**CSE 373 Analysis of Algorithms**

# Solution to HW 2

**Remark:**
**1. This file is for reference. We will give credits for any reasonable answers.**
**2. The point assigned for each problem (subproblem) is in the square bracket at the end of each step.**
**3. 100 points in total.**

**Problem 1.**
$[4-1]$ Sort the players by their values.*[4]* The first $n$ players are Team 1, the rest are Team 2.*[4]*
*Remark: 8 points for this problem.*

$[4-2]$ a) Find the maximum and minimum element in the array in $O(n)$ time and output their difference.*[3]*
b) $|A[n-1]-A[0]|$.*[3]*
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.*[3]*
d) Do the same as above (except the sorting).*[3]*
*Remark: 12 points for this problem.*

**Problem 2.**
$[4-5]$ Two ways to do this:
1) Sort and scan $O(nlogn)$: 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.*[6]*
*Remark: 6 points for this problem.*

$[4-6]$ Sort both the arrays in $O(nlogn)$ time. *[4]* You may delete the elements $> x$. Now for each element $i$ in $S1$, do binary search to find $x-i$ in $S2$. If it is present then you have your desired pair. The running time for binary search is $O(logn)$ and you will do it for every element of $S1$. Hence overall running time is $O(nlogn)$.*[4]*
*Remark: 8 points for this problem.*

**Problem 3.**
$[4-12]$ Build a min-heap in $O(n)$.*[4]* Then do extract-min $k$ times. Thus running time is $O(n+klogn)$.*[4]*
*Remark: 8 points for this problem.*

$[4-13]$ 1) Finding the maximum element is $O(1)$ in both a max-heap (the root of the heap) and a sorted array (the last element in the array), so for this operation, both data structures

are equally optimal.*[2]*

2) Assuming the index of of the element is known, a deletion on a heap costs $O(logn)$ time to bubble down. A sorted array requires all elements to be updated leading to a $O(n)$ operation.*[2]*

3) A heap can be formed in $O(n)$ time. The sorted array will require a sort costing $O(nlogn)$.*[2]*

4) Finding the minimum element in a max-heap requires visiting each of the leaf nodes in the worst case, i.e. is an $O(n)$ operation. Finding the minimum element in a sorted array is an $O(1)$ operation (it's the first element), so the sorted array performs (asymptotically) better.*[2]*
*Remark: 8 points for this problem.*

$[4-14]$ Scan through all $k$ lists in any order and use the stream of elements to build a heap of $k$ elements.*[4]* Since bubble_down works in $O(logk)$ for a heap of $k$ elements, we thus solve the problem in $O(nlogk)$.*[4]*

The elementary algorithm compares the heads of each of the $k$ sorted lists to find the minimum element, puts this in the sorted list and repeats. The total time is $O(kn)$. Suppose instead that we build a heap on the head elements of each of the $k$ lists, with each element labeled as to which list it is from. The minimum element can be found and deleted in $O(logk)$ time. Further, we can insert the new head of this list in the heap in $O(logk)$ time. An alternate $O(nlogk)$ approach would be to merge the lists from as in mergesort, using a binary tree on $k$ leaves (one for each list).
*Remark: 8 points for this problem.*

$[4-15]$ 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 $O(logn)$. 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 1s 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 $O(logn)$ to search the second minimum.*[5]*

b) For third-largest or in fact 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 $k_{th}$ largest elements we take $O(n + klogn)$ time.*[5]*

For more detailed notes and code for the above, you can see `https://blogs.oracle.com/malkit/entry/finding_kth_minimum_partial_ordering`.
*Remark: 10 points for this problem.*

**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]*

2. If ($pivotstart + 1 = n/2$) then return $A[pivot]$.*[2]*
3. Else if ($pivotstart > n/2$) then call $Median(A, start, pivot)$.*[2]*
4. Else call $Median(A, pivot, end)$.*[2]*

Since we only recurse on one side each time, our expected running time is $O(n)$.
*Remark: 8 points for this problem.*

$[4-20]$ 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. *[4]* The invariant is that $A[1 \cdots x-1]$ are negative and $A[y \cdots n]$ 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 $O(n)$. *[4]*
*Remark: 8 points for this problem.*

**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)*[4]*, and recursively sort each halves similarly. Thus in $logk$ steps, we have our sorted array, and the overall running time is $O(nlogk)$. *[4]*
*Remark: 8 points for this problem.*

$[4-24]$ We can sort the remaining $\sqrt{n}$ elements in $O(\sqrt{n}logn)$ time.*[4]* And then merge the two sorted halves in $O(n)$ time. Thus the running time is $O(\sqrt{n}logn+n) = O(\sqrt{n}\sqrt{n}+n) = O(n)$.*[4]*
*Remark: 8 points for this problem.*