Concrete models and lower bounds

In this lecture, we will examine some simple, concrete models of computation, each with a precise definition of what counts as a step, and try to get tight upper and lower bounds for a number of problems. Specific models and problems examined in this lecture include:

- The number of comparisons needed to find the largest and second-largest item in an array.

- The number of comparisons needed to sort an array.

- The number of queries on a graph to determine whether it is connected.

Objectives of this lecture

In this lecture, we want to:

- Understand the concept of a model of computation, with several examples (the comparison model, the query-edge model)

- Understand the definition of a lower bound in a specific model

- See some examples of how to prove lower bounds in specific models, specifically for sorting and selection problems

---

Recommended study resources

- CLRS, *Introduction to Algorithms*, Chapter 8.1, Lower bounds for sorting

- DPV, *Algorithms*, Chapter 2.3, Mergesort (Page 59)

---

1 Terminology: Upper Bounds and Lower Bounds

In this lecture, we will look at (worst-case) upper and lower bounds for a number of problems in several different concrete models. Each model will specify exactly what operations may be performed on the input, and how much they cost. Each model will have some operations that cost a certain amount (like performing a comparison, or swapping a pair of elements), some that are free, and some that are not allowed at all.

> ### Definition: Upper bound
>
> By an *upper bound* of U_n for some problem and some length n, we mean that <u>there exists an algorithm A that for every input x of length n costs at most U_n</u>.

A lower bound for some problem and some length n, is obtained by the negation of an upper bound for that n. It says that some upper bound is not possible (for that value of n). If we take the above statement (in italics) and negate it, we get the following. <u>*for every algorithm A there exists* an input x of length n such that A costs *more than* U_n on input x</u>. Rephrasing:

> ### Definition: Lower bound
>
> By a *lower bound* of L_n for some problem and some length n, we mean that <u>for any algorithm A there exists</u> an input x of length n on which A costs <u>at least</u> L_n steps.

These were definitions for a single value of n. Now a function $f : \mathbb{N} \to \mathbb{R}$ is an upper bound for a problem if $f(n)$ is an upper bound for this problem for every $n \in \mathbb{N}$. And a function $g(\cdot)$ is an lower bound for a problem if $g(n)$ is a lower bound for this problem for every n.

The reason for this terminology is that if we think of our goal as being to understand the "true complexity" of each problem, measured in terms of the best possible worst-case guarantee achievable by any algorithm, then an upper bound of $f(n)$ and lower bound of $g(n)$ means that the true complexity is somewhere between $g(n)$ and $f(n)$.

Finally, what is the *cost* of an algorithm? As we said before, that depends on the particular model of computation we're using. We will consider different models below, and show each has their own upper and lower bounds. When you're writing programs, you typically use rules-of-thumb to calculate the cost of opetations (adding two integers costs 1, copying strings of length ℓ should cost $\approx \ell$, etc) — we make this precise using certain cost models.

One natural model for examining problems like sorting and selection is the comparison model.

> ### Definition: Comparison Model
>
> In the *comparison model*, we have an input consisting of n items (typically in some initial order). An algorithm may compare two items (asking is $a_i < a_j$?) at a cost of 1. Moving the items around is free. No other operations on the items are allowed (using them as indices, adding them, etc).

2   Selection in the comparison model

2.1   Finding the maximum of n elements

How many comparisons are necessary and sufficient to find the maximum of n elements, in the comparison model of computation?

> **Claim: Upper bound on select-max in the comparison model**
>
> $n-1$ comparisons are sufficient to find the maximum of n elements.

Proof. Just scan left to right, keeping track of the largest element so far. This makes at most $n-1$ comparisons. □

Now, let's try for a lower bound. One simple lower bound is that we have to look at all the elements (else the one not looked at may be larger than all the ones we look at). But looking at all n elements could be done using $n/2$ comparisons, so this is not tight. In fact, we can give a better lower bound:

> **Claim: Lower bound on select-max in the comparison model**
>
> $n-1$ comparisons are needed in the worst-case to find the maximum of n elements.

Proof. Suppose some algorithm $\mathscr{A}$ claims to find the maximum of n elements using less than $n-1$ comparisons. Consider an arbitrary input of n distinct elements, and construct a graph in which we join two elements by an edge if they are compared by $\mathscr{A}$. If fewer than $n-1$ comparisons are made, then this graph must have at least two components. Suppose now that algorithm $\mathscr{A}$ outputs some element u as the maximum, where u is in some component C_1. In that case, pick a different component C_2 and add a large positive number (e.g., the value of u) to every element in C_2. This process does not change the result of any comparison made by $\mathscr{A}$, so on this new set of elements, algorithm $\mathscr{A}$ would still output u. Yet this now ensures that u is not the maximum, so $\mathscr{A}$ must be incorrect. □

Since the upper and lower bounds are equal, the bound of $n-1$ is tight.

2.2 Finding the second-largest of n elements

How many comparisons are necessary (lower bound) and sufficient (upper bound) to find the second largest of n elements? Again, let us assume that all elements are distinct.

> **Claim: Lower bound on select-second-max in the comparison model**
>
> $n-1$ comparisons are needed in the worst-case to find the second-largest of n elements.

Proof. The same argument used in the lower bound for finding the maximum still holds. □

Let us now work on finding an upper bound. Here is a simple one to start with.

> **Claim: Upper bound #1 on select-second-max in the comparison model**
>
> $2n-3$ comparisons are sufficient to find the second-largest of n elements.

Proof. Just find the largest using $n-1$ comparisons, and then the largest of the remainder using $n-2$ comparisons, for a total of $2n-3$ comparisons. $\square$

We now have a gap: $n-1$ versus $2n-3$. It is not a huge gap: both are $\Theta(n)$, but remember today's theme is tight bounds. So, which do you think is closer to the truth? It turns out, we can reduce the upper bound quite a bit:

> **Claim: Upper bound #2 on select-second-max in the comparison model**
>
> $n + \lg n - 2$ comparisons are sufficient to find the second-largest of n elements.

Proof. As a first step, let's find the maximum element using $n-1$ comparisons, but in a tennis-tournament or playoff structure. That is, we group elements into pairs, finding the maximum in each pair, and recurse on the maxima. E.g.,
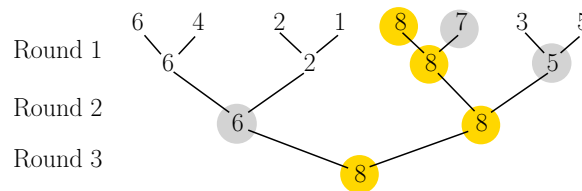
Now, given just what we know from comparisons so far, what can we say about possible locations for the second-highest number (i.e., the second-best player)? The answer is that the second-best must have been directly compared to the best, and lost.[1] This means there are only $\lg n$ possibilities for the second-highest number, and we can find the maximum of them making only $\lg(n)-1$ more comparisons. $\square$

At this point, we have a lower bound of $n-1$ and an upper bound of $n + \lg(n) - 2$, so they are nearly tight. It turns out that, in fact, the lower bound can be improved to exactly meet the upper bound.[2]

3   Sorting in the comparison model

For the problem of *sorting* in the comparison model, the input is an array $a = [a_1, a_2, \ldots, a_n]$, and the output is a permutation of the input $\pi(a) = [a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}]$ in which the elements are in increasing order. We begin this lecture by showing the following lower bound for comparison-based sorting.

---

[1]Apparently the first person to have pointed this out was Charles Dodgson (better known as Lewis Carroll!), writing about the proper way to award prizes in lawn tennis tournaments.

[2]First shown by Kislitsyn (1964).

> ### Theorem 1: Lower bound for sorting in the comparison model
>
> Any deterministic comparison-based sorting algorithm must perform at least $\lg(n!)$ comparisons to sort n elements in the worst case.[a] Specifically, for any deterministic comparison-based sorting algorithm $\mathcal{A}$, for all $n \geq 2$ there exists an input I of size n such that $\mathcal{A}$ makes at least $\lg(n!) = \Omega(n \log n)$ comparisons to sort I.
>
> ---
> [a]As is common in CS, we will use "lg" to mean "$\log_2$".

To prove this theorem, we cannot assume the sorting algorithm is going to necessarily choose a pivot as in Quicksort, or split the input as in Mergesort — we need to somehow analyze *any possible* (comparison-based) algorithm that might exist. We now present the proof, which uses a very nice information-theoretic argument. (This proof is deceptively short: it's worth thinking through each line and each assertion.)

Proof of Theorem **??**. Let us begin with a simple general claim. Suppose you have some problem where there are M possible different outputs the algorithm could require: e.g., for sorting by comparisons where the output can be viewed as a specific permutation of the input, each possible permutation of the input is possible, and hence $M = n!$. Suppose furthermore that for each of these outputs, there exists some input under which it is the only correct answer. This is true for sorting. Then, we have a worst-case lower bound of $\lg M$. The reason is that the algorithm needs to find out which of these M outputs is the right one, and each YES/NO question could be answered in away that removes at most half of the possibilities remaining from consideration. (Why half? Because each comparison has a binary output, so it breaks the set of possibilities into two parts. At least one of these parts contain at least half the possibilities.) So, in the worst case, it takes at least $\lg M = \lg n!$ steps to find the right answer. □

The above is often called an "information theoretic" argument because we are in essence saying that we need at least $\lg(M) = \lg(n!)$ bits of information about the input before we can correctly decide what output we need to produce.

What does $\lg(n!)$ look like? We have: $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ldots + \lg(1) < n \lg(n) = O(n \log n)$ and $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ldots + \lg(1) > (n/2)\lg(n/2) = \Omega(n \log n)$. So, $\lg(n!) = \Theta(n \log n)$.

However, since today's theme is tight bounds, let's be a little more precise. We can in particular use the fact that $n! \in [(n/e)^n, n^n]$ to get:

$$n \lg n - n \lg e \quad < \quad \lg(n!) \quad < \quad n \lg n$$
$$n \lg n - 1.443n \quad < \quad \lg(n!) \quad < \quad n \lg n.$$

Since $1.433n$ is a low-order term, sometimes people will write this fact this as: $\lg(n!) = (n \lg n)(1 - o(1))$, meaning that the ratio between $\lg(n!)$ and $n \lg n$ goes to 1 as n goes to infinity.

How Tight is this Bound? Assume n is a power of 2 — in fact, let's assume this for the entire rest of today's lecture. Can you think of an algorithm that makes at most $n \lg n$ comparisons, and so is tight in the leading term? In fact, there are several algorithms, including:

Binary insertion sort If we perform insertion-sort, using binary search to insert each new element, then the number of comparisons made is at most $\sum_{k=2}^{n} \lceil \lg k \rceil \leq n \lg n$. Note that insertion-sort spends a lot in moving items in the array to make room for each new element, and so is not especially efficient if we count movement cost as well, but it does well in terms of comparisons.

Mergesort Merging two lists of $n/2$ elements each requires at most $n-1$ comparisons. So, unrolling the recurrence, we get $(n-1)+2(n/2-1)+4(n/4-1)+\ldots+n/2(2-1)=n \lg n - (n-1) < n \lg n$.

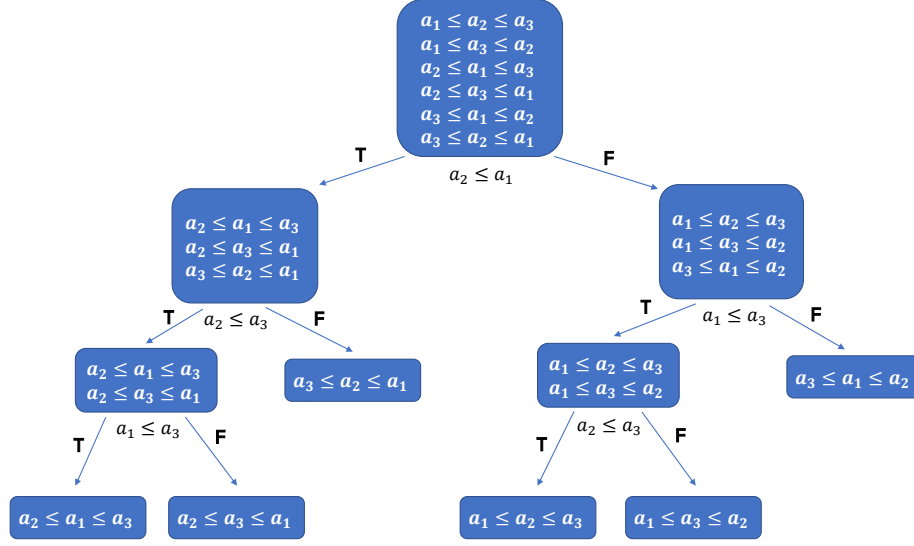# 4   A tree-based view: Decision trees

A handy way to visualize and analyze algorithms in the comparison model is to consider the *decision tree* of the algorithm. A decision tree is a binary tree that is defined as follows.

Definition: Decision Tree

The decision tree is a way of representing an algorithm in the comparison model. It is a binary tree such that:

- At the root node, it contains a list of all the possible valid outputs for the problem. For example, for sorting, the root node contains all $n!$ permutations

- Each internal node corresponds to a particular comparison operation, e.g., $a_5 < a_6$. The left child of the node corresponds to the result of the comparison being true, and the right node corresponds to it being false

- Each internal node consists of a list of possible valid outputs given the results of all of the comparison operations of its ancestors

- Leaf nodes correspond to a single valid output

Note that very importantly, a decision tree corresponds to a **specific algorithm**, but does not depend on the input to the algorithm (except that it may depend on the **size** of the input n). The decision tree encodes the exact behavior of the algorithm, so given a decision tree, you could simulate the algorithm on any given input by starting at the root, performing comparisons, and traveling down a path in the tree until you reach a leaf, which is the output of the algorithm. You can think of it like a flowchart that represents the behavior of the algorithm. Here is an example decision tree for some hypothetical sorting algorithm for $n = 3$.

$a_1 \leq a_2 \leq a_3$
$a_1 \leq a_3 \leq a_2$
$a_2 \leq a_1 \leq a_3$
$a_2 \leq a_3 \leq a_1$
$a_3 \leq a_1 \leq a_2$
$a_3 \leq a_2 \leq a_1$

T $a_2 \leq a_1$ **F**

$a_2 \leq a_1 \leq a_3$
$a_2 \leq a_3 \leq a_1$
$a_3 \leq a_2 \leq a_1$

$a_1 \leq a_2 \leq a_3$
$a_1 \leq a_3 \leq a_2$
$a_3 \leq a_1 \leq a_2$

T $a_2 \leq a_3$ **F**   **T** $a_1 \leq a_3$ **F**

$a_2 \leq a_1 \leq a_3$
$a_2 \leq a_3 \leq a_1$

$a_3 \leq a_2 \leq a_1$

$a_1 \leq a_2 \leq a_3$
$a_1 \leq a_3 \leq a_2$

$a_3 \leq a_1 \leq a_2$

T $a_1 \leq a_3$ **F**   **T** $a_2 \leq a_3$ **F**

$a_2 \leq a_1 \leq a_3$ $a_2 \leq a_3 \leq a_1$ $a_1 \leq a_2 \leq a_3$ $a_1 \leq a_3 \leq a_2$

It turns out that decision trees can be a useful tool for analyzing lower bounds. We must keep in mind that a decision tree always represents a particular algorithm, so to prove a lower bound, we must argue about the structure of *any possible decision tree* for the problem. Observe that the worst-case number of comparisons for the algorithm corresponds exactly to the longest root-to-leaf path, i.e., the height of the tree. Therefore, if we can argue about the height of any possible decision tree, we have an argument for a lower bound! Let's see the sorting lower bound theorem from before proved using decision trees.

Proof of Theorem **??** *using Decision Trees.* There are $n!$ possible sorted orders, each of which must appear as a leaf in the tree. Let the height of the tree be denoted by h, and then note that a binary tree of height h can not have more than 2^h leaves. Therefore $n! \leq 2^h$ and so taking the log of both sides gives us $h \geq \lg n!$ which implies that $\lg n!$ is a lower bound for the problem. $\qquad\square$

4.1   An Adversary Argument

A slightly different lower bound argument comes from showing that if an algorithm makes "too few" comparisons, then an adversary can fool it into giving the incorrect answer. Here is a little example. We want to show that any deterministic sorting algorithm on 3 elements must perform at least 3 comparisons in the worst case. (This result follows from the information theoretic lower bound of $\lceil \lg 3! \rceil = 3$, but let's give a different proof.)

If the algorithm does fewer than two comparisons, some element has not been looked at, and the algorithm must be incorrect. So after the first comparison, the three elements are w the winner of the first query, l the loser, and z the other guy. If the second query is between w and z, the adversary replies $w > z$; if it is between l and z, the adversary replies $l < z$. Note that in either case, the algorithm must perform a third query to be able to sort correctly.

In this kind of argument the goal is to construct an adversary Ada who will answer the algorithm's comparisons in such a way that (a) all Ada's answers are consistent with some input In,

and (b) her answers make the algorithm perform "many" comparisons.

5   Query models, and the evasiveness of connectivity

To finish with something totally different, let's look at the query complexity of determining if a graph is connected. Assume we are given the adjacency matrix G for some n-node graph. That is, $G[i, j] = 1$ if there is an edge between i and j, and $G[i, j] = 0$ otherwise. We consider a model in which we can *query* any element of the matrix G in 1 step. All other computation is free. That is, imagine the graph matrix has values written on little slips of paper, face down. In one step we can turn over any slip of paper. How many slips of paper do we need to turn over to tell if G is connected?

> ### *Claim: Upper bound*
>
> $n(n-1)/2$ queries are sufficient to determine if G is connected.

Proof. This just corresponds to querying every pair (i, j). Once we have done that, we know the entire graph and can just compute for free to see if it is connected.   □

Interestingly, it turns out the simple upper-bound of querying every edge is a lower bound too. Because of this, connectivity is called an "evasive" property of graphs.

> ### *Claim: Lower bound*
>
> $n(n-1)/2 = \binom{n}{2}$ queries are necessary to determine connectivity in the worst case.

Proof 1. We think of this as a game between two players: the *evader* and the *querier*. The querier asks a sequence of questions of the form "Is there an edge between vertices x and y?". For each question the evader must answer the question. The goal of the evader is to force the querier to ask as many questions as possible. We'll give a strategy for the evader which forces the querier to ask $\binom{n}{2}$ questions to determine if the graph is connected.

The evader's strategy will be to maintain the following invariant.

> At any point in time the edges that have been declared to exist form a forest of trees involving all the vertices of the graph. For each tree T all queries among the vertices of the T have already been asked. And for every pair of trees T and T' in the forest there exists a pair (x, y) with $x \in T$ and $y \in T'$ that has not been queried.

Note that until there is just one tree, the querier does not know if the graph is connected or not.

Initially each tree is just one vertex, and the invariant trivially holds. The evader maintains the invariant as follows. If the query is for two vertices in the same tree, then the evader just gives the answer it gave before to this query. Nothing changes and the invariant still holds.

Say the query is for vertices x and y in two different trees T_x and T_y. Then the evader determines if all the other possible edges between T_x and T_y have already been queried. If that is the case, then the evader answers 1 for "yes", otherwise it answers 0 for "no". Note that if it answers yes, (and joins the two trees) then, by induction, all the edges within the set $T_x \bigcup T_y$ have already been queried, and the invariant holds. (If it answers "no" the invariant trivially still holds.)

At any point in time before there is just one tree it is not known if the graph is connected or not. Finally when there is just one tree the graph is connected, but by that point all of the $\binom{n}{2}$ the edges have been queried. $\qquad\square$