It was a Game called Yes and No, where Scrooge's nephew had to think of something, and the rest must find out what; he only answering to their questions yes or no, as the case was. The brisk fire of questioning to which he was exposed, elicited from him that he was thinking of an animal, a live animal, rather a disagreeable animal, a savage animal, an animal that growled and grunted sometimes, and talked sometimes, and lived in London, and walked about the streets, and wasn't made a show of, and wasn't led by anybody, and didn't live in a menagerie, and was never killed in a market, and was not a horse, or an ass, or a cow, or a bull, or a tiger, or a dog, or a pig, or a cat, or a bear. At every fresh question that was put to him, this nephew burst into a fresh roar of laughter; and was so inexpressibly tickled, that he was obliged to get up off the sofa and stamp. At last the plump sister, falling into a similar state, cried out :

"I have found it out! I know what it is, Fred! I know what it is !"

"What is it?" cried Fred.

"It's your Uncle Scro-o-o-o-oge!"

Which it certainly was. Admiration was the universal sentiment, though some objected that the reply to "Is it a bear?" ought to have been "Yes;" inasmuch as an answer in the negative was sufficient to have diverted their thoughts from Mr Scrooge, supposing they had ever had any tendency that way.

— Charles Dickens, *A Christmas Carol* (1843)

19 Lower Bounds

19.1 Huh? Whuzzat?

So far in this class we've been developing algorithms and data structures to solve certain problems as quickly as possible. Starting with this lecture, we'll turn the tables, by proving that certain problems *cannot* be solved as quickly as we might like them to be.

Let $T_A(X)$ denote the running of algorithm A given input X. For most of the semester, we've been concerned with the the worst-case running time of A as a function of the input size:

$$T_A(n) := \max_{|X|=n} T_A(X).$$

The worst-case complexity of a *problem* Π is the worst-case running time of the *fastest* algorithm for solving it:

$$T_\Pi(n) := \min_{A \text{ solves } \Pi} T_A(n) = \min_{A \text{ solves } \Pi} \max_{|X|=n} T_A(X).$$

Any algorithm A that solves Π immediately implies an *upper bound* on the complexity of Π; the inequality $\boldsymbol{T_\Pi(n) \le T_A(n)}$ follows directly from the definition of T_Π. Just as obviously, faster algorithms give us better (smaller) upper bounds. In other words, whenever we give a running time for an algorithm, what we're really doing—and what most computer scientists devote their entire careers doing[1]—is bragging about how *easy* some problem is.

Now, instead of bragging about how easy problems are, we will argue that certain problems are *hard*, by proving *lower bounds* on their complexity. This is considerably harder than proving an upper bound, because it's no longer enough to examine a single algorithm. To prove an inequality of the form

[1]This sometimes leads to long sequences of results that sound like an obscure version of "Name that Tune":

Lennes: "I can triangulate that polygon in $O(n^2)$ time."
Shamos: "I can triangulate that polygon in $O(n \log n)$ time."
Tarjan: "I can triangulate that polygon in $O(n \log\log n)$ time."
Seidel: "I can triangulate that polygon in $O(n \log^* n)$ time." [Audience gasps.]
Chazelle: "I can triangulate that polygon in $O(n)$ time." [Audience gasps and applauds.]
"Triangulate that polygon!"

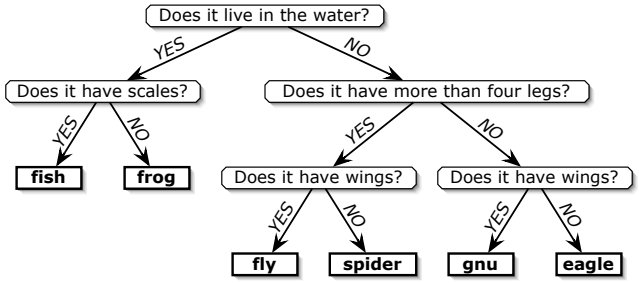
$T_\Pi(n) = \Omega(f(n))$, we must prove that ***every*** algorithm that solves Π has a worst-case running time $\Omega(f(n))$, or equivalently, that ***no*** algorithm runs in $o(f(n))$ time.

19.2 Decision Trees

Unfortunately, there is no formal definition of the phrase 'all algorithms'![2] So when we derive lower bounds, we first have to specify *precisely* what kinds of algorithms we will consider and *precisely* how to measure their running time. This specification is called a ***model of computation***.

One rather powerful model of computation—and the only model we'll talk about in this lecture—is ***decision trees***. A decision tree is, as the name suggests, a tree. Each internal node in the tree is labeled by a *query*, which is just a question about the input. The edges out of a node correspond to the possible answers to that node's query. Each leaf of the tree is labeled with an *output*. To compute with a decision tree, we start at the root and follow a path down to a leaf. At each internal node, the answer to the query tells us which node to visit next. When we reach a leaf, we output its label.

For example, the guessing game where one person thinks of an animal and the other person tries to figure it out with a series of yes/no questions can be modeled as a decision tree. Each internal node is labeled with a question and has two edges labeled 'yes' and 'no'. Each leaf is labeled with an animal.

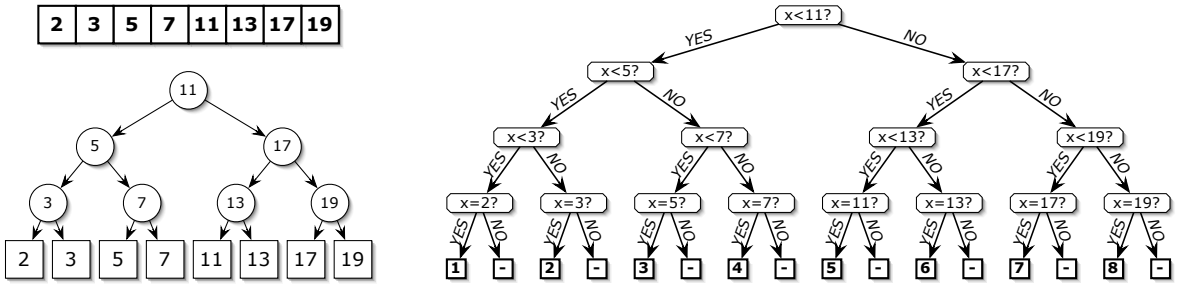


A decision tree to choose one of six animals.

Here's another simple and familiar example, called the ***dictionary problem***. Let A be a fixed array with n numbers. Suppose want to determine, given a number x, the position of x in the array A, if any. One solution to the dictionary problem is to sort A (remembering every element's original position) and then use binary search. The (implicit) binary search tree can be used almost directly as a decision tree. Each internal node in the *search* tree stores a key k; the corresponding node in the *decision* tree stores the question 'Is $x < k$?'. Each leaf in the *search* tree stores some value $A[i]$; the corresponding node in the *decision* tree asks 'Is $x = A[i]$?' and has two leaf children, one labeled 'i' and the other 'none'.

We *define* the running time of a decision tree algorithm for a given input to be the number of queries in the path from the root to the leaf. For example, in the 'Guess the animal' tree above, $T(\text{frog}) = 2$. Thus, the worst-case running time of the algorithm is just the depth of the tree. This definition ignores other kinds of operations that the algorithm might perform that have nothing to do with the queries. (Even the most efficient binary search problem requires more than one machine instruction per comparison!)

[2] Complexity-theory ~~snobs~~ purists sometimes argue that 'all algorithms' is just a synonym for 'all Turing machines'. This is utter nonsense; Turing machines are just another model of computation. Turing machines *might* be a reasonable abstraction of *physically realizable* computation—that's the Church-Turing thesis—but it has a few problems. First, computation is an abstract mathematical process, not a physical process. Algorithms that use physically unrealistic components (like exact real numbers, or unbounded memory) are still mathematically well-defined and still provide useful intuition about real-world computation. Moreover, Turing machines don't accurately reflect the complexity of physically realizable algorithms, because (for example) they can't do arithmetic or access arbitrary memory locations in constant time. At best, they estimate algorithmic complexity up to polynomial factors (although even that is unknown).



Left: A binary search tree for the first eight primes.
Right: The corresponding binary decision tree for the dictionary problem (- = 'none').

But the number of decisions is certainly a *lower bound* on the actual running time, which is good enough to prove a lower bound on the complexity of a problem.

Both of the examples describe *binary* decision trees, where every query has only two answers. We may sometimes want to consider decision trees with higher degree. For example, we might use queries like 'Is x greater than, equal to, or less than y?' or 'Are these three points in clockwise order, colinear, or in counterclockwise order?' A *k-ary* decision tree is one where every query has (at most) k different answers. **From now on, I will only consider k-ary decision trees where k is a constant.**

19.3 Information Theory

Most lower bounds for decision trees are based on the following simple observation: ***The answers to the queries must give you enough information to specify any possible output.*** If a problem has N different outputs, then obviously any decision tree must have at least N leaves. (It's possible for several leaves to specify the same output.) Thus, if every query has at most k possible answers, then the depth of the decision tree must be at least $\lceil \log_k N \rceil = \Omega(\log N)$.

Let's apply this to the dictionary problem for a set S of n numbers. Since there are $n+1$ possible outputs, any decision tree must have at least $n+1$ leaves, and thus any decision tree must have depth at least $\lceil \log_k (n+1) \rceil = \Omega(\log n)$. So the complexity of the dictionary problem, in the decision-tree model of computation, is $\Omega(\log n)$. This matches the upper bound $O(\log n)$ that comes from a perfectly-balanced binary search tree. That means that the standard binary search algorithm, which runs in $O(\log n)$ time, is *optimal*—there is no faster algorithm in this model of computation.

19.4 But wait a second...

We can solve the membership problem in $O(1)$ expected time using hashing. Isn't this inconsistent with the $\Omega(\log n)$ lower bound?

No, it isn't. The reason is that hashing involves a query with more than a constant number of outcomes, specifically 'What is the hash value of x?' In fact, if we don't restrict the degree of the decision tree, we can get constant running time even without hashing, by using the obviously unreasonable query 'For which index i (if any) is $A[i] = x$?'. No, I am *not* cheating — remember that the decision tree model allows us to ask *any* question about the input!

This example illustrates a common theme in proving lower bounds: *choosing the right model of computation is absolutely crucial.* If you choose a model that is too powerful, the problem you're studying may have a completely trivial algorithm. On the other hand, if you consider more restrictive models, the problem may not be solvable at all, in which case any lower bound will be meaningless! (In this class, we'll just tell you the right model of computation to use.)

19.5 Sorting

Now let's consider the classical *sorting* problem — Given an array of n numbers, arrange them in increasing order. Unfortunately, decision trees don't have any way of describing moving data around, so we have to rephrase the question slightly:

> Given a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find the permutation π such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$.

Now a k-ary decision-tree lower bound is immediate. Since there are $n!$ possible permutations π, any decision tree for sorting must have at least $n!$ leaves, and so must have depth $\Omega(\log(n!))$. To simplify the lower bound, we apply *Stirling's approximation*

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta\left(\frac{1}{n}\right)\right) > \left(\frac{n}{e}\right)^n.$$

This gives us the lower bound

$$\lceil \log_k(n!) \rceil > \left\lceil \log_k \left(\frac{n}{e}\right)^n \right\rceil = \lceil n \log_k n - n \log_k e \rceil = \Omega(n \log n).$$

This matches the $O(n \log n)$ upper bound that we get from mergesort, heapsort, or quicksort, so those algorithms are optimal. The decision-tree complexity of sorting is $\Theta(n \log n)$.

Well... we're not quite done. In order to say that those algorithms are optimal, we have to demonstrate that they fit into our model of computation. A few minutes of thought will convince you that they can be described as a special type of decision tree called a *comparison* tree, where every query is of the form 'Is x_i bigger or smaller than x_j?' These algorithms treat any two input sequences exactly the same way as long as the same comparisons produce exactly the same results. This is a feature of any comparison tree. In other words, *the actual input values don't matter, only their order*. Comparison trees describe almost all well-known sorting algorithms: bubble sort, selection sort, insertion sort, shell sort, quicksort, heapsort, mergesort, and so forth—but *not* radix sort or bucket sort.

19.6 Finding the Maximum and Adversaries

Finally let's consider the ***maximum problem***: Given an array X of n numbers, find its largest entry. Unfortunately, there's no hope of proving a lower bound in this formulation, since there are an infinite number of possible answers, so let's rephrase it slightly.

> Given a sequence $\langle x_1, x_2, \ldots, x_n \rangle$ of n distinct numbers, find the index m such that x_m is the largest element in the sequence.

We can get an upper bound of $n-1$ comparisons in several different ways. The easiest is probably to start at one end of the sequence and do a linear scan, maintaining a current maximum. Intuitively, this seems like the best we can do, but the information-theoretic bound is only $\lceil \log_2 n \rceil$. And in fact, this bound is tight! We can locate the maximum element by asking only $\lceil \log_2 n \rceil$ 'unreasonable' questions like "Is the index of the maximum element odd?" No, this is *not* cheating—the decision tree model allows *arbitrary* questions.

To prove a non-trivial lower bound for this problem, we must do two things. First, we need to consider a more reasonable model of computation, by restricting the kinds of questions the algorithm is allowed to ask. We will consider the ***comparison tree model***, where every query must have the form "Is

$x_i > x_j$?". Since most algorithms[3] for finding the maximum rely on comparisons to make control-flow decisions, this does not seem like an unreasonable restriction.

Second, we will use something called an ***adversary argument***. The idea is that an all-powerful malicious adversary *pretends* to choose an input for the algorithm. When the algorithm asks a question about the input, the adversary answers in whatever way will make the algorithm do the most work. If the algorithm does not ask enough queries before terminating, then there will be several different inputs, each consistent with the adversary's answers, the should result in different outputs. In this case, whatever the algorithm outputs, the adversary can 'reveal' an input that is consistent with its answers, but contradicts the algorithm's output, and then claim that that was the input that he was using all along.

For the maximum problem, the adversary originally pretends that $x_i = i$ for all i, and answers all comparison queries accordingly. Whenever the adversary reveals that $x_i < x_j$, he *marks* x_i as an item that the algorithm knows (or should know) is not the maximum element. At most one element x_i is marked after each comparison. Note that x_n is never marked. If the algorithm does less than $n-1$ comparisons before it terminates, the adversary must have at least one other unmarked element $x_k \neq x_n$. In this case, the adversary can change the value of x_k from k to $n+1$, making x_k the largest element, without being inconsistent with any of the comparisons that the algorithm has performed. In other words, the algorithm cannot tell that the adversary has cheated. However, x_n is the maximum element in the original input, and x_k is the largest element in the modified input, so the algorithm cannot possibly give the correct answer for both cases. Thus, in order to be correct, any algorithm must perform at least $n-1$ comparisons.

The adversary argument we described has two very important properties. First, no algorithm can distinguish between a malicious adversary and an honest user who actually chooses an input in advance and answers all queries truthfully. But much more importantly, **the adversary makes *absolutely no* assumptions about the order in which the algorithm performs comparisons.** The adversary forces *any* comparison-based algorithm[4] to either perform $n-1$ comparisons, or to give the wrong answer for at least one input sequence.

Exercises

0. Simon bar Kokhba thinks of an integer between 1 and 1,000,000 (or so he claims). You are trying to determine his number by asking as few yes/no questions as possible. How many yes/no questions are required to determine Simon's number in the worst case? Give both an upper bound (supported by an algorithm) and a lower bound.

1. Consider the following *multi-dictionary* problem. Let $A[1..n]$ be a fixed array of distinct integers. Given an array $X[1..k]$, we want to find the position (if any) of each integer $X[i]$ in the array A. In other words, we want to compute an array $I[1..k]$ where for each i, either $I[i] = 0$ (so zero means 'none') or $A[I[i]] = X[i]$. Determine the *exact* complexity of this problem, as a function of n and k, in the binary decision tree model.

[3] but not all—see Exercise 4

[4] In fact, the $n-1$ lower bound for finding the maximum holds in a much powerful model called *algebraic* decision trees, which are binary trees where every query is a comparison between two polynomial functions of the input values, such as 'Is $x_1^2 - 3x_2x_3 + x_4^{17}$ bigger or smaller than $5 + x_1x_3^5x_5^2 - 2x_7^{42}$?'

2. We say that an array $A[1..n]$ is *k-sorted* if it can be divided into k blocks, each of size n/k, such that the elements in each block are larger than the elements in earlier blocks, and smaller than elements in later blocks. The elements within each block need not be sorted.

   For example, the following array is 4-sorted:

1	2	4	3	7	6	8	5	10	11	9	12	15	13	16	14

   (a) Describe an algorithm that k-sorts an arbitrary array in $O(n \log k)$ time.

   (b) Prove that any comparison-based k-sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst case.

   (c) Describe an algorithm that completely sorts an already k-sorted array in $O(n \log(n/k))$ time.

   (d) Prove that any comparison-based algorithm to completely sort a k-sorted array requires $\Omega(n \log(n/k))$ comparisons in the worst case.

   In all cases, you can assume that n/k is an integer.

3. Recall the nuts-and-bolts problem from the lecture on randomized algorithms. We are given n bolts and n nuts of different sizes, where each bolt exactly matches one nut. Our goal is to find the matching nut for each bolt. The nuts are bolts are too similar to compare directly; however, we can test whether any nut is too big, too small, or the same size as any bolt.

   (a) Prove that in the worst case, $\Omega(n \log n)$ nut-bolt tests are required to correctly match up the nuts and bolts.

   (b) Now suppose we would be happy to find *most* of the matching pairs. Prove that in the worst case, $\Omega(n \log n)$ nut-bolt tests are required even to find $n/2$ arbitrary matching nut-bolt pairs.

   *(c) Prove that in the worst case, $\Omega(n + k \log n)$ nut-bolt tests are required to find k arbitrary matching pairs. *[Hint: Use an adversary argument for the $\Omega(n)$ term.]*

   *(d) Describe a randomized algorithm that finds k matching nut-bolt pairs in $O(n + k \log n)$ expected time.

*4. Suppose you want to determine the largest number in an n-element set $X = \{x_1, x_2, \ldots, x_n\}$, where each element x_i is an integer between 1 and $2^m - 1$. Describe an algorithm that solves this problem in $O(n + m)$ steps, where at each step, your algorithm compares one of the elements x_i with a *constant*. In particular, your algorithm must never actually compare two elements of X! *[Hint: Construct and maintain a nested set of 'pinning intervals' for the numbers that you have not yet removed from consideration, where each interval but the largest is either the upper half or lower half of the next larger block.]*