Purdue University S. Hambrusch - C. Guerra
CS 381 Fall 1999

Solution Sketches to Assignment 4

1) (Graded by Cenyu Zhang)
Suppose that, instead of sorting, we wish to find the m smallest elements (in arbitrary order) in a given array of size n, where $1 \leq m \leq n$.

(i) Describe how *quicksort* can be adapted to this problem, doing less work than a complete sort.

The idea is to obtain a pivot element, partition around this element, and then recurse only on one side of the pivot, instead of two sides as done in *quicksort*. Assume all elements are distinct (so we don't have to include the case of equality). After partitioning, we know that all the elements on the left side of the pivot are smaller than the pivot, and all the elements on the right side are larger than the pivot. We use the relationship between the position of the pivot and m to decide the subproblem to recurse on:

- If the total number of elements on the left side of the pivot is equal to m, then we return those m elements - they are just what we want. No further recursive calls are made.
- If the total number of elements on the left side of the pivot is equal to $m - 1$, then we return all the elements on the left and also the pivot itself. No further recursive calls are made.
- If the total number of elements on the left side and including pivot is less than m, then we output the pivot and all elements smaller (there is no need to sort them). Then, we recurse on the subset to the right of the pivot element with a new m (we subtract from the original m the number of elements output already).
- If the total number of elements on the left side is larger than m, then we need to output only some of the element to the left of the pivot. We recurse on these element with m.

(ii) Analyze the worst-case time performance when the pivot element is chosen at random.

The worst case is when $m = 1$ and each time the partition returns the pivot which is the largest element of the array, or when $m = n$ and each time the pivot happens to be the smallest element. We then have $T(n) = T(n-1) + cn = O(n^2)$.

(iii) Analyze the worst-case time performance when the pivot element is chosen in $O(n)$ time by a given median finding algorithm.

Since finding a median takes $O(n)$ time and we recurse on a subproblem of size at most $n/2$, we have $T(n) \leq T(n/2) + cn = O(n)$.

2) (Graded by Cenyu Zhang)
The input is d sequences of elements, $S_1, S_2, \ldots, S_d$. Each sequence is already sorted and there is a total of n elements (in the d sequences). Design and analyze an $O(n \log d)$ algorithm based on Merge Sort to merge the d sequences into one sorted sequence of length n.

In the standard mergesort algorithm, the bottom of the recursion tree corresponds to single elements. We now have the leaves of the tree correspond to the given sequences, $S_1, S_2, \ldots, S_d$. Imposing a binary tree on these d sequences gives a tree of height $\log d$. Use this tree in a bottom-up way to guide the merging. At the leaves we merge two of the original sequences, creating $d/2$ new sequences. These are now paired up and merged to create $d/4$ sequences. At every level of the tree we do $O(n)$ work, where n is the total number of elements. Recall that merging two sequences of size n_i and n_j, respectively, costs $O(n_i + n_j)$ time. In the last step we merge two sequences in $O(n)$ time. Since the tree has $\log d$ levels, generating a single sorted sequence costs $O(n \log d)$ time overall.

3) (Graded by Biana Babinsky.)
Since the algorithm returns 4 as a celebrity on {1,3,4,5,6,7,8}, everyone in the set {1,3,4,5,6,7,8} knows 4, but 4 does not know anyone in this set. Since 4 fails to be a celebrity for 1,2,3,4,5,6,7,8, it is the relationship between 4 and 2 that prevents 4 from being a celebrity in the whole set. There are 3 different ways this could happen:
4 knows 2 AND 2 knows 4
4 knows 2 AND 2 does not know 4
4 does not know 2 AND 2 does not know 4
The relationship between other elements will not affect 4 being or not being a celebrity under the conditions described in the problem. Thus, the other elements

may know or not know each other.

4) (Graded by Biana Babinsky.)
i) Consider $\{1,1,2,3,4,5,6\}$. In this set, 1 is the mode, but when we run the majority algorithm, there is no majority, since no element occurs more than half the time. In general, if the mode occurs less then half the time, the majority algorithm will not find the mode.

ii) Observation: A mode that occurs at least $\frac{3n}{4}$ times, is also a majority.
Algorithm: Run the majority algorithm and find the candidate for majority. Then go through the array and count how many times the candidate occurs in the array. If it occurs more then $\frac{3n}{4}$ times, the answer is yes, otherwise, the answer is no. This is an $O(n)$ time algorithm.