CSE 373 Analysis of Algorithms March 10, 2015

Solutions to HW 2

PROBLEM 1

[4–1] Sort the n players by their values, which takes $O(n \log n)$. Then simply let the first n players to be Team A, the rest to be Team B.

[4–2]

(a) Simply find the maximum element a and the minimum element b in $O(n)$ time, and the answer is $|a - b|$.
(b) The answer is $|A_{n-1} - A_0|$.
(c) First sort the array. Then do a linear scan, and return the minimum difference of every two adjacent element.
(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 the whole array. For each element, 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 which are greater than x. Now for each element k in S_1, do binary search to find $x - k$ in S_2. If it is present then you have the desired pair. The running time for binary search is $O(\log n)$ and we do it for every element of S_1, which is $O(n \log n)$ in total. Hence overall running time is $O(n \log n)$.

PROBLEM 3

[4–12] Build a min-heap in $O(n)$. Then do *extract-min* for k times to get the k smallest elements. Thus, total running time is $O(n + k \log n)$.

[4–13]

(a) Both max-heap and sorted array support finding maximum in $O(1)$ time.
(b) Deletion takes $O(\log n)$ in max-heap, and $O(n)$ in sorted array. Thus max-heap is better.
(c) It takes $O(n)$ to build a max-heap, and $O(n \log n)$ to build a sorted array. Thus max-heap is better.

(d) It takes $O(n)$ to find the minimum element in a max-heap, because in the worst case you need to check every leaf node. For a sorted array, it takes only $O(1)$, thus sorted array is better.

[4–14] The basic idea is: we keep an array of k pointers, each pointing to the start of the corresponding sorted list initially. Then for each step, we pick the smallest element among the k current elements , and move forward the corresponding pointer. It takes $O(k)$ time to pick the smallest element, and we have n elements in total, thus now we have an $O(nk)$ algorithm. Then, we speed up this algorithm by maintaining a min-heap of those k elements: for each step, we simply pick the minimum element in the heap and move forward the corresponding pointer. It takes $O(\log k)$ time to pick and delete the smallest element and insert the next element, and we have n elements in total. This gives us an $O(n \log k)$ algorithm.

[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 – this process is like a tournament. The height of this tree is $O(\log n)$. To find the second largest element, note that for the root to 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 maximum of the numbers it was compared to at each step. This is our second largest element. See the picture below to understand this algorithm better: (Note that this figure is for finding the minimum, but the algorithm is the same)

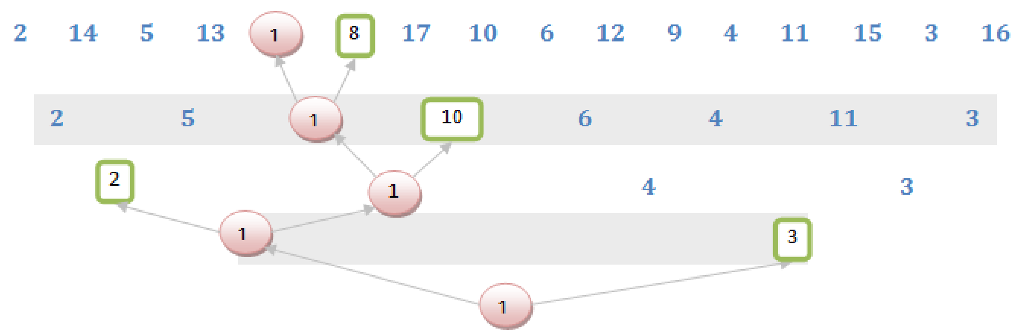

Figure 1: Finding minimum and second minimum using Tournament method

The count of comparisons is $n/2 + n/4 + \cdots + 1 = n$ for building the tree, and $\log n$ for finding the second-largest. Thus, we only need $n + \log n$ comparisons in total.

(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 or largest element must have been compared against the third largest at some point and so on. otherwise, the third largest element should "win" all the tournaments because it's larger than all the other elements, and this leads to contraction. Thus, we can check in the tree which elements have been compared with the largest or the second largest elemnt, and pick the largest among these elements. Thus, finding the kth largest elements takes $O(n + k \log n)$ time. And in the process, we must determine which key is largest and second-largest.

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 do the recursion on one side as follows:

Median (A, start, end)

1. (1) Partition the array A around randomly selected pivot. This takes $O(n)$ for n elements.
2. (2) If $(pivot - start + 1 == n/2)$ then return $A[pivot]$.
3. (3) Else if $(pivot - start > n/2)$ then the median is in the left side of the array. Simply call $Median(A, start, pivot)$.
4. (4) Else call $Median(A, pivot, end)$.

With a good pivot, for each step we roughly reduce the number of elements by half. Thus, the total time for finding the median is $n + n/2 + n/4 + \cdots + 1 = 2n = O(n)$.

[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 then just increase x by 1 and continue, if it is positive then decrease y by 1 and then swap $A[x]$ with $A[y]$. At each step we either increase x or decrease y, hence our algorithm is in-place and runs in $O(n)$ time.

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 each step takes no more than $O(n)$. Thus, the overall running time is $O(n \log k)$.

[4–24] Two steps:

[1] Sort the last $\sqrt{n}$ element using any sorting algorithm. Let's say we are using bubble sort, and this step takes $O(n)$.

[2] Merge the first $n - \sqrt{n}$ elements with the last $\sqrt{n}$ elements using the same method as merge sort. This step also takes $O(n)$. In total the time complexity is $O(n) = o(nlogn)$.