**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 $\lim \frac{g(n)}{f(n)} \to 0$ 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 within 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—if not all—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 \( \mathcal{F} \) be the set of permutations that have, at any given point during the execution of the algorithm, not yet been ruled out. \( \mathcal{F} \) initially equals \( P \); after a comparison, \( \mathcal{F} \) 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 a connected \( n \)-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 a **decision tree** for an algorithm. A decision tree captures the conditional flow of control of an algorithm. A 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. In some models, conditional nodes have two outcomes (true/false) while in others they are multi-way; we’ll see some examples with 3-way conditionals later on.

If, for inputs of size \( n \), an algorithm can produce \( N \) possible distinct outputs, then, assuming \( c \)-way conditionals, the depth of the decision tree must be at least \( \log_c N \). For sorting, there are \( N = n! \) possible distinct permutations that the input might produce, and for each such permutation, there must be a distinct leaf that represents outputting that permutation. 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 **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. Let \( v \) be a leaf of maximum depth \( d \) and parent \( u \) and let \( w \) be a leaf with depth at most \( d - 2 \). Note that if \( w \) has another child, that child also is a leaf (of depth \( d \)). Now by moving \( w \) to be the child of the shallower leaf \( u \), 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 numeric values). Then we could imagine conditional statements based on algebraic operations. A linear decision tree allows the comparison of linear combinations of values, or put differently, of expressions of the form \( L(x_1, x_2, \ldots, x_n) \) where \( L \) is a linear function. The branching then depends of the sign \( \{-1, 1, 0\} \) of the result; that is, it is 3-way branching. Allowing these more general types of conditional statements doesn’t really affect the asymptotic lower bound for sorting, beyond changing the base of the \( \log \) from 2 to 3. But this seems to depend heavily on the size of the the output space (the \( n! \) possible permutations).

However, suppose we consider a different set of problems: decision problems. Consider the fundamental problem of *element uniqueness*: Given a set of \( n \) values, determine whether there exist any duplicate values. If the only operation is equality testing, then clearly each pair must be tested for equality, giving an \( O(n^2) \) lower bound. Suppose, however, that the values are numeric. Then we could sort them in \( O(n \log n) \) time and then compare consecutive pairs for equality. Can we do better than \( O(n \log n) \) time in using the linear decision tree model?

The answer turns out to be "no", but a more sophisticated approach appears to be necessary, since there are only 2 possible outcomes (2 types of leaves): "true" leaves (all elements are unique), and "false" leaves (not all elements are unique). Thus, the size of the output space isn’t much help, but maybe we can take advantage of the size of the input space.

So, let’s think abstractly: We can think of the input for element uniqueness as a vector in \( \mathbb{R}^n \) whose coordinates are the array of values, and of the element uniqueness problem as the computation of the function \( f : \mathbb{R}^n \to \mathbb{R} \) where \( f(\overline{v}) = 1 \) if the elements are unique and \( f(\overline{v}) = 0 \) otherwise. Let \( F_1, F_0 \) denote the partition of \( \mathbb{R}^n \) into the vectors having all unique elements and those not having all unique elements, respectively.

For an arbitrary node \( t \) in the decision tree, let \( C_t \) denote the set of points in \( \mathbb{R}^n \) consistent with all of its ancestor nodes. For the root \( r \) of the tree, \( C_r = \mathbb{R}^n \). Each conditional statement corresponds to computing the sign of some linear function \( L(\overline{v}) \). The first such conditional divides \( C_r \) into 3 components \( C^+_r, C^0_r, C^-_r \) corresponding to the sign of \( L(\overline{v}) \).

The sets in \( C_t \) have some interesting properties

- Each \( C_t \) is a convex set: the line segment between any two points in \( C_t \) is contained entirely within \( C_t \).
- And so each \( C_t \) is connected: For any two points in \( C_t \) there is a path between them that is completely contained in \( C_t \).
- The sets \( C_t \) for each leaf \( t \) partition \( \mathbb{R}^n \).
- For each leaf \( t \), either \( C_t \subseteq F_1 \) or \( C_t \subseteq F_1 \); let \( N_t \) be the number of \( C_t \) for which \( t \) is a leaf and \( C_t \subseteq F_i \).
- Then the decision tree must have \( N_0 + N_1 \) leaves.

If \( C_t \) is associated with a leaf having value 1, then any two vectors \( \overline{v}, \overline{w} \in C_t \), each of which have distinct coordinates, must have the same order type: that is, for every \( 1 \leq i < j \leq n \) if \( v_i < w_j \), then \( v_i < w_j \).

If not, then some point \( \overline{w} \) along the segment \( \overline{v}, \overline{w} \) must have \( \overline{w}_i = \overline{w}_j \) (by the Intermediate Value Theorem), but then \( f(\overline{v}) = 1 = f(\overline{w}) \) while \( f(\overline{w}) = 0 \), violating the property that each component is convex.
Thus the number of leaves corresponding to “true” answers must be at least the number of order types, which is \( n! \).
But any tree with at least this many leaves has depth at least \( O(n \log n) \).
This same approach can be used to get lower bounds for other common decision problems.

### 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; element uniqueness still requires \( \Theta(n \log n) \) operations in the worst case, although this is a non-trivial result!

### 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!

#### 5.2 Closest Pair

Consider the problem of finding the distance between closest pair of points among \( n \) points in \( \mathbb{R}^d \). It turns out that this problem also requires \( \Theta(n \log n) \) time in the algebraic decision tree model. We can justify this claim by providing a linear-time reduction from Element Uniqueness to this problem. For each value \( x \), create the point \( (x, x, \ldots, x) \in \mathbb{R}^d \) then use an optimal closest pair algorithm to determine whether the closest pair distance is 0. If this could be done in asymptotically faster than \( O(n \log n) \) time, then the lower bound for element uniqueness would be violated.

### 6 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