Solution to Homework 2

Yangwei Liu

October 16, 2014

1 Problem 1

The approach is very much the same as shown in the lecture slides. Below are solutions to the problem with group sizes 3 and 7. For problems with group sizes 4,6,9 and 11, only conclusions will be shown.

1. When group size is 3:

   When we partition the elements into group of 3, at least half of the medians are greater than the median-of-medians x. Thus at least half of the $\lceil \frac{n}{3} \rceil$ groups contribute 2 elements that are greater than x, except for the group that has less than 3 elements and the group containing x itself. So the number of elements greater than x is at least $2(\lceil \frac{1}{2} \lceil \frac{n}{3} \rceil \rceil - 2) \geq \frac{n}{3} - 4$. Similarly, the number of elements that are less than x is at least $\frac{n}{3} - k$ for some constant k. So in the worst case SELECT is called recursively on at most $\frac{2}{3}n + 4$ elements. So we have the recurrence:

   $$T(n) \leq T(\lceil \frac{n}{3} \rceil) + T(\frac{2n}{3} + 4) + O(n)$$

   Note that $\lceil \frac{n}{3} \rceil + (\frac{2n}{3} + 4) > n$. So the running time is super linear.

2. When group size is 7:

   The analysis is similar, except that the number of elements greater than x is at least $4(\lceil \frac{1}{2} \lceil \frac{n}{7} \rceil \rceil - 2) \geq \frac{2n}{7} - 8$, and the number of elements that are less than x is at least $\frac{n}{7} - 8$. So in the worst case SELECT is called on at most $\frac{5}{7}n + 8$ elements. So we have the recurrence:

   $$T(n) = T(\lceil \frac{n}{7} \rceil) + T(\frac{5n}{7} + 8) + O(n), \ \textit{where } n \geq 126$$

   Using a similar proof in the slides, we can show $T(n)$ is linear.

3. When group size is 4: non-linear

4. When group size is 6: linear

5. When group size is 9: linear

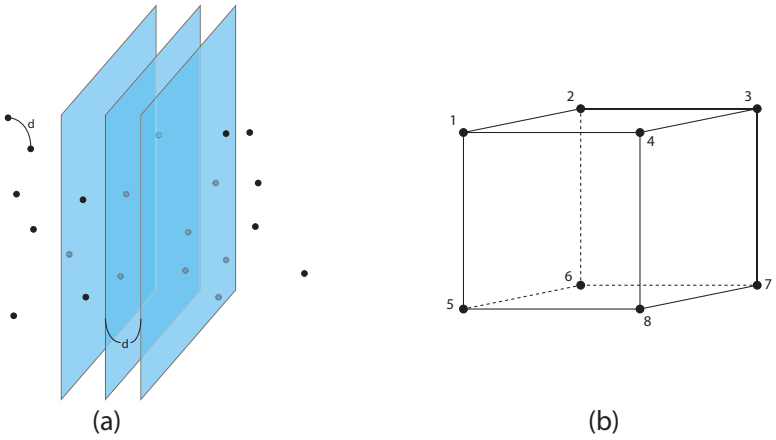

Figure 1: (a). Vertical strip with width 2δ; (b). There exist at most 8 points in a cube where each pair of points has distance at least δ

2 Problem 2

The 3d problem is very similar to the 2d one.

2.1 Divide :

Here we use a plane, instead of a line, to partition the points into two subsets of equal size. Recursively get the solution to the two subproblems, namely δ_a and δ_b, and let $\delta = min(\delta_a, \delta_b)$.

2.2 Conquer :

The closest pair distance is either δ, which is the solution to one of the subproblem, or $\delta^{'} < \delta$, with its two endpoints in different sides of the dividing plane. Again we draw two planes parallel to the dividing plane, forming a strip with width 2δ. The reason for the strip to have a width of 2δ is exactly the same as that in the 2d problem. Then we project all points within the 2δ strip to the dividing plane, and using the 2d closest pair algorithm to get the closest pair with their original 3d distances as the distance function. This step takes $O(n \log n)$ time. The recurrence for the running time is as follows:

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

the running time would be $O(n \log^2 n)$.

The correctness of this algorithm needs careful analysis. First, after the projection, within a δ square on the dividing plane, there can be at most constant number of points. This is because there can be at most constance number of points in the original δ width cube, so after projection the number of points

within that region remains the same. Now using a modified version of the 2d closest pair algorithm with the constant changed from 7 to a larger one suitable for this problem, and using the 3d distance as distance function, we can get the actual closest pair within the 2δ strip in $O(n \log n)$ time.

There are better algorithms for this problem with running time $O(n \log n)$, but they need much more complicated analysis. If you give an algorithm with running time equal to or better than the one given above, you should receive full marks.

3 Problem 3

3.1 a

When a_k goes from the smallest a_i to the largest a_i, the sum $\sum_{i:a_i<a_k} w_i$ goes from w_{i_1} to $w_{i_1} + \cdot + w_{i_n}$, where w_{i_j} is the weight of the j'th smallest a. If $w_{i_1} \geq \frac{1}{2}$, we have $\sum_{i:a_i<a_{i_1}} w_i = 0 < \frac{1}{2}$, $\sum_{i:a_i>a_{i_1}} w_i = 1 - w_{i_1} \leq \frac{1}{2}$, so the weighted median is a_{i_1}; If $w_{i_1} \leq \frac{1}{2}$, there exists an $i_j \in [1, n]$ such that $\sum_{i:a_i<a_{i_j}} w_i < \frac{1}{2}$ and $\sum_{i:a_i<a_{i_{j+1}}} w_i \geq \frac{1}{2}$. So now we have $a_{i_{j+1}}$ as the weighted median. So the weighted median exists.

3.2 b

Here we use the SELECT algorithm to find the weighted median. Firstly we use SELECT to find the median of $\{a_1, \cdot, a_n\}$. Suppose it's a_k. If a_k satisfies the definition of the weighted median, return a_k. Otherwise one of the two inequalities has to be violated. Suppose it's the first inequality violated, i.e. $\sum_{i:a_i<a_{i_1}} w_i \geq \frac{1}{2}$, then we try the median of the smaller half of a_i. Similarly, if the second inequality is violated, we try the median of the larger half. The pseudocode of the algorithm is as follows:

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 $\quad sumleft \leftarrow sumleft + \omega_j$

5 **for** $(j \leftarrow i + 1\ to\ end)$

6 $\quad sumright \leftarrow sumright + \omega_j$

7 **if** $\left(sumleft \leq \frac{1}{2}\ AND\ sumright \leq \frac{1}{2}\right)$

8 $\quad$ **return** *A[i]*

9 **if** $\left(sumleft > \frac{1}{2}\right)$

10 $\quad$ **return** *Weighted-Median(A,start,i,0,sumright)*

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

The running time for lines 1-8 is $O(n)$, and for the recurrence part, since each time we deal with half the size, so we have the following recurrence:

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

So $T(n) = \Theta(n)$

4 Problem 4

First sort the array into $S^{'}$. For each element $a_i \in S^{'}$, find a pair after a_i so that their sum is $B - a_i$. The pseudocode is as follows:

```
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
```

The correctness of the algorithm is obvious. For the running time, sorting takes $O(n \log n)$ time, and each inner loop takes $O(n)$ time, and there are $O(n)$ inner loops. So the total running time is $O(n^2)$.

5 Problem 5

Here we modify the mergesort algorithm to find the number of inversions. The pseudocode is as follows:

FindPair(A[p, r])

1 **if** $r == p$

2 **then return** 0 and STOP

3 $q = (p + r)/2;$

4 $\mathrm{N}_1 \leftarrow FindPair(A[p, q])$

5 $\mathrm{N}_2 \leftarrow FindPair(A[q + 1, r])$

6 $\mathrm{N}_3 \leftarrow ModifiedMerge(A, \mathrm{p}, \mathrm{q}, \mathrm{r})$

7 **return** $\{\mathrm{N}_1 + \mathrm{N}_2 + \mathrm{N}_3\}$

The ModifiedMerge function is a modified version of the original Merge function. Below is its 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]* ($\mathrm{q}+1 \leq \mathrm{j} \leq \mathrm{r}$) in *A2,* we find the index i_j (in the range $l \leq i_j \leq q$) such that $A[i_j]$ is the largest number in *A1* that is smaller than *A[j]*. Namely $A[i_j] \leq A[j] < A[i_j + 1]$. 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.

The running time is exactly the same as merge sort, which is $\Theta(n \log n)$.

6 Problem 6

Solving the equations, we get:

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