

Sample proofs involving selection problems

During lecture, we looked at several proofs involving comparison-based bounds for various selection problems. In this document, we are providing the formal proofs, in written form. This is meant to serve as a review of the lecture material, but also as a model of a well-written proof.

Proving lower bounds using an adversarial argument

The issue here is that we want to prove a lower bound on the performance of any possible algorithm, however we do not want to have to try to enumerate the list of “possible algorithms.” One method for making such a general claim is by using what is known as an *adversarial argument*.

We can generally think of a game where a person gets to pick a “secret” input and then an algorithm starts asking the person to answer questions about the input (i.e., comparisons) until the algorithm is able to determine the final answer. The number of comparisons used by the algorithm is the cost on that input.

If we consider this person to be a particularly clever (or cruel) adversary, he may take advantage of the input being a “secret” and actually never really choose the exact input at the beginning of the game. The adversary can start running the algorithm and answering questions in a way so that there are many possible inputs which would all be consistent with the set of answers which have been given thus far. (This is similar to playing the game “20 questions” against a cheater!)

Of course, the adversary cannot give completely non-sensical answers to the questions; at any point, the adversary must be able to provide a specific input which is consistent with all of the answers he has given. But so long as the adversary is able to provide two different inputs with different final answers, both of which are consistent with the answers given, then the adversary can be sure that the algorithm has not yet determined the final answer.

We can use such an argument about any general algorithm A , so long as we give rules of how an adversary would answer questions in a way to force A to use a lot of comparisons before discovering the final answer.

Finding both the minimum and maximum elements

In this section, we show an algorithm that finds the minimum and maximum elements of n items using at most $\lceil \frac{3n}{2} \rceil - 2$ comparisons in the worst case. Further, we prove that no algorithm can guarantee to use fewer comparisons, in the worst case.

Upper Bound

An algorithm is discussed in Chapter 9.1 of CLRS, finding both the minimum and maximum elements using at most $\lceil \frac{3n}{2} \rceil - 2$.

Note, this total is stated slightly differently in the text. However, it is easy to verify that $\lceil \frac{3n}{2} \rceil - 2 = \frac{3n}{2} - 2$ for even values of n , and $\lceil \frac{3n}{2} \rceil - 2 = 3\lfloor \frac{n}{2} \rfloor$, for odd values of n .

Lower Bound

Theorem 1. *No algorithm which simultaneously finds both the minimum and maximum from a set of totally ordered elements can guarantee a worst-case performance of c comparisons for $c < \lceil \frac{3n}{2} \rceil - 2$.*

Proof: At any point in time, the set of answers which have been given to an algorithm break the set of items into four groups.

- Unknown – elements which have not yet been involved in a comparison
- Winners – items which have won one or more comparisons, but which have never lost a comparison
- Losers – items which have lost one or more comparisons, but which have never won a comparison
- Eliminated – items which have won one or more comparisons and lost one or more comparisons.

We will refer to these sets as U , W , L , and E respectively. We make three observations, the first of which is that at the beginning of time, all items are in the set U . The second observation is that the algorithm cannot be sure of the final answer until it gets in a situation with exactly one item in group W (i.e., the max), one item in group L , and the rest of the items in group E . Finally, we notice that individual elements cannot jump directly from group U to group E based on one comparison, they must first stop in either group W or group L . If we then count the number of “steps” which are taken overall by items in moving from one group to another, there must be at least $2n - 2$ steps before

an algorithm determines the final answer (most elements take two steps, two elements take only one step).

Now we give an adversarial strategy which forces any given algorithm to use at least $\lceil \frac{3n}{2} \rceil - 2$ comparisons to solve the problem. Throughout the process, the adversary will also monitor the sets U , W , L , and E , and it will keep track of the exact partial order which is defined by all previous answers it has given. We note that the partial order can always be extended by the adversary so that all items in W are larger than items of E which in turn are larger than items in L .

Now, whenever the algorithm asks a question of the adversary, the adversary will answer the question using the following rules:

- A comparison is made between two items from U : The adversary will choose a winner, in effect moving one of these items from U to W and one item from U to L .
- A comparison is made between two items from W : The adversary will choose a winner, in effect moving one of these items (the loser of the comparison) from W to E .
- A comparison is made between two items from L : The adversary will choose a winner, in effect moving one of these items (the winner of the comparison) from L to E .
- A comparison is made between an item from U and an item from W : The adversary will say that the item from U is the smaller, in effect moving that item from U to L , while leaving the other item unchanged.
- A comparison is made between an item from U and an item from L or E : The adversary will say that the item from U is the larger, in effect moving that item from U to W , while leaving the other item unchanged.
- A comparison is made between an item from W and an item from L or E : The adversary will say that the item from W is the larger, in effect not moving either item between groups.
- A comparison is made between an item from L and an item from E : The adversary will say that the item from E is the larger, in effect not moving either item between groups.
- A comparison is made between an item from E and another item from E : In this case, the adversary will ensure that it gives an answer which is consistent with the partial order representing all previous answers. In either case, both items remain in group E .

It can be verified that the above list comprises all possible cases for a comparison, and that the answer given is always consistent with all previous answers. If we now consider the number of “steps” which take place moving items from one group to another, we notice the first type of comparison (U vs. U) effects two steps. All of the other comparison types will effect at most one step.

Since we earlier saw that an algorithm cannot complete until there have been at least $2n - 2$ such steps, we are now ready to prove the final result. The key is that the algorithm can make at most $\lfloor \frac{n}{2} \rfloor$ comparisons of the type U vs. U , because each such comparison removes both elements from the set U . Therefore, these types of comparisons account for at most $2\lfloor \frac{n}{2} \rfloor$ of the “steps” and this means that the other type of comparisons must account for at least $2n - 2 - 2\lfloor \frac{n}{2} \rfloor$ steps. Overall, the algorithm is forced to do at least $\lfloor \frac{n}{2} \rfloor + 2n - 2 - 2\lfloor \frac{n}{2} \rfloor$ comparisons. Using the identity $n = \lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil$, we can verify that $\lfloor \frac{n}{2} \rfloor + 2n - 2 - 2\lfloor \frac{n}{2} \rfloor = n + \lceil \frac{n}{2} \rceil - 2 = \lceil \frac{3n}{2} \rceil - 2$, proving the theorem. \square

Finding both the largest and second largest elements

Upper Bound

For $n = 2$ and 3 it is clear how to find the largest and second largest elements using $n + \lceil \lg n \rceil - 2$ comparisons.

For a larger value of n , pair off the elements into $\lfloor n/2 \rfloor$ pairs and choose the largest element in each pair to form a new array. If an element was unpaired, put it in the new array. Now, the new array is of size $\lceil n/2 \rceil < n$. So we can apply induction: by induction hypothesis it is possible to find the largest element A and second largest element B in this new array using

$$\lceil n/2 \rceil + \lceil \lg \lceil n/2 \rceil \rceil - 2 = \lceil n/2 \rceil + \lceil \lg n \rceil - 3$$

comparisons¹. Using at most one more comparison we can now find the largest two elements of the given array. Clearly A is the largest. Now let C be the element that A was paired with. If the second largest element did not make it to the new array, it must be because it lost the comparison with A . Therefore the second largest element is $\max(B, C)$. If A was unpaired the second largest element is B .

So the total number of comparisons used is $\lfloor n/2 \rfloor$ for the pairing plus $\lceil n/2 \rceil + \lceil \lg n \rceil - 3$ for the recursive subproblem plus 1 after the subproblem is solved (at

¹To be really careful, we should prove that $\lceil \lg \lceil n/2 \rceil \rceil = \lceil \lg n \rceil - 1$. But that's easy: suppose $\lceil \lg n \rceil = k + 1$ so that $2^k < n \leq 2^{k+1}$. Then $2^{k-1} < n/2 \leq \lceil n/2 \rceil \leq 2^k$, so that $k - 1 < \lg \lceil n/2 \rceil \leq k$, whence $\lceil \lg \lceil n/2 \rceil \rceil = k$.

most). Since $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$, the number of comparisons is upper-bounded by $n + \lceil \lg n \rceil - 2$ as desired.

Another way to look at it: A more common way to find the two largest elements is to first run a "single-elimination tournament" to find the largest element. Since everyone except the winner loses exactly once, this part of our procedure uses exactly $n - 1$ comparisons.

Now, we know that the second largest element could only have been defeated by the winner, and so we can run a secondary tournament for all elements which lost to the winner. Since a tournament will require at most $\lceil \lg n \rceil$ rounds (giving first-round byes, if not exactly a power of two). Therefore, the secondary tournament will use at most $\lceil \lg n \rceil - 1$ comparisons, and together, we have used at most $n + \lceil \lg n \rceil - 2$ comparisons.

Lower Bound

Theorem 2. *No algorithm which simultaneously finds both the largest and second largest from a set of totally ordered elements can guarantee a worst-case performance of c comparisons for $c < n + \lceil \lg n \rceil - 2$.*

Proof: Let's assume that A is the largest element, B the second largest, and then the other $(n - 2)$ below that. For any algorithm to be sure of its answer, it certainly must know that A is greater than B , or else we could switch the values of A and B without the algorithm realizing it, and it would incorrectly identify the second largest element. Similarly, an algorithm must have identified the group of $(n - 2)$ elements which lie below B . In order to verify that all of these elements do indeed lie below B , each element of this set must have lost a comparison either to B or else to some other member of this group². This implies that for any input, an algorithm must do at least $n - 2$ comparisons which do *not* involve the maximum element. Now we will show that an adversary can force any algorithm to do at least $\lceil \lg n \rceil$ comparisons which do involve the maximum element, and therefore such an algorithm performs at least $n + \lceil \lg n \rceil - 2$ comparisons overall.

To do so, we will consider the following adversary. Given a set of previous comparison answers, we will say that an answer to a new comparison is "known" if that answer is the same for all total orders which are consistent with the previous comparisons. Our adversary will maintain for each element x , the set $L(x)$ of elements which are "known" to be less than or equal to x based on

²If there were some element in the lower group for which this was not true, then we could increase that element's value to be greater than B without violating any of the comparisons. Therefore the algorithm's answer for the original input and this modified input would be identical, and yet B is not the second largest element in the modified case.

previous comparisons. Initially, $L(x) = \{x\}$ for all elements, as the only thing we can be sure of is that x is less than or equal to itself. Now, when asked the question, “is $a \leq b$?”, if this answer is known, our adversary will give the known answer. Otherwise, our adversary will say that a wins the comparison exactly when $|L(a)| \geq |L(b)|$. By answering questions in this way, we can be sure that there will always be at least one total ordering which is consistent with all of our adversary’s answers.

Using this adversary, we claim that if a particular element x has been involved in exactly k comparisons, then $|L(x)| \leq 2^k$. We prove this by induction. As a base case, we have seen that when $k = 0$, $|L(x)| = 1 = 2^0$. For the inductive step, assume that $|L(x)| \leq 2^{k-1}$ after $k - 1$ comparisons involving x , and we consider the k^{th} such comparison between x and some y . In the case where x loses the comparison or if it was already “known” that x was at least as large as y , then no new elements are added to $L(x)$, and so $|L(x)| \leq 2^{k-1} \leq 2^k$. If the answer was not previously known, and the adversary answers that x wins the comparison, then it must have been the case that $|L(y)| \leq |L(x)|$. If $L'(x)$ is equal to the resulting set after the k^{th} comparison, then we see that $L'(x) = L(x) \cup L(y)$. But these are the only elements which can be added to $L(x)$. Therefore, $|L'(x)| \leq |L(x)| + |L(y)|$ (some of $L(y)$ may have already been in $L(x)$). However, by induction we know that $|L(x)| \leq 2^{k-1}$, and by the adversary’s rules, we know that $|L(y)| \leq |L(x)| \leq 2^{k-1}$, and therefore, $|L'(x)| \leq 2^{k-1} + 2^{k-1} = 2^k$.

Finally, we are ready to show that any algorithm will have made at least $\lceil \lg n \rceil$ comparisons which involve the maximum element. Notice that since any algorithm eventually knows the maximum element A , then it must be that $L(A) = n$. If A was involved in k comparisons, we know that $n = |L(A)| \leq 2^k$, and so rearranging, we see that $k \geq \lg n$. Since k must be integral, we see that $k \geq \lceil \lg n \rceil$, and therefore, the maximum element was involved in at least $\lceil \lg n \rceil$ comparisons. Based on the first paragraph, this completes our proof. \square