CS 466/666
The Complexity of Comparison Based Problems
Lectures: July 11 - 25

Suppose we have a set of n (distinct) values and are to find the ith largest, or the elements of several ranks. Our model of computation will be that of simple comparisons between elements, and the measure of difficulty will be the number of comparisons In earlier courses you studied the problem of sorting, which is, essentially to find the elements of rank (i: i = 1 …n). Mergesort, for example, uses $n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$ comparisons, which is $n \lg n - n + 1$ if n is a power of 2. We will look at algorithms, but another key aspect of our study will be lower bounds: proving that any method must use at least "such and such" a number of comparisons. You have also seen that under the comparison model, an sorting algorithm must distinguish between all n! permutations of the input, and so must use at least lg(n!) comparisons. This lower bound holds not only in the worst case for any method, but also averaging over all possible input sequences. Using Stirlings approximation, we know that n! is "roughly" $(n/e)^n ((2n+ 1/3)\pi)^{1/2}$, so $\lg(n!) \approx n \lg n + n \lg e + (1 + \lg n + \lg \pi)/2$, or about n lg n – 1.44.. n comparisons. The gap between the upper and lower bounds is, then rather small. Indeed there are algorithms that do a bit better than mergesort and so close the gap even more.

It is rather curious that (essentially) the sorting method that takes the fewest comparisons (in the worst case) dates back to 1959. It is the method generally referred to as Ford-Johnson Merge insertion (A Tournament Problem, American Math Monthly, 66:387-389, 1959). The idea goes as follows:

Pair the elements (so n/2 comparisons, getting the largest and smallest of n/2 pairs)
Recursively sort the pairs maxima
"cleverly" (as we will outline) use binary insertion to insert the pair minima into the sorted array of maxima.

The "clever" insertion is to try to insert an element into a segment of length 1 less than a power of 2, so the number of positions into which it could fall is a power of 2. We start with the following situation

Here we note that the leftmost minimum is smaller than the leftmost maximum, so goes to its left. That gives the second leftmost minimum 3 places where it may fit, so insert the 3rd leftmost min into one of the 4 places along the "max line". Now the second left min may have 4 places on the max line where it may fit; in any case, insert it by binary search.

The method proceeds in stages by finding the leftmost element still among the minima that has a power of 2 locations where it could fit on the max line. Binary insert it, then back up inserting all the minima to the left into the max line.

We see the key trick is to try to do binary searches in the best situation, i.e. when each comparison gives the same number of spots where the element could fit. It can be shown that then number of comparisons used in the worst case is n lg n – an oscillating term. This term goes between 1.33n and 1.39n as n moves from one power of 2 to the next. The idea of discussing this method is to point to a very simple but clever tuning method, and

also to note that there are some very simple questions that have remained open for a long time.

Let's jump from this "hardest" selection problem to the easiest, namely finding the maximum of a set. The "obvious" method is

```
max = A[0]
for i  = 1 to n-1 do  if max<A[i] then max = A[i]
```

This technique uses n-1 comparisons, which is "clearly" the best we can do … but prove it. Recalling the information theoretic lower bound from sorting, we see that n elements could be the maximum so lg n is a lower bound. While true, this is hardly interesting. A better tack is to note that to know one particular element is the maximum we must "disqualify" the other n-1. An element is "disqualified" from being the maximum by being compared with another and discovered to be less than that other value. A comparison, then, gives at most one disqualification and so at least n-1 comparisons are required to discover the maximum of a set, so:

Theorem: n-1 comparisons are necessary and sufficient to find the maximum of n values.

Finding both the maximum and the minimum of a set is a bit more interesting. 2n -3 comparisons clearly suffice. However the following method does better:

1. Pair the elements and compare the elements in a pair ($\lfloor n/2 \rfloor$ comparisons, if n is odd one value is left out.
2. Find the maximum of the pair maxima (and the extra if n is odd) ($\lceil n/2 \rceil$ -1 comparisons)
3. Find the minimum of the pair minima (and the extra if n is odd) ($\lceil n/2 \rceil$ -1 comparisons)

This gives a total of $\lceil 3n/2 \rceil$ -2 comparisons. The question is whether or not we can do better. A mildly improved version of the lower bound argument for maximum answers the question..

We have to disqualify n-1 elements from being the maximum, and n-1 from being the minimum. The first "win" an element has disqualifies it from being the minimum and the first loss disqualifies it from being the maximum. This is a total of 2n – 2 disqualifications. We will give a strategy by which all comparisons can be answered in a consistent manner that leads to a lower bound on the number of comparisons required in the worst case. This is called an adversary argument.

Categorize the elements by the results of their previous comparisons:

V	no previous comparisons
W	no losses, at least 1 win
L	no wins, at least 1 loss
O	at least 1 win and 1 loss

Now the results of comparisons:

VV	one wins, one loses: WL
VW	the W wins: LW
VL	the V wins: WL

LW	the W wins: WL
LL	one wins: LO
WW	one wins: WO
OW	the W wins: WO
OL	the O wins: OL
VO	don't really care, but say V wins: WO

Note that, other than the VV case, none of these result in more than 1 disqualification: that is at most one undefeated element loses or at most one winless element wins. Furthermore there are at most $\lfloor n/2 \rfloor$ VV comparisons (as an element loses its V status in its first comparison). To solve the problem we require 2n – 2 disqualifications, so
$2n\text{-}2 \geq \text{total_compares} + \lfloor n/2 \rfloor$, so the total number of comparisons is at least $\lceil 3n/2 \rceil$ -2. Hence,

Theorem: $\lceil 3n/2 \rceil$ -2 comparisons are necessary and sufficient to find the maximum and minimum in a set of n elements.

Next consider the problem of finding the ith largest value. ($i \leq n/2$ as the other caae is symmetric). The method below is a minor modification of Quicksort:

Find i^{th} smallest of the elements in array segment A[p…r]

```
        Qselect(A,p,r,i)
        if  p = r  then return  A[p]
        Choose an element, x,  at random from  A[p..r]
        Partition  A[p..r]  so          A[q] = x
                                values  A[p..q-1]  are  < x
                                values  A[q+1..r]  are  > x
        k = q-p+1
        if  i = k  then  return  A[q]
        elseif  i < k  return Qselect(A,p,q-1,i)
        else return  Qselect(A,q+1,r,i-k)
```

The key issue is, of course, the runtime. The worst case is $\Theta(n^2)$. However, if we eliminate half the elements on each path, we use about 2n comparisons. It would seem the method should be linear in expected time assuming all input orders are equally likely. The partition value is chosen at random with all values equally likely to be taken. If we hit the right answer, we quit after a verifying scan, otherwise we recurse on one side or the other.

Let T(n) denote the maximum over all i of the expected number of comparisons to find the ith largest of n values by Qselect. We will show $T(n) \leq cn$ and then solve for c. The proof is by induction on n. It must be done simultaneously for both the odd and even cases. We do the odd case, the even is completely analogous. The base case is T(1)=0. We will assume that we always recurse on the larger side, though we do have a 1 in n chance of hitting the ith largest and having no more work to do. This gives an overestimate of the cost.

$$T(n) \le (n-1) + (2/n)\ _{i=0..\lfloor n/2\rfloor -1} 2T(\lceil n/2\rceil + i)$$
$$\le n-1\ (2/n)\ ½\ (\ c\lceil n/2\rceil + c(n-1))\lfloor n/2\rfloor$$
(inductive hypothesis & summing an arithmetic series)
$$\le n-1 + ½\ c\ 3n/2$$
$$\le n-1 + c(3n/4)$$
Now if c=4, which is valid in the base case
$$T(n) \le n-1 + 4(3n/4) < 4n$$
This proves the result; indeed the analysis is an over estimate the right answer is that c is roughly 2+2 ln(2) or about 3.44.

One can actually do better than this algorithm. To find the element of rank r in an array of n elements, we can actually do better by a more sophisticated sampling approach. Taking a simple of size s, the method due to Floyd and Rivest finds the two elements of ranks $rs/n \pm (rs/n)^{1-\epsilon}$. This gives us two values, one almost certainly larger than the desired value and one almost certainly smaller. By symmetry we can assume $r \le n/2$. Then compare all elements with the cutoff value of sample rank $rs/n(rs/n)^{1-\epsilon}$ and if necessary with the other value. Assuming the cutoff values bracket the desired value, and there are not too many other values between them, we finish off with a selection problem on a small number of values. s and   have to be chosen so that the possibility of success is very high, but the cost of checking the sample and finishing off with the values between the cutoffs is not too high. Indeed we can achieve an expected bound of n+r+o(n) comparisons. A lower bound of n+r-o(n) comparisons can be shown (though it will not be done in this course), so the technique is essentially optimal.

A more interesting, though clearly less practical method is dues to Blum, Floyd, Pratt, Rivest and Tarjan. It does selection in O(n) time in the worst case. In its simplest form it is described in the text. The constant is "impressively high", so as an exercise in tuning we give the more general version of the original authors.

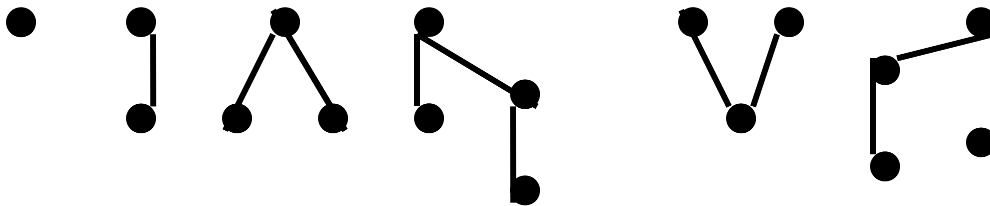
1) If n is small … sort.
2) Otherwise choose c to be an odd positive integer. Break the n values in to groups of size c and sort each group or "column"
3) Recursively find the median of the columns, call this medmed
4) We now know the relation of medmed to each column median. Use a binary search to "insert" medmed into each column. Note this is a binary search on $\lfloor c/2\rfloor$ elements performed n/c times,
5) The relation of medmed is now known with every element. Discard all the elements on the "wrong side" of medmed. At least $\lceil c/2\rceil$ elements are discarded from n/(2c) columns. That means we discard at least a quarter of the elements ... indeed a bit more because of the ceiling in the formula. Furthermore, we have a lot of partial columns in sorted order, so we reconstitute the remaining elements into sorted columns of length c and recurse starting at step 3.

We now just need a good choice of c and do some arithmetic. C=15 is a good choice. The initial sort can be done with 42 comparisons per column, so that is a total of 42/15 n. Look at the algorithm as a recursive procedure starting at step 3, so we have to enter the procedure with the columns already sorted. Consider the costs of the steps:

3) 42/225 n + T(n/15)
4) 4n/15
5) We claim we can reconstitute the sorted columns 7*17/450 n comparisons with 22/30 n elements … or we have eliminated enough more than the minimum to make up for this

So T(n) $\leq$ T(n/15) +T (22n/30) + (42/225 + 4/15 + 119/450)n. which leads to a solution of T(n)= 3.59.. n. But don't forget the initial column sorting which costs another 2.80 n. This gives a total of about 6.39 n comparisons, which doesn't sound too bad. Of course keeping track of what you are doing will take a fair bit of time and space.

Next consider lower bounds on finding the ith largest of n elements. As with the max-min problem we give an adversary argument. First we give a 3n/2 bound, which as noted earlier is the "right answer" for the expected case. We start with n singleton values, and permit pairs if singletons to be compared (clearly not caring abut the outcome). At some point we compare an element in a pair (say wlg the element at the top of a pair) with some other value. The adversary declares the pair member (if at the top of the pair) to be the winner. It also declares that this element is larger than any other till in the system and so we attribute 2 comparisons to it. It is removed from the system and its partner goes back as a singleton. So we have removed 1 element and forced 2 comparisons. Unfortunately we have to get rid of elements both "up" and also "down", so the next time we have a pair, we simply declare the bottom element to be the smallest in the system, remove it and charge it 1 comparison. This leads to an i+n-1 lower bound for finding the ith largest.
This idea can be pushed a bit farther: Rather than allowing only singletons and pairs, we permit several other structures: In the diagram below, comparisons are indicated by lines, the higher element on a line "won" the comparison. We allow the structures below, clearly the last two are symmetric cases of the middle two. The argument will be a case analysis.



The real focus is on the triple and quadruple, as any comparison involving only singletons and pairs results in one of these ... with one exception.

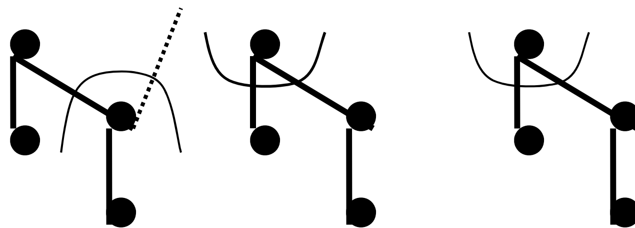
If the top of a pair is compared with the bottom of another pair ... clearly we declare the "top" to be the winner. We declare top winner to be the maximum and remove it, charging it both of its comparisons. The next time such a situation arises we declare the bottom element to be the minimum, remove it and charge two comparisons.

Moving on to the triple: A comparison with the top value allows us to send it up (charging 3); the next time we have a triple we send one of the bottom elements down, charging 1. This gets rid of 2 elements (1 up, 1 down and a cost of 4 comparisons)

Now the quadruple: A comparison involving either the top or the bottom value again leads to balanced removal at a charge of 2 per element. Comparing with the top of the quadruple lets us send it up (charge 3) and send the bottom down (charge 1). Comparing with the bottom sends it down, charging 2 and we also send the top up charging two. In either case, one element goes up, one down, and 2 singletons re-enter the system. We attribute 4 comparisons to the values removed.

The other two cases are not as easy, nor as successful. A comparison with the "middle of three" value sends it and the element below it down, charging 3 comparisons. The next two times we have this quadruple, we send the maximum up charging two comparisons. So we send 2 elements up and 2 down charging 7 comparisons (and returning some singletons and pairs to the system).

A comparison with the other element below the maximum is similar. It goes down as does the minimum of three. That is two go down charging 3 comparisons. Again we send the maximum up the next 2 times a quadruple is formed. This gives the same 2 up, 2 down, 7 comparisons outcome. The diagram illustrates this final case:



Most of our cases were leading to a 2n lower bound for medians; but the last two stall the argument at 1 ¾ n.

Theorem: At least 1 ¾ n comparisons are necessary, in the worst case, to find the median of n elements.

Note we have a lower bound on the worst case of the median problem that is higher then the expected performance of the Floyd-Rivest algorithm. This suggests the question as to how much better techniques can do on average than in the worst case.

We also discussed, and you looked at in the last assignment, a method for finding the second largest element that gets n + lglg n behavior in the worst case. See the "Matula" reference for details.