> *Number Six: What do you want?*
> *Number Two: Information!*
> *Number Six: Whose side are you on?*
> *Number Two: That would be telling. We want information!*
> *Number Six: You won't get it!*
> *Number Two: By hook or by crook, we will!*
>
> — Opening sequence of 'The Prisoner' (1967–68)

H Lower Bounds

H.1 What *Are* Lower Bounds?

So far in this class we've been developing algorithms and data structures for solving certain problems and analyzing their time and space complexity.

Let $T_A(X)$ denote the running of algorithm A given input X. Then the worst-case running time of A for inputs of size n is defined as follows:

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

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

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

Now suppose we've shown that the worst-case running time of an algorithm A is $O(f(n))$. Then we immediately have an *upper bound* for the complexity of Π:

$$T_\Pi(n) \le T_A(n) = O(f(n)).$$

The faster our algorithm, the better our upper bound. In other words, when we give a running time for an algorithm, what we're really doing — and what most theoretical computer scientists devote their entire careers doing[1] — is bragging about how *easy* some problem is.

Starting with this lecture, we've turned the tables. Instead of bragging about how easy problems are, now we're arguing that certain problems are *hard* by proving *lower bounds* on their complexity. This is a little harder, because it's no longer enough to examine a single algorithm. To show that $T_\Pi(n) = \Omega(f(n))$, we have to 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.

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

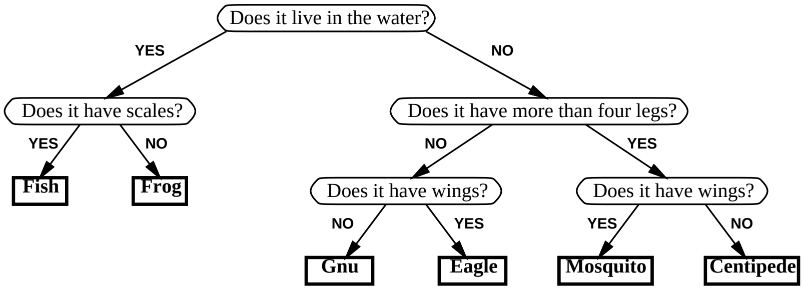
Lennes: "I can triangulate that polygon in $O(n^2)$ time."
Shamos: "I can triangulate that polygon in $O(n \log n)$ time."
Tarjan: "I can triangulate that polygon in $O(n \log\log n)$ time."
Seidel: "I can triangulate that polygon in $O(n \log^* n)$ time."
[Audience gasps.]
Chazelle: "I can triangulate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

H.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, formally, what an algorithm is and how to measure its running time. This specification is called a *model of computation.*

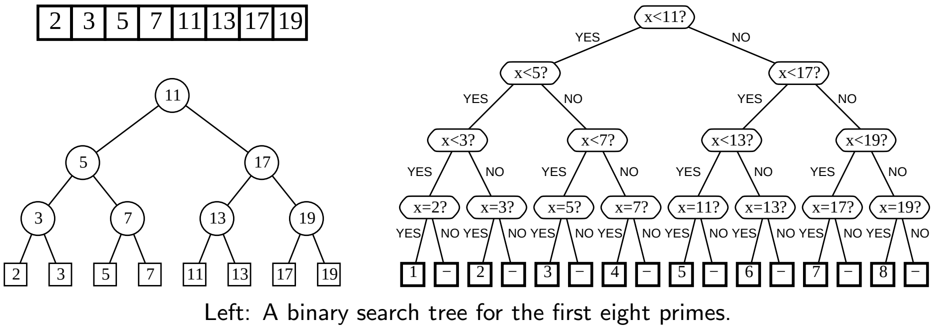
One rather powerful model of computation is *decision trees.* A decision tree is (as the name suggests) a tree. Each internal node in the tree is labeled by a *query*, which is just a question about the input. The edges out of a node correspond to the various answers to the query. Each leaf of the tree is labeled with an *output.* To compute with a decision tree, start at the root and follow a path down to a leaf. At each internal node, the answer to the query tells you which node to visit next. When you reach a leaf, 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 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 the 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'.



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

[2] Complexity-theory purists might argue that 'all algorithms' is just a synonym for 'all Turing machines'. (If you want to know what a Turing machine is, take 375.) In my opinion, this is nonsense. Or it might not be nonsense, but it isn't a particularly *useful* definition. Turing machines are just another model of computation.

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!) 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.**

H.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.

H.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.)

H.5 Sorting

Now let's consider the *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 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 sorting algorithms: bubble sort, selection sort, insertion sort, shell sort, quicksort, heapsort, mergesort, and so forth — but *not* radix sort or bucket sort.

H.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 lower bound is only $\lceil \log_2 n \rceil$.

To prove that $n - 1$ comparisons are actually necessary, we 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 appropriately. 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.

It is very important to notice that the adversary makes *no* assumptions about the order in which the algorithm does its comparisons. The adversary forces *any* algorithm (in this model of computation[3]) to either perform $n-1$ comparisons, or to give the wrong answer for at least one input sequence. Notice also that no algorithm can distinguish between a malicious adversary and an honest user who actually chooses an input in advance and answers all queries truthfully.

In the next lecture, we'll see several more complicated adversary arguments.

[3] Actually, 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 $5 + x_1x_3^5x_5^2 - 2x_7^{42}$?'