Supplement : A Lower Bound for MAX-AND-MIN[1]

In class, I said that *any* algorithm for finding the maximum element of an array $A[1:n]$ must make $\geq n-1$ comparisons. I didn't give any good reason for it. And after seeing the $\approx \frac{3n}{2}$ comparison algorithm for MAX-AND-MIN, perhaps you should be a little doubtful.

The statement above is an example of a "lower bound" statement; it is asserting a lower bound on the abilities of algorithms. It is a statement of the form you *can't* do that. Such statements are much harder to prove than statements of the form, "Yes, we can." After all, for the latter, we just need to show an algorithm running in so-and-so time, and we are done. To show the lower bound, we need to somehow argue, not any of the infinitely many algorithms possible for solving our computation problem can run in time better than what asserted. We can do this only for a handful of problems.

Theorem 1. Any correct algorithm for finding the maximum of an n element array must make $(n-1)$ comparisons among the array elements.

Remark: *One should be careful about what the above theorem and the forthcoming proof is really saying. The "time" is calculated as the number of comparisons made between elements of the array. So, if for some reason, your algorithm compares $A[1]+A[2]$ with $A[3]+A[4]$, which elements are being compared? The proof below will assume that there are* four *comparisons being made: $(A[1], A[3])$, $(A[1], A[4])$, and $(A[2], A[3])$, $(A[2], A[4])$. More formally, it is a theorem in the "comparison model" where the array items are assumed to be immutable objects which cannot be arithmetically manipulated, but only compared against.*

Proof. Any algorithm in the comparison model runs as follows: at any step it makes a comparison between $A[i]$ and $A[j]$ for some two indices, and given the result of this comparison, it makes some calculations, and then makes the next comparison. Fix any array $A[1:n]$ and consider the run of the algorithm. As the algorithm runs, construct the following graph $G=(V,E)$. The vertices $V=\{1,2,\ldots,n\}$ correspond to the *indices* of the array. Every time the array compares $A[i]$ and $A[j]$, add the edge (i,j) to the set E. After the algorithm completes, and returns an answer, say $A[k]$, our graph is fully defined. If the algorithm makes $<(n-1)$ comparisons, then this graph has $<(n-1)$ edges.

To illustrate this, Figure 1 below shows what the graph is for the MAX algorithm we in the lecture notes, on the array $[20,-3,14,39,7,16]$.

Now, the graph G has n vertices but $<(n-1)$ edges. This implies that the graph G is *not connected*. This spells trouble for the algorithm. The algorithm has decided to return the number $A[k]$. Let U be the set of vertices v which are connected to k, that is, have a path in G from v to k. Since G is not connected, there is some vertex ℓ which is *not* connected to k. Let X be the set of all these vertices. Note that there is *no* edge (i,j) in G with $i \in X$ and $j \notin X$.

Here is the kicker. Consider an array $B[1:n]$ which is defined as: $B[i]=A[i]+M$ for $i \in X$, and $B[i]=A[i]$ otherwise, where M is a large enough integer such that $B[k]$ (which is $A[k]$) is *not* the maximum in $B[1:n]$. On the other hand, note that the *answers* to the comparisons made by the algorithm

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

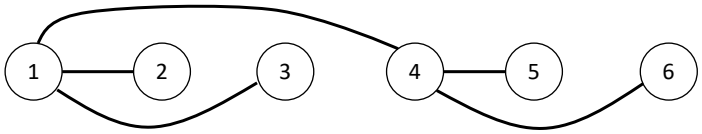

Figure 1: The graph obtained upon running MAX on $[20, -3, 14, 39, 7, 16]$. First $(20, -3)$ are compared, then $(20, 14)$, and then $(20, 39)$. At this point, we move running max to 39, and then compare $(39, 7)$ and $(39, 16)$.

is *exactly the same* in both B and A. This is because, whenever the algorithm is comparing i and j, these two either both lie in X (in which case the answer in B as the same as in A), or both outside X (in which case B is the same as A). Why can't it be that $i \in X$ and $j \notin X$? Because, then (i, j) would be an edge in G, and such edges don't exist.

Since the comparison results are the same in both A and B, the answers given by the algorithm also must be the same. (Here we are using the comparison model; the algorithm is not using any "external data" to make its decision.) So, the algorithm would again answer $B[k]$, and thus would be wrong.

□