Francisco Rocha
CMPS 102
1/27/15

Homework #1 (Divide and Conquer)

1. Solve in *O(log n):*

The key detail in this problem that one needs to take advantage of is that the array in which we are asked to find the largest integer is sorted but there is just a shift in the order.

Observing this key detail in this problem allows us to divide our problem in the following manner and solve it with the following algorithm:

Let k = $\left\lceil \frac{n}{2} \right\rceil$, (ie the middle element of the sorted array)
where n = size of array

Check if k is bigger than the first element in array.
- if it is ignore everything that precedes k
- else if it isn't ignore k and everything that follows it.

Repeat algorithm on new array range with smaller size until you have a two item array and choose largest of the two.

Proof of Correctness:

Assume that for the sake of contradiction that largest integer was not a part of the newly formed array after we compared k to element 1 in array.

Consider the two cases that arise in algorithm:

Case 1: *k > first element*

Since we assume the array is sorted and, since everything before k is smaller than k, if k is greater than the first element then we would discard everything that precedes k (ie elements 1 to k-1). If greatest element was not in the range k to n then the array was not sorted by increasing order to begin with and this contradicts our original assumption that the array is sorted.

Case 2: *k < first element*

Again, since we assume the array is sorted, if k is less than element 1 we would discard k and everything that follows it since all greater numbers than k would be grater than k but less than element 1. If the greatest element in the array was not in

the range 1 to k-1 then array was not sorted by increasing order to begin with and this contradicts original assumption that the array is sorted.

Pictorial examples:

Proof of runtime accurateness:

The problem gets split roughly in half during each step of recursion so we have half as much data to look through every time we make a k to element 1 comparison until we get to the trivial state which would either leave us with an array of size 2, in which we would just compare the two and choose the bigger one, or one of size one which would indicate said element is the greatest element in array.

Very similar to how Binary Search works and recursion relation would look something like:

$$T(n) = T(n/2) + 1$$

Therefor by Master Theorem runtime would be $O(\log n)$

2. Solve in *O(n log n):*

The trick to solving this problem lies completely on using the requirement of the machine you are given, which can only answer the question "are these two ballots for the same candidate, yes or no?" to your advantage to sort the ballots by grouping so that you will be able to go through them in linear time and use the comparison machine to tally the votes and decide weather a runoff is needed in this election.

Using the "comparison machine" as stated in the problem we will use a type of modification on the MergeSort algorithm to help us divide problem into sub-problems and then merge them back together in a way for us to intelligently use the "comparison machine" again on sorted ballots to check weather a runoff is needed. Algorithm is as follows:

Let $k = \lceil n/2 \rceil$.
Until each sub-problem is of size 1, split problem into sub-problems reclusively:
- i. from 1 to k
- ii. from k+1 to n

Perform merge in the following manner:
- i. insert first element on left sub-array into merged array, remove items in right sub-array equal to it and insert into merging array.

New merged array is sorted, repeat till all elements are back into a single array.

In new array, since it is sorted:

(cont. next pg)

i. from element 1 to n, make comparisons until "comparison machine returns a "no" counting number of "yeses"
ii. check if number of [yeses + 1] ≥ $\lceil n/2 \rceil$+1
   - if yes, then candidate being inspected wins, no run off needed
   - else start counting again from index of "no" returned, repeat loop.
iii. if no candidate passes [yeses + 1] ≥ $\lceil n/2 \rceil$+1 test then runoff is needed

Proof of Correctness:

The correctness to this algorithm lies in the fact that if the list is sorted into groups by the time we reach a "no" we would have counted all of the possible tallies of the votes for a certain candidate. Therefor, it is possible to simply go through the array once and count the votes for every candidate.

To show that this is correct assume, for the sake of contradiction, that at the time where "counting machine" reaches a no while tallying votes it missed a vote of the candidate it was counting. That would mean that when final array is being traversed there will be more "nos" than the number of candidates in election which means that the array was not sorted into groups to begin with which contradicts original assumption that the array is sorted into groups.

Pictorial examples:

Proof of runtime accurateness:

By the Master Theorem and knowledge we have about how merge sort works we are able to observe that the recurrence relation and runtime will be as follows:

$$T(n) = 2T(n/2) + 2n$$

Every time we recur through the sub-problem we are splitting the size of the array in to two equal parts. The merging of the two sub-problems will take n amount of work and then the iteration though the completed sorted array will once again be linear in time since we only traverse it once. Therefor the algorithm will run in

$$O(n \log n)$$

Note for part b:

The fact that we are to assume for part (a) of this problems that the size of the array is a power of 2 does not really affect the runtime of algorithm. As the problem returns to the lower parts of the recursion we are still sorting the votes into their appropriate groups by comparing two sub-arrays and using the right sub-array as the incidence of comparison. I will show how size of array doesn't not matter pictorially using a previous example and adding one more element to it:

Pictorial example(part b):

3. Solve in *O(log n)*:

The key to solving this problem lies solely on the definitions of the local minimum of a binary search tree, T. I will explicitly the local minimum of tree T as: *a node whose value is smaller than any other nodes joined to it*. With this clarifying definition the algorithm falls into place quite nicely and is as follows:

Check to see if node *v* is smaller than both of its children, starting with root *r*.
-if yes, we have found the local min of this tree
- else we choose the smaller child and recursively apply this check.
Algorithm terminates if either:
i. we find a node that is smaller than both of its children
ii. or, the recursion bottoms out and we reach a leave