1 Introduction: Lower Bounds for Problems, not only for Algorithms

We’ve seen many arguments that provide upper bounds on the running times (or space usage) of algorithms. Occasionally, we’ve seen lower bounds for worst-case performance. For example, Quicksort takes \( \Omega(n^2) \) time on an already sorted list because of poor choice of pivot. But how do you know whether an algorithm is optimal: that is, how do you know whether or not there might be some other algorithm that performs asymptotically better? Note: Asymptotically better means: If algorithm A takes \( f(n) \) time and algorithm B takes \( g(n) \) time, and \( \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0 \) then algorithm A is asymptotically faster than algorithm B.

To answer such a question, it would seem that we need to be able to say something about the inherent difficulty of the problem \( X \) that the algorithm solves. Something of the form: “Any algorithm that solves problem \( X \) takes (in the worst case—or perhaps on average) at least \( \Omega(f(n)) \) time”. That sounds hard. [Spoiler alert: It is.]

The good news is that to some extent this can be done and the methods used—as well as the results themselves—are worth learning something about. That’s the goal for this week.

2 How Can We Reason About All Algorithms That Solve \( X \)!!

Ok, we probably can’t. But we can reason about families of algorithms that solve \( X \). Said differently, we can specify a model of computation and then argue about how long any algorithm consistent with that model might take to solve problem \( X \). We’ll look at the following models

**Comparison-based:** The only operations on data are comparisons.

**Decision Tree:** We model the algorithm base on its conditional statements. Of particular note are

- Linear Decision Tree
- Algebraic Decision Tree

**Turing Machines** and other equivalent models

Among the argument used are

- Adversary (oracle) arguments
- Topological arguments
- Problem reduction arguments

3 Comparison-Based Sorting

Most of the sorting algorithms you’ve seen are comparison-based; that is, the only questions asked of the items being sorted are pair-wise comparisons. Quicksort, Mergesort, Heapsort, Selectionsort, Insertionsort are all examples.

We assume that the algorithm takes as input an array \( A[1..n] \). A comparison has the form “\( x < y \)”. The output produced is a permutation (reordering) of the elements of \( A \). We can imagine associating that permutation with a permutation \( P[1..n] \) of the indices of \( A \), where \( P[i] \) is the location in \( A[] \) of the \( i \)th smallest element. That is, \( A[P[1]], A[P[2]], \ldots, A[P[n]] \) are the elements of \( A \) in increasing order. [We don’t require the algorithm to produce \( P \); \( P \) will merely allow us to analyze the algorithm’s performance.]

Note that there are \( n! \) possible permutations that could be produced, and that for each possible permutation, there is some input that would produce exactly that permutation and no other. Let \( P \) be the set of all permutations of \( \{1, \ldots, n\} \).

Each time a comparison is performed by the algorithm, some set of permutations are ruled out as possible outputs of the algorithm. Let \( F \) be the set of permutations that have, at any given point during the execution of the algorithm, not yet been ruled out. \( F \) initially equals \( P \); after a comparison, \( F \) may lose elements.
If all of the elements of \( A \) are distinct, then the algorithm cannot be finished executing if \( \mathcal{F} \) contains more than one permutation.

Here’s where we get sneaky: Any comparison partitions the set \( \mathcal{F} \) into two subsets \( \mathcal{F}_T \) and \( \mathcal{F}_F \), corresponding to whether the permutation corresponds to the comparison returning \text{true} or \text{false} respectively. Further, one of these sets must be at least half the size of \( \mathcal{F} \).

So imagine that you hold the array \( A \) so that all comparisons come to you to be answered. You merely respond \text{true/false} so that the larger of \( \mathcal{F}_T \) and \( \mathcal{F}_F \) is retained as \( \mathcal{F} \).

This means that in order for the algorithm to halt, at least \( \log_2(n!) \) comparisons must be made! But

\[
\log_2(n!) = \log_2 n + \log_2(n-1) + \ldots + \log_2 1 \geq (n/2) \log_2(n/2)
\]

and \((n/2) \log_2(n/2) \in \Omega(n \log n)\). Therefore any such algorithm requires at least this many computations in the worst case.

### 3.1 Finding the Maximum Element

The above proof is called an adversary argument, and it is a widely used technique for establishing lower bounds. With it, one can easily show, for example, that any algorithm to find the maximum element in an array must perform at least \( n - 1 \) comparisons. Here’s how:

Consider the elements of \( A \) to be the vertices of a graph. Initially the graph has no edges. Each time a comparison \( A[i] > A[j] \) is performed, add a directed edge from the smaller of \( A[i] \) and \( A[j] \) to the larger (if it does not already exist). Now suppose again that an adversary is answering the questions. As long as the graph is disconnected, the algorithm cannot know which component corresponds to the maximum value of \( A \). Since an \( n \)-vertex graph requires at least \( n - 1 \) edges, at least \( n - 1 \) comparisons must be made.

### 4 Decision Trees

#### 4.1 Comparison-Based

Another way of approaching lower bound questions is to consider the decision tree for an algorithm. The decision tree captures the conditional flow of control of an algorithm. The decision tree has a node for the execution of each conditional statement and the parent of a node was the most recent previous conditional executed. Each terminal conditional has a leaf for each possible outcome of the decision. We assume that all conditionals are \text{true/false}, as opposed to multi-way.

If, for inputs of size \( n \), an algorithm can produce \( N \) possible distinct outputs, then the depth of the decision tree must be at least \( \log_2 N \). For sorting, there are \( N = n! \) possible distinct outputs, and for each such output, there is an input that is consistent with only that output. Thus a decision tree for a comparison-based sorting algorithm must have depth at least \( \log_2(n!) \in \Omega(n \log n) \).

This approach can also be used to obtain lower bounds on the average run-time of algorithms. For example, we can show that any deterministic comparison-based sorting algorithm for which all inputs are equally likely must have \textit{average case} run time at least \( \Omega(n \log n) \).

To see this, consider the decision tree for such an algorithm. As noted above, it must have at least \( n! \) leaves. Note that if the tree were balanced—that is, if any two leaves had depths that differed by at most 1—then the height of the tree would be \( \Theta(\log_2(n!)) = \Theta(n \log n) \). Suppose then that the tree has two leaves whose depths differ by at least 2. By moving the deeper leaf and making it the child of the shallower leaf we do not increase the average leaf depth; repeatedly adjusting leaves will eventually balance the tree, and so its depth must also have been at least \( \Theta(n \log n) \).

#### 4.2 Linear Decision Trees

Suppose that the data we wish to sort is numeric (or comparable via associated values). Then we could imagine comparisons based on algebraic operations. A linear decision tree allows the comparison of linear combinations of
values, or put differently, of expressions of the form \( f(x_1, x_2, \ldots, x_k) > 0 \) for fixed \( k \), where \( f \) is a linear function. It can be shown that under this more powerful model, sorting still requires \( \Theta(n \log n) \) operations in the worst case.

### 4.3 Algebraic Decision Trees

We can enhance the model more by allowing comparisons of the form \( f(x_1, x_2, \ldots, x_k) > 0 \) where \( f \) can now be any polynomial expression of bounded degree; sorting still requires \( \Theta(n \log n) \) operations in the worst case.

### 5 Lower Bounds via Problem Reduction

#### 5.1 Convex Hull

The convex hull problem takes as input a finite set of points in the plane and returns the smallest polygon containing all of those points as an ordered (in say counterclockwise order) list of the vertices of the polygon. The output is always a subset of the input.

**Claim:** In the algebraic decision tree model, computing the convex hull requires at least \( \Omega(n \log n) \) operations in some cases.

We prove the claim by problem reduction: We will show that an array \( A \) of \( n \) distinct numbers can be converted into a set of points in the plane in \( O(n) \) time so that the convex hull of the corresponding points contains information that can be converted in \( O(n) \) time into a sorted list of the numbers. This yields a sorting algorithm that requires time at most \( O(n + h(n)) \), where \( h(n) \) is the time required to solve an \( n \)-point convex hull instance. Thus \( n + h(n) \geq cn \log n \) for some constant \( c > 0 \), and so \( h \) cannot be \( o(n \log n) \).

The transformation is simple: Each \( A[i] \) become the point \((A[i], A[i]^2)\). These all lie on the parabola \( y = x^2 \) and therefore all of the points will be on the convex hull and the \( x \)-coordinates, read off in order, produce the sorted array!

### 6 Other Problems

#### 6.1 Element Uniqueness

The element uniqueness problem takes as input an array of values, say integers, and asks whether any two of them are equal. Note that this problem can be solved in \( O(n \log n) \) time by reducing it to sorting. However, it doesn’t seem possible to show that it requires \( \Omega(n \log n) \) time by a reduction from sorting.

It turns out, though, that in the algebraic decision tree model of computation, element uniqueness algorithms do require at least \( \Omega(n \log n) \) time for some inputs, although the known proofs are not simple. We can get a sense of why this might be true by working within the simpler comparison-based model.

Consider any array of \( n \) numbers as a point in \( n \)-dimensional Euclidean space. Every point in that space is a potential input to the problem. For any two dimensions \( i \) and \( j \), the subspace of all points where \( x_i = x_j \) is an \( n - 1 \)-dimensional subspace (hyperplane). The \( \binom{n}{2} \) such hyperplanes divide the space into several regions, called cells, where each cell is bounded by some set of hyperplanes. Points in the same cell represent arrays that have the same permutation associated with their sorted order.

This means that there are at least \( n! \) such cells and any element uniqueness algorithm must determine which cell the point corresponding to the array lies in. To see this, imagine you are an adversary answering the questions asked by an element uniqueness algorithm. If you can guarantee that there are always at least two cells whose points are consistent with the answers you have given to the algorithm’s comparisons, then the algorithm can’t yet have determined element uniqueness. Since you can force the algorithm to ask at least \( \Omega(\log(n!)) = \Omega(n \log n) \) questions, the result follows.
7 Fun Problems

- Find a lower bound for number of compares to merge two sorted lists.
- Find a lower bound for number of compares needed to determine the median of a list of \(n\) integers.
  
  **Note 1.** *Aim for* \(n - 1\). *Challenge:* (\(3n - 3\))/2.

- Find a lower bound for number of compares to find both the max and min elements.
  
  **Note 2.** *Aim for* (\(3n - 4\))/2.

- Problem You are given 12 balls and a two-pan balance scale (it you’re too young to know what this is, Google it!). Of the 12 balls, 11 have same weight, one has a different weight. Figure out which ball has the different weight with as few weighings as possible and whether its weight is lower or higher than that of the other 11 balls.

- Read Erickson’s notes (Lecture 29) on adversary arguments and evasive graph properties

- Problem: Do Problem 2 from Erickson’s Lecture 29

- Problem: Do Problem 4 from Erickson’s Lecture 29

- Often one can prove a lower bound for a particular problem via problem reduction, as done above for the Convex Hull problem. Here’s another geometric example. The Closest Pair problem takes as input an array of \(n\) points in the plane and returns a pair of points of minimum distance. Show that in the algebraic decision tree model of computation any algorithm that solves this problem requires at least \(\Omega(n \log n)\) time in some cases.

  **Note 3.** *Element uniqueness.*