Lower Bound

An algorithm for a class $\mathcal{P}$ of problems is a computable procedure A to solve any instance I from that class. Example problem classes: sort finite number sequences, find minimum spanning trees of weighted graphs. Notation below: P is an instance of $\mathcal{P}$, and $|P|$ is the size of the instance. Example: instance is a particular weighted graph of V vertices, k edges, k weights; size is $V + k + k$ in some representation.

Given an algorithm A for a class $\mathcal{P}$, let its runtime on an instance P be $T_A(|P|)$. Example: selection sort on an N length number sequence has runtime N^2 (up to Θ equivalence).

A lower bound on algorithms for a problem class $\mathcal{P}$ is a function $T(n)$, such that for any algorithm A for the class there is some problem instance P the runtime $T_A(|P|) \geq T(|P|)$. Watch the order of the quantifiers: $\forall A \exists P \ldots$.

You saw an information theoretic agrument that a lower bound for sorting N sequence numbers is $NlogN$ if comparisons are counted.

We will look at a different technique for establishing lower bounds, called adversary arguments.

Some observations: (i) if $T(n)$ is a lower bound, then so is $U(n)$ for any function such that $U(n) \in \mathcal{O}(T(n))$. So, in many cases $\mathcal{O}(1)$ will be a lower bound, but because it is very loose it tells us nothing. (ii) If $T(n)$ is a lower bound for class $\mathcal{P}$ and you find an algorithm A for that class such that $T_A(n) = T(n)$, then two things follow — A is optimal, and $T(n)$ is the tightest lower bound.

Adversary

The adversary technique in establishing lower bounds can be viewed thus.

Imagine an algorithm working its way to a solution of P. As it does that, along the way it has to access information that can be regarded as questions, e.g., is (u, v) an edge of this graph? is $contents(x) > contents(y)$? What should an adversary do?

One possibility is for the adversary to pick a very hard instance of the problem (see the graph connectivity example below).

Another is for the adversary to change the data as the algorithm proceeds. This must be done in such a way that (i) the change does not affect past decisions of the algorithm — e.g., suppose $x > y$ before, then for changed values we must have $x' > y'$; and (ii) the changes force the algorithm to work very hard from then on. In this example, the purpose of changing from x to x' (likewise for y) is to affect future comparisons with x'.

It is OK for an adversary to do a pass through the algorithm to find ways of doing the above changes to get a difficult input for it. It may store auxiliary information for its own use to do that. There is no cost for adversary tactics. See the adversary for finding the second largest key for an example.

Example I — Find the Maximum

Find the maximum of a set of N keys $\{X_1, \ldots, X_N\}$. Significant operation: compare two keys. What is a lower bound on the number of comprisons?

The adversary changes future key values to force any algorithm to use at least $N-1$ comparisons. An intuitive model of this is to think of the keys as tennis players who have transitively ranked abilities, so if A is better than B and B is better than C, then A is better than C. Any player J who has not lost a match by the end of the game is still a possible best player. If there is more than one such non-defeated player J, the adversary can always alter its rank to remain consistent with past match results and make it the best player.

So, if the algorithm to find the maximum (best player) is to work, every non-maximum key (non-best player) has to have been compared (played a match) and found to be lower (lost).

Hence there needs to be at least $N-1$ comparisons. The selection algorithm uses $N-1$ comparisons, and is therefore optimal and the lower bound is tight.

Example II — Test a graph for connectivity

Given a graph G of $2N$ vertices, how many vertex pairs (v_i, v_j) must be queried (“Is (v_i, v_j) an edge of G?”) so that an algorithm can determine if G is connected. A lower bound is N^2.

Here is an adversary for the problem to show this.

Partition the vertices of G into two equal subsets, V_1 and V_2, each with N vertices. Within each partition the adverasary will answer queries as if the subgraphs are complete. Across partitions it will answer ”no”, meaning the graph G is disconnected. For any algorithm to discover the disconnection across the partition it has to ask N^2 queries.

Observe that there is no cost associated with the adversary’s decisions.

Remarks on Computational Model

The same graph question for connectivity will have a different lower bound answer if the model of what counts as permissible questions (i.e. significant costs) changes.

For instance, if the problem description is altered so that the instance is a graph G with its vertices and edges given, then here is a simple way to determine not only connectivity but also connected components:

Initialize by regarding each vertex v as its own component $C(v)$. Subsequently, if an edge $e = (v_1, v_2)$ joins two components $C(v_1)$ and $C(v_2)$, merge them into a single component. (Notationally, for if u and v are in the same component, $C(u) = C(v)$.)

Depending on how you count the cost of merging and edge processing, this is is proportional to the sum of the number of vertices and edges.

Can you think of an adversary for this?

Example III — Find Largest and Second-largest Keys

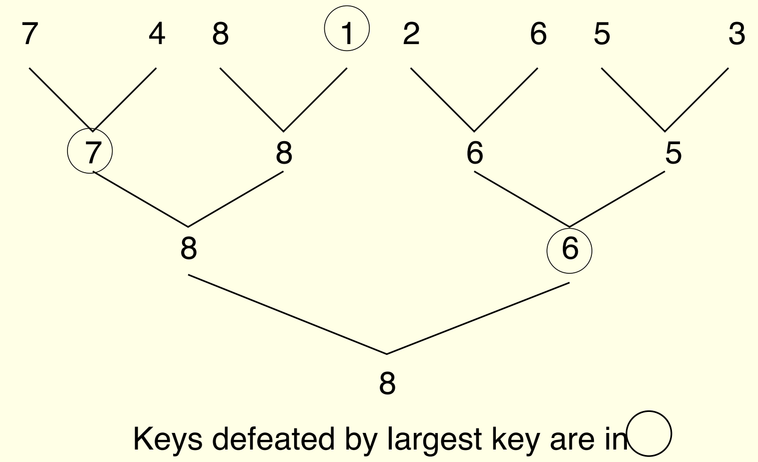
What is a good algorithm for finding not only the largest but also the second largest keys among N keys?

We saw that to find the largest of N keys we need $N-1$ comparisons and there is an optimal algorithm — selection — to do it. To quickly get an algorithm to find the second largest key, we apply selection again to the remaining $N-1$ keys, for which we know that $N-2$ comparisons are needed — (or are they?) — hence altogether we use $2N-3$ comparisons.

However, in the first phase of finding the largest, we have actually accumulated information that we did not use in the second phase!

A Better Algorithm — Find Largest and Second-largest Keys

The following observation, attibuted to Lewis Carroll of Alice in Wonderland fame, extracts information from the first phase in this way: The second largest key has to be among the ones that lost in a comparison with the largest one, no matter what algorithm is used. Using this and a "tournament" as suggested by:



we obtain a much better algorithm. The number of keys that lost to the largest key is the height of the tree, i.e. $logN$. Then using the selection algorithm to find the largest among them takes $logN - 1$ comparisons. Thus the total number of comparisons to find the largest and second largest keys is $N + logN - 2$.

There is an adversary that shows this is also a lower bound, so the tournament algorithm is optimal. The basic intuitive idea in the adversary is to adjust the original keys to new ones (yet remaining consistent with the results of comparisons done up to now) so as to force there to be at least different $logN$ losers to the largest key. This is done after each comparison. This ensures that any algorithm has to find the maximum among the losers, hence has to use at least $logN - 1$ comparisons to do it.

The adversary described

The adversary (call it B) will maintain "weights" for each key that the algorithm (call it A) uses in a comparison. The weights are auxiliary information used only by B and are not part of the data for A. These weights are changed by B as A executes. Initially B sets the weights of all keys to 1, hence the sum of all weights is N. B maintains this sum as the invariant. If A compares x with y, B adjusts their new weights according to these conditions and gives an answer: (i) if $W(x) > W(y)$, B says $x > y$ and changes the weights to $W'(x) = W(x) + W(y)$ and $W'(y) = 0$; (ii) if $W(x) = W(y) > 0$, B does as in previous case; (iii) otherwise B says anything that does not disturb past answers, and does not change weights.

It is not hard to see the following: (a) $W(x) = 0$ iff x has lost in a comparison (b) if $W(x) > 0$, x has not lost yet and can still be the largest key (c) the sum of all weights is always N. Also, A cannot correctly terminate unless there is a unique x such that $W(x) > 0$, in which case $W(x) = N$. This unique x (which is the largest key) has had how many increments $W_1(x), W_2(x), \ldots, W_k(x)$ from its initialized weight of 1? If you follow the algebra, k is at least $logN$, and each increment is due to the defeat of a potential second largest key. Hence there must be at least $logN$ losers.

It is also not hard to use the above to actually find a sequence of keys as input for A to force it to follow exactly the comparisons as indicated.