Last week we learned about heaps, priority-queues and so heapsort alg.

Today we'll prove an $\Omega(n \log n)$ lower bound for comparison based Sort and then we'll show how to break it!

Time permitting we'll start discussing ~~very useful algorithm~~ binary search trees.

Mid-term: Date is set to Oct. 25th

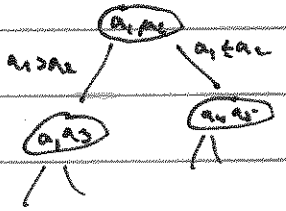
We will also see efficient algorithms for returning the i'th largest element in A.

Lower bound for comparison-based sorting.

A lower bound is an argument showing that any alg. for performing a certain task is required to make at least a certain number of operations.
Usually, proving lower bounds is a difficult task as it requires studying all possible algs.
We are going to prove a lower bound for a restricted class of algorithms for sorting. Specifically, we'll prove that algorithms that only make comparisons between array entries, cannot run faster than $\Omega(n \log n)$. Moreover, we'll prove they need to make at least this number of comparisons.

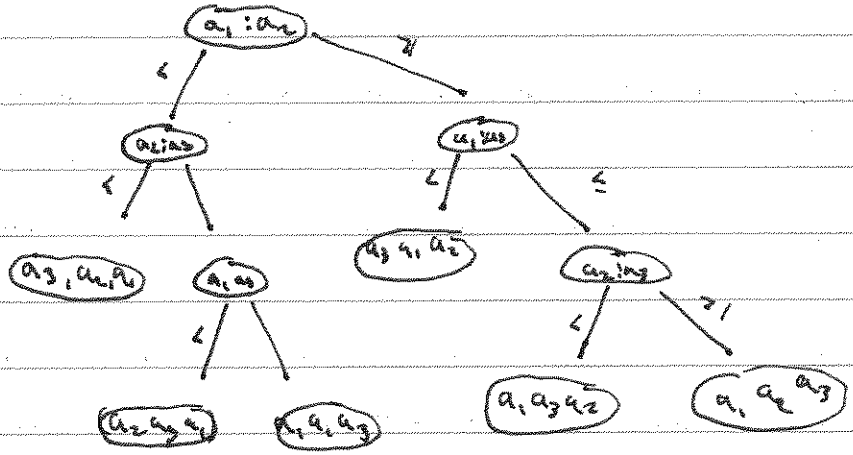
We call such algorithms comparison algorithms and they can be modeled as follows. At each step the alg. picks two elements a_i, a_j, compares them and according to the outcome ($a_i > a_j$, $a_i < a_j$, $a_i = a_j$) proceeds to the next step. At the end the alg. has to come up with a permutation that sorts the elements.

Pictorially, we can imagine the alg. as a decision-tree:



A leaf is the permutation the alg. outputs.

Here's a concrete example to sorting 3 elements



Notice that we have 6 leaves corresponding to all possible $3! = 6$ permut. Indeed, when we have n elements there must be at least $n!$ leaves as every permutation is possible

Now, what's the running time in the worst case? It is the height of the tree! Because this is a computation path that corresponds to some input and that makes that many comparisons.

How to obtain a lower bound? Notice that a depth h tree has $\leq 2^h$ leaves. Thus $n! \leq 2^h \Rightarrow h \geq \log n! = \Omega(n \lg n)$

Corollary: Heapsort and Mergesort are asymptotically optimal alg. (up to a const. factor perhaps).

5.3

The lower bound holds for comparison sorts. Can we come up with a different kind of alg. that beats the lower bound?

Counting sort is such an alg.! but it is based on the assumption that all elements of A are integers in $\{0, 1, \ldots, O(n) = k\}$

The way counting sorts works is that for every element x it counts the number of elements less than x and then outputs the elements in the right order. E.g. if we know that, say, 17 elements are smaller than x then we put x in location 18. etc.

The alg. will use two extra arrays. C will first count for each i the # of elements equal to i and then the # of elements smaller or equal to i. Thus, C will have k+1 elements C[0:k].
B will contain the sorted array.

```
Counting-Sort (A,B,C,k)
1. For i=0 to k
2.     C[i]=0
3. For j=1 to A.length
4.     C[A[j]] = C[A[j]] +1          C[i] contains # elements = i
5. For i=1 to k
6.     C[i] = C[i]+C[i-1]            " " " " ≤ i
7. For j = A.length to 1
8.     B[C[A[j]]] = A[j]             B puts A[j] in the right location
9.     C[A[j]] = C[A[j]] -1          C updated that A[j] is removed
```

5.4

Example:

A = [2 5 3 0 2 3 0 3] k = 5

First C counts C = [2 0 2 3 0 1] (0 1 2 3 4 5)

Then it adds C = [2 2 4 7 7 8]

Then we start writing B

First we go to C[A[8]] = C[A[8]] = C[3] = 7

We write in location 7 in B the value A[8] = 3

B = [(0 1 2 3 4 5 6 7) 3]

We update C[3] to 6 C = [2 2 4 6 7 8]

We then go to C[A[7]] = C[0] = 2

B = [0 3]

C = [1 2 4 6 7 8]

Then to C[A[6]] = C[3] = 6

B = [0 3 3]

C = [1 2 4 5 7 8]

Etc.

Running time:

First For loop = $\Theta(k)$

Second For loop = $\Theta(n)$

Third For loop = $\Theta(n)$ (assuming addition takes $\Theta(1)$)

Fourth loop takes $\Theta(n)$

Thus run time is $\Theta(n+k) = \Theta(n)$ when $k = \Theta(n)$.

HW: Prove counting sort is stable.

5.5

A task related to sorting is that of finding the i'th largest element in an array.

Some important examples are the largest and smallest elements and the median.

E.g. to return the ~~k~~th largest element we have the following simple alg.

```
Maximum(A)
1. max = A[1]
2. For i = 2 to A.length
3.     If A[i] > max
4.         max = A[i]
5. Return max
```

This is clearly an $O(n)$ time alg.

But what about the median? The i'th element?

Idea: similarly to quicksort we shall partition A to two parts. If the pivot is at location k then we will use recursion on the side corresponding to weather $i < k$ or $i > k$ (if $i = k$ we return the pivot).

We can use the same randomized partition alg. of quicksort and get in expectation an $O(n)$ time alg.

```
Randomized-Select(A,p,q,i)        (We want to return the ith element in
1. If p=r                                                  A[p:r]
2.     return A[p]
3. q = Randomized-Partition(A,p,r)        Same alg. from quicksort
4. k = q-p+1                              k = location of pivot in A[p..r]
5. If i=k
6.     return A[q]
7. Else if i<k
8.     return Randomized-Select(A,p,q-1,i)        smallest
9.     Else return Randomized-select(A,q+1,r,i-k)   As k elements were
                                                    removed
```

Same analysis as in quicksort shows that whp we have running time $O(n)$ (each step of partition is at most 0.9 of size and besides this constant cost)

Next we will show how to get a deterministic algorithm. The idea is to find a pivot that has $\Omega(n)$ elements smaller and larger than it.

What do we do?

Partition n to groups of 5 elements. In each group find the median. So far time $O(n)$. Now, keep only these elements and recurse. Note that at each step at least half of the $\frac{n}{5}$ medians are larger than the median of medians ☺ Thus half of the groups contain 3 elements larger than

med. of med. $\rightarrow x$ etc.

5.7 Select(A, i)

The steps of the alg. Select are:

1. Partition to $\frac{n}{5}$ grps of 5 ~~and keep the medians~~ (pretend $n = 5^k$)
2. Keep the median of each grp.
3. Using select, find the median of the $\frac{n}{5}$ medians
4. Partition the input array around that ~~the~~ element
5. depending on wheater $i = k$ (k is loc. of that element) run Select by recursion.

Running time: steps 1, 2 take $O(n)$ and so does step 4.
step 3 takes $T(\frac{n}{5})$ and step 5 takes at most $T(\frac{7n}{10})$

explanation: As we said, if x is the median then at least $3 \cdot \frac{1}{2} \cdot \frac{n}{5} = \frac{3n}{10}$ elements are smaller than it and that number larger than it.
Thus, in step 5 the size of the part we run on is $\leq \frac{7}{10}n$

$\Rightarrow \quad T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$

this seems tricky but since $\frac{n}{5} + \frac{7}{10} = \frac{9}{10} < 1$ (this is why we picked 5)
we get a linear time alg!

Assume the $O(n)$ is $\leq an$ and that $T(n) \leq cn$.
We will find c that works.

$T(n) \leq c(\frac{n}{5}) + c(\frac{7}{10}n) + an \overset{!}{\leq} cn$

Thus, we need $c(1 - \frac{7}{10} - \frac{1}{5}) \geq a$

or $c \geq 10 \cdot a$ ∎

Observe that we run select for different i's in steps 3, 5 !