**CSE 373 Analysis of Algorithms** March 11, 2014

# Solutions to HW 2

## Problem 1

[4–1] Sort the players by their values. The first $n$ players are Team 1, the rest are Team 2.

[4–2] a) Find the maximum and minimum element in the array in $O(n)$ time and output their difference.

b) $|A[n-1] - A[0]|$.

c) First sort the array. Then do a linear scan comparing the difference of every two adjacent element with a *min* counter and return *min* once you are done.

d) Do the same as above (except the sorting).

## Problem 2

[4–5] Two ways to do this:

1) *Sort and scan $O(n \log n)$:* Sort the array and scan through it keeping track of the longest run of duplicate elements seen.

2) *Hashing or using a counter array:* Scan through the array. For every element you see store it in a hash table along with its counter. At the end, scan through hash table and return the element whose counter is the largest. This is $O(n)$ expected running time.

[4–6] Sort both the arrays in $O(n \log n)$ time. You may delete the elements $> x$. Now for each element $i$ in $S_1$, do binary search to find $x - i$ in $S_2$. If it is present then you have your desired pair. The running time for binary search is $O(\log n)$ and you we do it for every element of $S_1$. Hence overall running time is $O(n \log n)$.

## Problem 3

[4–12] Build a min-heap in $O(n)$. Then do extract-min $k$ times. Thus running time is $O(n + k \log n)$.

[4–15] a) Build a tree bottom up as follows. The elements are the leaves. Compare adjacent pair of elements and move the maximum to the next level. Keep doing this till you get to one element (the maximum of all) as the root. The height of this tree if $O(\log n)$. To find the second largest element, note that for the root to have become the maximum, it must have been compared against the second largest at some point! So now we can just go down the tree following the trail of 1's and keep track of the max of the numbers it was compared to at each step. This is our second largest! The running time is $O(n)$ to build the tree and $O(\log n)$ to search the second minimum.

b) For third-largest or infact the $k^{th}$ largest element, we build the same tree as above. Follow the same logic – the second largest element must have been compared against the third largest at some point and so on. Thus for finding $k$ largest elements we take $O(n + k \log n)$ time.

For more detailed notes and code for the above, you can see `https://blogs.oracle.com/malkit/entry/finding_kth_minimum_partial_ordering`.

## Problem 4

[4–16] To find the median of an array using quicksort, we only have to recurse on one side as follows:

```
Median (A, start, end)
1. Parition the array $A$ around randomly selected pivot.
2. If $(pivot - start + 1 = n/2)$ then return $A[pivot]$.
3. Else if $(pivot - start > n/2)$ then call $Median(A, start, pivot)$.
4. Else call $Median(A, pivot, end)$.
```

Since we only recurse on one side each time, our expected running time is $O(n)$.

[4–20] Keep two counters – $x$, the index after the last negative key found so far and $y$, the first index of the positive keys found so far. The invariant is that $A[1 \ldots x-1]$ are negative and $A[y \ldots n]$ are positive. Initially $x = 1$ and $y = n + 1$. We scan the array from left to right and at each step examine $A[x]$, if it is negative we just increment $x$ and continue, if it is positive we decrement $y$ and then swap $A[x]$ with $A[y]$. At each step we either increment $x$ or decrement $y$ and hence our algorithm is in-place and $O(n)$.

## Problem 5

[4–22] We can do quicksort by partitioning the array around $k/2$ (since the elements are in the range $[1, k]$, this is the median), and recursively sort each halves similarly. Thus in $\log k$ steps, we have our sorted array, and the overall running time is $O(n \log k)$.

[4–24] We can sort the remaining $\sqrt{n}$ elements in $O(\sqrt{n} \log n)$ time and then merge the two sorted halves in $O(n)$ time. Thus the running time is $O(\sqrt{n} \log n + n) = O(\sqrt{n}\sqrt{n} + n) = O(n)$.