

Lower Bounds

0.0.1 Lower Bounds

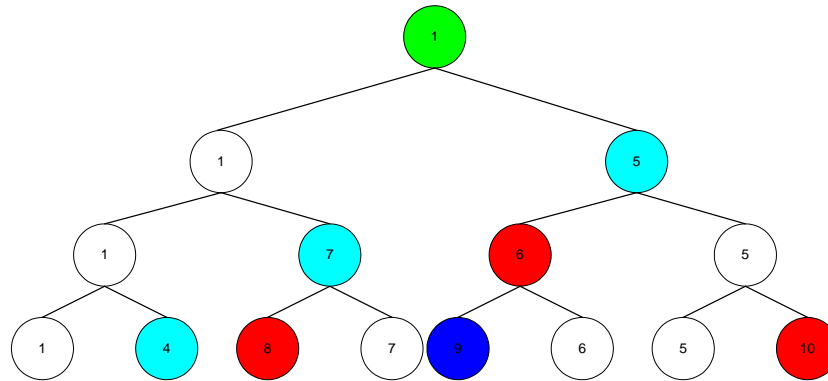
Example 1 *Given an array of numbers $A[1, 2, \dots, n]$, find the maximum and the minimum of the set of the given numbers using only pairwise comparisons.*

Example 2 *Given an array of numbers $A[1, 2, \dots, n]$, find the minimum and the second lowest elements of the set of the given numbers using only pairwise comparisons.*

For both these problems, the straight forward process of finding one and then the other takes $2n - 3$ comparisons. Again, for both these problems, divide-and-conquer methods give a better bound of $\lceil \frac{3}{2}n - 2 \rceil$. We now ask if there is a better algorithm for these two problems.

A similar question for sorting using only comparisons is treated in the book in pp. 172-4.

For the problem of finding minimum and second minimum, there is a better algorithm based on tournaments. Here we compare $A[2i - 1]$ with $A[2i]$ for $i = 1$ to n . (We assume for simplicity that n is even.) We compare the "winners" in pairs and so on. For example, see the diagram below:

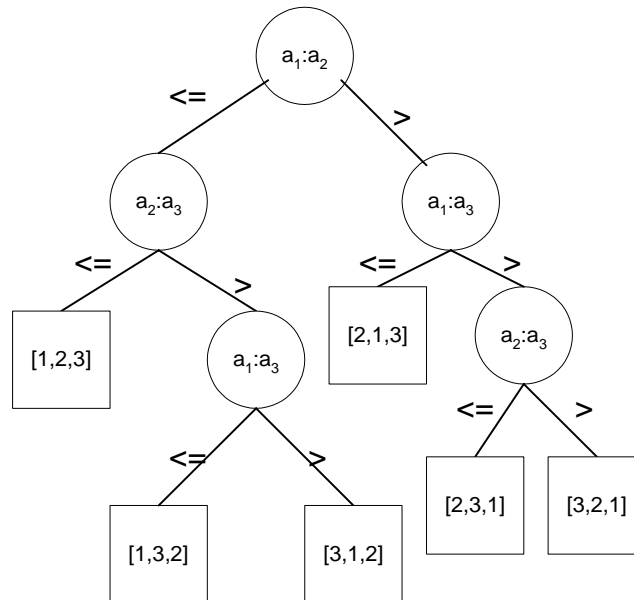


The light blue cells are those that lost to the final winner (in green). The red are those that lost to the light blue ones and dark blue is one that lost to a red. So for the second minimum we need only compare the light blue ones. There is exactly one of these per level and there are $\lceil \lg n \rceil$ levels and hence we get a total of $n + \lceil \lg n \rceil - 2$ comparisons in all.

The question about the existence of still better algorithms still is around. Now we show that any algorithm based on comparisons has lower bounds equal to the amount of work done by these algorithms for these two problems. But before that, we show lower bounds for some other simpler problems.

Sorting: Decision Tree Model (Chapter 9.1)

Using only comparisons of pairs of elements, we want to sort an array of n elements. That is given two elements, a_i and a_j , we perform one of the following tests: $a_i < a_j$; $a_i \leq a_j$; $a_i = a_j$; $a_i \geq a_j$; $a_i > a_j$. This determines the relative order among these two elements. In order to get lower bounds, we may assume the input has special characteristics with out loss. For, if the special case has a lower bound, then this lower bound applies to the general case as well. In this example, we assume that the elements are distinct. Given this assumption, we never perform the middle test and the remaining are equivalent. So from now on we talk of a test in the form $a_i > a_j$. Any algorithm does a comparison and based on the outcome of this comparison, branches into two parts and for each part does another comparison (and these two do not have to be the same) and so on. This creates a binary tree called the decision tree. For example, here is one taken from your book :



This corresponds to insertion sort. The square boxes represent possible permutations of the input. Thus, this tree must have at least $n!$ leaves. The complexity of the algorithm in the worst case, corresponds to the depth of the tree, Since this is a binary tree, if h is depth, then we must have

$$n! \leq 2^h$$

and hence the lower bound on the complexity of any algorithm based only on comparisons is $\lg(n!) = \Omega(n \lg n)$.

Adversary Arguments:

In order to get lower bounds we employ an adversary scenario. The algorithms designer makes the decision as to, say, the next comparison. Knowing

this, an adversary gives the result of this comparison so as to make the algorithm look bad (i.e. take many steps). The only restriction is that the adversary needs to be consistent. Since the entire input is not used in the algorithm at any step, and only the results of the comparisons can be used, the adversary does not need to specify the input till the end. But when it does, this must be consistent with all the answers given by it from the beginning. The key is to design a strategy for the adversary so that the algorithm designer is forced to take as long a time as possible. Let us take some examples to illustrate this approach.

Example 3 Find minimum of an unsorted array $A[1, 2, \dots, n]$ of n numbers by using comparisons of pairs of elements.

For the purposes of lower bound analysis, we may assume without loss that the elements are distinct. Let us call the smaller of two elements in a comparison the "winner" and the other the "loser". For the purposes of lower bounds we assume that the elements are distinct. At any stage, we have possibly two types of elements: (A) those that have lost to some other element; (B) those that have not lost any comparison. In the beginning we have only type B. When the algorithm designer, chooses to compare two elements at any stage of the algorithm, the comparison may be one of three types: AA, BA, or BB. The adversary strategy that we choose is the following:

$\begin{array}{c} y \leftarrow \\ x \downarrow \end{array}$	B	A
B		$x < y$
A	$x > y$	

In the above, blank space refers to being consistent with previous answers. Note that the specific answers in the other cases also maintains consistency. This can be done by suitably increasing the value of the B type.

In this process, we count the number of elements of type A. In order for the algorithm to end, we must have $(n - 1)$ elements of type A. Let us call this count, the information content of the algorithm. The adversary is trying to keep this as low as possible while the algorithm designer is trying to increase this quantity. Note that when a comparison of the type BB is made, we increase this quantity by 1 regardless of what the adversary says. But in the remaining cases, the adversary strategy does not allow any increase of this quantity. **So the best the algorithm designer can do is make sure that at each step, we are doing a BB type comparison.** And in this case, we take $(n - 1)$ comparisons. Thus, this is a lower bound for this problem. **Any algorithm that only does BB type comparison is optimal.** Thus, out of this analysis, we not only got a lower bound but also a description of optimal algorithms.

Let see our first example again from the point of lower bounds. We repeat the example for the sake of convenience.

Example 4 Given an array of numbers $A[1, 2, \dots, n]$, find the maximum and the minimum of the set of the given numbers using only pairwise comparisons.

We can declare an element to be maximum only when we know that every other element has "lost" in some comparison where "lost" means that it was not the larger element. Similarly, we can declare an element to be minimum only when we know every other element has "won" some comparison. For each element we keep track of whether it has won any comparison and lost any comparison. The total of these "wins" and "losses" must add up to $2(n-1)$ for us to conclude the algorithm. In this process, we count only the first time an element wins and only the first time it loses. So at any stage, we have elements of the following types:

- W: This has element has won in some comparison but never lost
- L: This has element has lost in some comparison but never won
- WL: This has element has won in some comparison and lost in some comparison
- N: This element has not yet participated in any comparison.

The strategy for the adversary depends on the nature of the pair being compared. It always tries to give out the least amount of information consistent with previous answers. The algorithm designer is trying to increase the amount of information gathered. If the designer compares two elements both of which are of type N, we get two pieces of information because one of them wins and the other loses – both for the first time. The adversary arranges it so that in every other comparison, we get no more than one piece of information. Each element can be in only one comparison when its condition is type N. Thus, there can be no more than $\frac{n}{2}$ comparisons in any algorithm where both elements are of type N – recall these yield two pieces of information per comparison. Since each other comparison yields no more than one unit of information, we need at least $[2(n-1) - n]$ such comparisons. Thus, we need a total of $\frac{n}{2} + (n-2) = \frac{3}{2}n - 2$ comparisons. Now we describe the actual strategy of the adversary.

$\begin{array}{c} y \longrightarrow \\ x \downarrow \end{array}$	N	W	L	WL
N	2	1	1	1
W	1	1	0	≤ 1
L	1	0	1	≤ 1
WL	1	≤ 1	≤ 1	0

We want to avoid giving two pieces of information in all cases except the NN case. This forces specific answers in WN, LN, and LW cases as shown. That we can do this and maintain consistency is why this strategy works. The adversary maintains the following condition at all times: for each comparison except WL-WL combination, we choose so that either **the winner of this choice has never lost before** or **the loser of this choice has never won before** or both. This way we never give more than one piece of information at each step

except in NN. This is always possible by choosing a high enough value for the winner in the first case or a small enough value for the loser in the second case.

$\frac{y \rightarrow}{x \downarrow}$	N	W	L	WL
N		$x < y$	$x > y$	
W	$x > y$		$x > y$	
L	$x < y$	$x < y$		
WL				

All the blank squares correspond to the case when the adversary simply chooses answers consistent with previous answers in an arbitrary manner. In the WN case, choose y small enough – this can not conflict with previous answers. In the LN case, choose y large enough and this can not cause conflicts with previous answers. In the LW case, either increase the value of y or decrease the value of x .

While this gives us a lower bound, it also gives us an algorithm that achieves this bound.

- Never include an element of the type WL in any comparison; this element is not a candidate for either maximum or minimum.
- Do as many NN comparisons as possible – this is $\lfloor \frac{n}{2} \rfloor$. Thus, we give priority to NN comparisons.
- Never do LW comparisons.

Thus, we do a series of NN first; then if the number of elements is odd, do **one** NX comparison where $X \neq WL$. At this stage, there no more N type elements. Now we do either WW type or LL type depending which type has more than one element. Each comparison yields at least one unit of information and the NN ones yield two. And the bound is achieved. Within this framework, there are many possible implementations and each one is optimal.

Now we take up the second example:

Example 5 *Given an array of numbers $A[1, 2, \dots, n]$, find the minimum and the second lowest elements of the set of the given numbers using only pairwise comparisons.*

So far we have an algorithm that does $n + \lceil \lg n \rceil - 2$ comparisons. The obvious question is: "Can we do better or is this also a lower bound?" To produce a lower bound we turn to the adversary argument. For the lower bound, we may assume that elements are distinct and we do this from now on. Before we proceed further, we note that in order to know that a certain element is the second smallest, we must know which is smallest. This is so because if we declare element x as the second smallest, (and not the smallest), must have lost to some element y . In this case, y must be the smallest. Thus, we already have a lower bound of $(n - 1)$ comparisons. But we can get a better lower bound.

Adversary Strategy:

Let n_j represent the number of elements that have "lost" to j or more elements at the end in some algorithm that finds the second smallest element. The total number of comparisons must be equal to $\sum_{j=1} n_j$. We know, from the above discussion, that $n_1 = (n - 1)$. Now we will show that $n_2 \geq \lceil \lg n \rceil - 1$. Suppose at the end, the smallest element has been involved in p comparisons in which the other element has not lost to any element prior to this comparison. These and only these can be candidates for second smallest element. All other elements have "lost" at least to two elements or lost to an element that has lost to another element and are not candidates for second smallest. Hence $n_2 \geq p - 1$. So we complete the proof by showing that the smallest element must be involved in at least $\lceil \lg n \rceil$ comparisons of the above type and the adversary scenario is to find such a strategy.

So here is the adversary strategy: decide $x < y$, if x has never lost before and y has or if both have never lost before this comparison and x has won more comparisons than y as of now. In the remaining cases, make a decision consistent with previous results.

We say x supersedes y either if they are the same or if x beats y and y was unbeaten prior to this or if x supercedes z which supercedes y . An element that has won p comparisons (in which the other element has not lost to any element prior to this comparison) and lost none can supercede at most 2^p elements. But the smallest element must supercede n elements at the end. Hence it must have been involved in $\lceil \lg n \rceil$ direct comparisons. This completes the proof.