CSE431/531 Problem Set 2

Solution

Dayu He

Problem 1

The approach is very much as same as shown in the lecture slides. Here are two sample analysis with group size of 3 and 7. For the group size of 4,6,9 and 11, only result will be shown.

For group size is 3:
When we partition the elements into group of size 3, at least half of the medians are greater than the median-of-medians *x*. Thus at least half of the $\left\lceil \frac{n}{3} \right\rceil$ groups contribute 2 elements that are greater than *x*, except for the group that has less than 3 elements if 3 does not divide n exactly, and the group containing x itself. So the number of elements greater than x is at least $2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil - 2\right) \geq \frac{n}{3} - 4$. Similarly, the number of elements that are less than *x* is at least $\frac{n}{3} - k$ for a constant *k*. So in the worst case SELECT is called recursively on at most $\frac{2}{3}n + 4$ elements. We can therefore obtain the recurrence: $T(n) \leq T\left(\left\lceil \frac{n}{3} \right\rceil\right) + T\left(\frac{2n}{3} + 4\right) + O(n)$.

Note that the sum of the inputs of the two recursive call is: $\left\lceil \frac{n}{3} \right\rceil + \left(\frac{2n}{3} + 4\right) > n$. So the run time of the algorithm cannot be $O(n)$.

For group size is 7:
When we partition the elements into group of size 7, at least half of the medians are greater than the median-of-medians *x*. Thus at least half of the $\left\lceil \frac{n}{7} \right\rceil$ groups contribute 4 elements that are greater than *x*, except for the group that has less than 7 elements if 7 does not divide n exactly, and the group containing x itself. So the number of elements greater than x is at least $4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7} \right\rceil \right\rceil - 2\right) \geq \frac{2n}{7} - 8$. Similarly, the number of elements that are less than *x* is at least $\frac{n}{7} - 8$. So in the worst case SELECT is called recursively on at most $\frac{5}{7}n + 8$ elements. We can therefore obtain the recurrence:

$$T(n) \le \begin{cases} O(1) & \text{if } n < 126 \\ T\left(\left\lceil \frac{n}{7} \right\rceil\right) + T\left(\frac{5n}{7} + 8\right) + O(n) & \text{if } n \ge 126 \end{cases}$$

Since the sum of the inputs of the two recursive calls is $\left\lceil \frac{n}{7} \right\rceil + \left(\frac{5n}{7} + 8\right) \approx \frac{6n}{7} < n$, we can show the running time is linear as shown in the slides. The proof is by induction.

Assume that $T(n) \le cn$.

$$\begin{aligned} T(n) &\le c\lceil n/7 \rceil + \mathrm{c}(5n/7 + 8) + an \\ &\le cn/7 + c + 5cn/7 + 8c + an \\ &= cn + (-cn/7 + 9c + an) \end{aligned}$$

which is at most *cn* if $-cn/7 + 9c + an \le 0$. This is equivalent to

$$c \ge 7an/(n - 63)\, , when\ n > 63$$

Because we assume that $n \ge 126$, we have $n/(n - 63) \le 2$. So choosing $c \ge 14a$ will make above inequality true. Thus $T(n) \le cn$ for all *n* by induction. Therefore the algorithm still runs in linear time.

For case of group 4, $T\left(\left\lceil \frac{n}{4} \right\rceil\right) + T\left(\frac{5n}{8} + 6\right) + O(n)$, non-linear.

For case of group 6, $T\left(\left\lceil \frac{n}{6} \right\rceil\right) + T\left(\frac{2n}{3} + 8\right) + O(n)$, linear.

For case of group 9, $T\left(\left\lceil \frac{n}{9} \right\rceil\right) + T\left(\frac{13n}{18} + 10\right) + O(n)$, linear.

For case of group 11, $T\left(\left\lceil \frac{n}{11} \right\rceil\right) + T\left(\frac{8n}{11} + 12\right) + O(n)$, linear.

Problem 2

Idea: (to simplify the illustration, we assume that x_i and y_j are distinctive)

Step1: Sort the points in *P* in decreasing *x*-coordinates by using an *O*(*nlogn*) sorting algorithm. (i.e. MergeSort). And put the first point p_1 of the new sorted array $\acute{P}$ into the set of maximum point called *Max(P)*. (The reason p_1 must be in this set is that it has the biggest x_i of all the point, thus it cannot be dominated by other points)

Step2: Scan $\acute{P}$ from $p_1\ to\ p_n$, for each point p_i $(2 \le i \le n)$, we compare p_{i_y} with the last point of *Max(P)*. If p_{i_y} is bigger, then $p_i \in Max(P)$, else discard p_i. (The reason we only compare p_{i_y} with the last point in *Max(P)* is: in our way, the points in *Max(P)* is in increasing y-coordinates. It is easy to prove this!). Since we have just one scan for all the points, the run time is linear in *O(n)*.

Pseudo-code:

Maximum-Point-ALG($\acute{P}$, $Max(P)$)

```
1   $Max(P) \rightarrow \mathrm{Append}(\acute{P}[1])$
2   for $i \leftarrow 2$ to $n$
```

```
3   do if $\acute{P}_{i_y} \le Max(P) \to getlast()_y$
4       then $Max(P) \to Append(\acute{P}_i)$
```

In 3D space case, instead of just comparing the p_{i_y} to form $Max(P)_1$, we also compare p_{i_z} to form $Max(P)_2$. Then in step3, we combine these two subset into our final result *Max(P).* Since both comparing step take O(n) and the final combine step also takes O(n). So, the whole run time would also be O(nlogn).

Problem 3

(a) For the given weight set $\{\omega_1, \omega_2, .. \omega_n\}$. If $\omega_1 \ge \frac{1}{2}$, then we have $\sum_{i<1} \omega_i = 0 < \frac{1}{2}$ and $\sum_{i=2}^{n} \omega_i = 1 - \omega_1 < \frac{1}{2}$. Thus, by the definition we have ω_1 as our weighted median. If $\omega_1 < \frac{1}{2}$, then there exist a number *k* where $k \in [1, n]$, makes $\sum_{i=1}^{k-1} w_i < \frac{1}{2}$ and $\sum_{i=1}^{k} w_i \ge \frac{1}{2}$. Since $\sum_{i=1}^{n} w_i = 1$, then $\sum_{i=k+1}^{n} w_i = \sum_{i=1}^{n} w_i - \sum_{i=1}^{k} w_i = 1 - \sum_{i=1}^{k} w_i < 1 - \frac{1}{2} = \frac{1}{2}$. Thus, by the definition, we have ω_k as our weighted median. So the statement is proved.

(b) We could modify the binary search algorithm, and let it works for our problem. First the median a_k of $\{a_1, a_2, .. a_n\}$ is checked to see if it is the weighted median (by calculating the weight of left and right part of the array). If so, it is returned. If not, this must be because one of the two inequalities defining the weighted median has been violated. If the left part violate, then the weighted median must be further to the left side of the array, and then left part of the array has to check recursively for the weighted median. Similarly, if the right part violate the inequality, then the right part of the array has to check re cursively for the weighted median.

Pseudo-code:

Weighted-Median(A, start, end, sumleft, sumright)

```
1   $i \leftarrow \lceil (end - start)/2 \rceil$
2   $m \leftarrow SELECT(A, start, end, i)$
3   for ($j \leftarrow start\ to\ i - 1$)
4       $sumleft \leftarrow sumleft + \omega_j$
5   for ($j \leftarrow i + 1\ to\ end$)
6       $sumright \leftarrow sumright + \omega_j$
7   if $\left(sumleft \le \frac{1}{2}\ AND\ sumright \le \frac{1}{2}\right)$
8       return A[i]
9   if $\left(sumleft > \frac{1}{2}\right)$
10      return Weighted-Median(A,start,i,0,sumright)
```

```
11  **return** *Weighted-Median(A, i,end,sumleft,0)*
```

Run Time:

The run time for line 1 to line 8 is $O(n)$.

For the recursion part (line 9 & line 10, or line 9 & line 11), each time we will deal with half size of the original array.

So, for the worst case, the run time of this algorithm is

$$T(n) = T\left(\frac{n}{2}\right) + O(n)$$

By using Master Theorem, we can get $T(n) = \theta[n]$.

Problem 4

This is a typical 3sum problem.

Firstly, sort the array $S = \{a_1, a_2, .. a_n\}$ into a new array S'. Then for each element $s'_i \in S'$, find a pair from the elements after s'_i, with the sum of that pair equals to $B - s'_i$ (details of finding the pair will be given in the pseudo-code blow). Do the iteration from s'_1 *to* s'_{n-2} until you find the final solution.

Pseudo-code:

```
for i=1 to n-2 do
    a = $s'_i$;
    k = i+1;            #Set left pointer at the first element on the right of $s'_i$#
    l = n -1;           #Set right pointer at the last element in S'#
    while (k<l) do
        b = $s'_k$;
        c = $s'_l$;
        if (a+b+c == B) then
            output a, b, c;
            exit;
        else if (a+b+c > B) then
            l = l - 1;          #in this case, move right pointer one step to left#
        else
            k = k + 1;          #in this case, move left pointer one step to right#
        end
    end
end
```

Run Time:

In the inner loop of this algorithm, we only need O(n) time to find the matched pair. And on the outer loop of the algorithm, we will need at most n-2 iterations.

So the total run time of this algorithm is $\mathrm{O}(\mathrm{n}^2)$.

Problem 5

Key: Modify the MergeSort algorithm (which we will call FindPair) as follows. In addition to sort the input array, it also returns the number of inversions in A:

Pseudo-code:

***FindPair*(*A*[*p*, *r*])**

```
1  if $r == p$
2    then return 0 and STOP
3  $q = (p+r)/2$;
4  $N_1 \leftarrow FindPair(A[p,q])$
5  $N_2 \leftarrow FindPair(A[q+1,r])$
6  $N_3 \leftarrow ModifiedMerge(A, p, q, r)$
7  return $\{N_1 + N_2 + N_3\}$
```

The *ModifiedMerge* function called within the *FindPair* algorithm is a modification of the original **Merge** function within **MergeSort.** Here is the description:

- When $ModifiedMerge(A, \mathrm{p}, \mathrm{q}, \mathrm{r})$ is called, the left sub-array *A1=A[p..q]* and the right sub-array *A2=A[q+1,r]* have been sorted. The *ModifiedMerge* will merge *A1* and *A2* into a single sorted array. It also finds and returns the number of inversions(*A[i]*,*A[j]*) where *A[i]* is in *A1* and *A[j]* is in *A2*.
- During the merging, for each entry *A[j]* ($\mathrm{q}+1 \le \mathrm{j} \le \mathrm{r}$) in *A2,* we find the index i_j (in the range $l \le i_j \le q)$ such that $A[i_j]$ is the largest number in *A1* that is smaller than *A[j]*. Namely $A[i_j] \le A[j] < A[i_j+1]$. In other words, when we merge *A1* and *A2*, *A[j]* will be inserted right after $A[i_j]$.
- Since all elements in $A[i_j+1..q]$ are larger than *A[j]*, the number of inversions that involves *A[j]* is then $x_j = q - i_j$.
- Thus the total number of inversions is: $X = x_{q+1} + x_{q+2} + .. x_r$. This sum can be accumulated during the merging process. The function returns *X* to FindPair.

Run Time:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n > 1 \end{cases}$$

By using Master Theorem (*a=2, b=2*), we have $T(n) = \theta(nlogn)$

Problem 6

$$as + bu = p_3 + p_7 - p_5 + p_2$$
$$at + bv = p_4 + p_5$$
$$cs + du = p_6 + p_7$$
$$ct + dv = p_3 + p_4 - p_6 - p_1$$