

6 Selection Algorithms

This section is on **selection** problems such as finding the k -th smallest element in a list (a.k.a. the element of **rank** k) or finding the **median** element.

Definition 6.1. *The **median** element in a list of n elements for n odd is that element for which $(n - 1)/2$ are smaller and $(n - 1)/2$ are larger. (For n even, we fudge and use the average of the middle two elements.)*

In dealing with the lower bounds for such problems, one can often make use of **adversary arguments**.

- Given a list of n elements, we can clearly find the element of rank k with $\Theta(n \log n)$ key comparisons by sorting the list.
- Given a list of n elements, we can clearly find the smallest (respectively, largest) element with $n - 1$ key comparisons by linear search.

The underlying metaphysics of an **adversary argument** is to force an algorithm to work as hard as possible.

Think of this as being the opposite of the minimax principle. At every stage, you want to provide as little information as possible, so that the algorithm makes as little progress as possible toward solving the problem.

If an adversary can force **any** algorithm to take at least $f(n)$ steps to solve a problem, then the lower bound for solving that problem is at least $f(n)$. (Remember, though, that the word “any” is absolutely crucial.)

6.1 Max and Min

Theorem 6.2. *Any algorithm to find the max and min of n elements by comparison of keys must make at least $3n/2 - 2$ comparisons in the worst case.*

Proof. First we comment that it's possible to do both max and min simultaneously in $3n/2 - 2$ comparisons, so this bound is sharp.

The totally naive, and slow, method would be to find the max first, taking $n-1$ comparisons, and then to find the min, taking $n-2$ further comparisons.

We can do better, indeed $3n/2$, by imitating the CREW PRAM algorithm and by using some comparisons twice. Compare the odd subscript elements (say) with the even subscript elements to get $n/2$ winners. Compare the winners to get $n/4$ new winners. Continue until we get the max in $n/2 + n/4 + \dots = n - 1$ steps.

Now go back and use the $n/2$ smaller elements of the pairs from the first step, and find the $n/4$ smaller pairs of these. Then the $n/8$ smaller pairs. Instead of $2n-3$ steps in the naive approach, we get to use the first set of $n/2$ comparisons twice, so that the min only takes us $n/4 + n/8 + \dots = n/2 - 1$ comparisons extra beyond those needed for the max.

The proof itself: We may assume that the keys are distinct. (That is, if we can get a lower bound assuming distinct keys, then the lower bound holds for any algorithm, since any algorithm may be presented with a distinct key problem.)

To know that key M is the max and that key m is the min it is necessary to know that every other key besides M has lost some comparison and that every other key besides m has won some comparison. We count each win

and each loss as a unit of information, and deduce that any algorithm must accumulate at least $2n - 2$ units of information.

Keys therefore can have a status:

- Status W means that a key has won at least one comparison and has never lost a comparison.
- Status WL means that a key has won at least one comparison and has lost at least one comparison.
- Status L means that a key has lost at least one comparison and has never won a comparison.
- Status N means that a key has never participated in any comparisons.

The worst case information result of comparisons is in the following table.

Status before	Worst Case	Status after	Information gain
N, N	$x > y$	W, L	2
W, N	$x > y$	W, L	1
WL, N	$x > y$	WL, L	1
L, N	$x < y$	L, W	1
W, W	$x < y$	WL, W	1
L, L	$x > y$	WL, L	1
W, L	$x > y$	W, L	0
WL, L	$x > y$	WL, L	0
W, WL	$x > y$	W, WL	0
WL, WL	whatever	WL, WL	0

Now, assuming that the table above can be applied throughout an algorithm, the lower bound is $3n/2 - 2$ for the following reason. We need $2n - 2$

units of information which we will get as

$$2n - 2 = 2 \cdot C_{N,N} + 1 \cdot C_1 + 0 \cdot C_0$$

where $C_{N,N}$ is the number of comparisons of N, N pairs from which we get 2 units of information, C_1 is the number of comparisons of pairs from which we get 1 unit of information, and C_0 is the number of comparisons of pairs from which we get no information.

We want to minimize

$$C_{N,N} + C_1 + C_0$$

subject to the above constraint. Clearly we want C_0 to be 0 and we want to maximize $C_{N,N}$. The number of such comparisons can be at most $n/2$, which means that C_1 must be $n - 2$. So the minimum number of comparisons is at least $n/2 + n - 2 = 3n/2 - 2$, as desired.

Now, must it be this bad? We need to show that the minimal progress of the table can in fact be forced to happen.

In all cases in the table except the last one, either the key chosen as the winner has never yet lost or the key chosen as the loser has never yet won.

In the former case, if we come to compare x against y , and x would be too small to be the winner, we could always make x larger so as to make it bigger than y . This wouldn't affect the status it brought to this particular comparison.

Similarly for losers.

□

6.2 The Second Largest Key

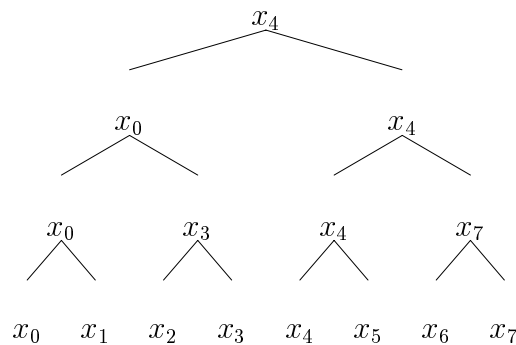
We can clearly find the second largest key by doing two passes of max and taking $(n - 1) + (n - 2)$ comparisons.

We would like to do better.

The second largest key is the key that loses a comparison only to the max. If there were a way to keep track of this, and use it, we might be able to do with fewer comparisons.

6.3 Tournaments

Arrange the n elements as leaves in a binary tree, and then have them compete in a comparison tournament. For example, we might have:



We have x_4 as the max. The only possible candidates for the **second** largest element are those that lost to x_4 , namely, x_0, x_7, x_5 . In general, then, we can find the second largest element in $n + \lg n - 2$ comparisons. Actually, $n + \lceil \lg n \rceil - 2$

Theorem 6.3. *In the worst case, an algorithm that finds the second largest element of n keys by comparison of keys must make at least $n + \lceil \lg n \rceil - 2$ comparisons.*

Proof. All we need to prove to get this lower bound is that there is an adversarial strategy so that any algorithm is required to compare the max against $\lceil \lg n \rceil$ further distinct keys.

The adversary assigns a weight $w(x)$ to every key x . Initially, $w(x) = 1$ for all x . Now, when we compare, we update as follows.

Case	Adversary reply	Updating of weights
$w(x) > w(y)$	$x > y$	$w(x) \leftarrow w(x) + w(y)$ $w(y) \leftarrow 0$
$w(x) = w(y) > 0$	$x > y$	$w(x) \leftarrow w(x) + w(y)$ $w(y) \leftarrow 0$
$w(x) < w(y)$	$y > x$	$w(y) \leftarrow w(x) + w(y)$ $w(x) \leftarrow 0$
$w(x) = w(y) = 0$	Consistent	No change

- Build trees to interpret the weights.
- The weight of the root of a tree is the number of nodes in the tree.
- The weight of a nonroot of a tree is zero.
- Comparing nonroots of trees doesn't change them.
- Combine trees when roots of the trees are compared.

So, is there some input consistent with the rules, and do the rules make any algorithm take at least $\lg n$ comparisons of the max against other keys?

- A key x has lost a comparison $\iff w(x) = 0$.
- In the first three cases, the winning key has nonzero weight, so it has never lost. The adversary can arbitrarily increase its value if necessary to force this comparison to go as in the table.

- The total sum of all weights is always n .
- On termination, only one key has nonzero weight. Otherwise, we'd have two keys that had never lost, and we could choose values so as to make the algorithm's determination of second largest incorrect.

When the algorithm stops, the key x with nonzero weight is the max.

And x has directly won against $\lceil \lg n \rceil$ distinct keys because:

Let $w_k = w(x)$ be the weight just after the k th comparison that x wins against a previously undefeated key. Then $w_k \leq 2w_{k-1}$. Thus, if x wins against K previously undefeated keys, then

$$n = w_K \leq 2^K w_0 = 2^K,$$

so $K \geq \lceil \lg n \rceil$.

Look at Table 5.2.

Implementation

If we put the elements to be examined as subscripts n through $2n - 1$ in an array, then as we run the tournament we can fill in the first half of the array as a heap with the winners.

There are also other ways to do this.

This takes n additional storage locations.

□

6.4 Finding the Median

Theorem 6.4. *Any algorithm to find, by comparison of keys, the median of n keys for n odd must do at least $3(n - 1)/2$ comparisons in the worst case.*

Proof. We assume keys are distinct.

We claim that any algorithm that outputs the median must also construct the information that relates all other keys to the median. Otherwise, the value of some key y whose relation relative to the median isn't known could be changed so as to make the output incorrect.

Definition 6.5. *A comparison involving a key x is a **crucial comparison** for x if it is the first comparison where $x > y$, for some $y \geq \text{median}$, or $x < y$, for some $y \leq \text{median}$. A comparison is **noncrucial** if $x > \text{median}$ and $y < \text{median}$ (or symmetrically).*

We have to do at least $n - 1$ crucial comparisons in order to find the median.

The adversary's strategy is as follows. Choose a particular value to be the median. The first time that a key is used in a comparison, a value will be assigned to that key. For as long as it is possible to do so, the number of keys larger than and the number smaller than the median will be balanced.

Case	Adversary strategy
N, N	One smaller, one larger
L, N	Assign a value smaller than median to the N key
N, L	Assign a value smaller than median to the N key
S, N	Assign a value larger than median to the N key
N, S	Assign a value larger than median to the N key

All of these are noncrucial.

We can force the algorithm to make $(n - 1)/2$ noncrucial comparisons using the above table.

Hence $(n - 1) + (n - 1)/2$ total.

□