the array (by swap
operations) right after each comparison, or equally, we can first save all
the content-changing operations and then perform all of them after all the
comparisons are done. As an example of how the computation of
Insertion Sort would be modified to satisfy this assumption, we saw the
computation tree of Insertion Sort for an array of three entries.

The idea can be generalized as the following: for all comparison based
algorithms, we can assume that all the comparisons can be done before
all the swaps.

For any computation tree, the sequence of swaps for every leaf node is
different. That is because every leaf corresponds to different “case”.
Formally speaking, each leaf uniquely corresponds to one distinct input.
Suppose the input is an array of size \( n \), then there are all together \( n! \)
possible inputs. Given that the algorithm is correct, each of those inputs
will lead to a distinct leaf node. Consequently, there are (at least) n! leaves.

9. Next, we want to relate the worst-case computation time of the algorithm to properties of the computation tree. It is lower-bounded by the length of longest path in the computation tree from the root node to any leaf node, which is exactly the height of the tree (represented by h). This is assuming that each comparison takes one step, and all other computation is completely free. Clearly, this is a lower bound. Notice that of course, different sorting algorithms will have different computation trees with different longest paths. But for each sorting algorithm, there is some tree, and it has some longest path, of length h.

About the computation tree, we know:

(1) It is a binary tree;

(2) It has at least n! nodes.
A binary tree with height $h$ has at most $2^h$ leaves. In addition to that, we know that $n! > (n/e)^n$, from Stirling's formula for factorials. After a little bit of mathematics derivation ($2^h \geq n! > (n/e)^n \Rightarrow h \geq \log(n/e)^n = n(\log(n)) - n(\log(e)) = \Omega(n(\log(n)))$, we obtained the correct general lower bounds of comparison based algorithm for a size $n$ array. In other words, at least $\Omega(n(\log(n)))$ comparisons (yes/no branching) are needed to be able to deal with all the $n!$ possible input arrays.

10. All the above calculations and analysis are only correct for “comparison based” algorithms, and other situations may be quite different. In fact, if all array elements are small integers between 1 and some $k$, then sorting can be done much faster, pretty much in linear time.