

Solutions - Problem Set #2

Alice Bob (uni). Collaborators: A. Turing. - `uni@columbia.edu`

September 26, 2019

1 Problem 1

1. The median x of the elements x_1, x_2, \dots, x_n , is an element $x = x_k$ satisfying $\{x_i : 1 \leq i \leq n \text{ and } x_i < x\} \leq n/2$ and $\{x_i : 1 \leq i \leq n \text{ and } x_i > x\} \leq n/2$. If each element x_i is assigned a weight $w_i = 1/n$, then we get

$$\begin{aligned}\sum_{x_i < x} w_i &= \sum_{x_i < x} \frac{1}{n} \\ &= \frac{1}{n} \cdot \sum_{x_i < x} 1 \\ &= \frac{1}{n} \cdot \{x_i : 1 \leq i \leq n \text{ and } x_i < x\} \\ &\leq \frac{1}{n} \cdot \frac{n}{2} \\ &= \frac{1}{2},\end{aligned}$$

and

$$\begin{aligned}\sum_{x_i > x} w_i &= \sum_{x_i > x} \frac{1}{n} \\ &= \frac{1}{n} \cdot \sum_{x_i > x} 1 \\ &= \frac{1}{n} \cdot \{x_i : 1 \leq i \leq n \text{ and } x_i > x\} \\ &\leq \frac{1}{n} \cdot \frac{n}{2} \\ &= \frac{1}{2},\end{aligned}$$

which proves that x is also the weighted median of x_1, x_2, \dots, x_n with weights $w_i = 1/n$, for $i = 1, 2, \dots, n$.

2. We first sort the n elements into increasing order by x_i values. Then we scan the array of sorted x_i 's, starting with the smallest element and accumulating weights as we scan, until the total exceeds $1/2$. The last element, say x_k , whose weight caused the total to exceed $1/2$, is the weighted median. Notice that the total weight of all elements smaller than x_k is less than $1/2$, because x_k was the first element that caused the total weight to exceed $1/2$. Similarly, the total weight of all elements larger than x_k is also less than $1/2$, because the total weight of all the other elements exceeds $1/2$.

The sorting phase can be done in $O(n \lg n)$ worst-case time (using merge sort or heapsort), and the scanning phase takes $O(n)$ time. The total running time in the worst case, therefore, is $O(n \lg n)$.

3. We find the weighted median in $\Theta(n)$ worst-case time using the $\Theta(n)$ worst-case median algorithm in (Although the first paragraph of the section only claims an $O(n)$ upper bound, it is easy to see that the more precise running time of $\Theta(n)$ applies as well, since steps 1, 2, and 4 of SELECT actually take $\Theta(n)$ time.)

The weighted-median algorithm works as follows. If $n \leq 2$, we just return the brute-force solution. Otherwise, we proceed as follows. We find the actual median x_k of the n elements and then partition around it. We then compute the total weights of the two halves. If the weights of the two halves are each strictly less than $1/2$, then the weighted median is x_k . Otherwise, the weighted median should be in the half with total weight exceeding $1/2$. The total weight of the “light” half is lumped into the weight of x_k , and the search continues within the half that weighs more than $1/2$. Here's pseudocode, which takes as input a set $X = \{x_1, x_2, \dots, x_n\}$:

```

Weighted-Median( $X$ )
1  if  $n == 1$ 
2      return  $x_1$ 
3  elseif  $n == 2$ 
4      if  $w_1 \geq w_2$ 
5          return  $x_1$ 
6      else return  $x_2$ 
7  else find the median  $x_k$  of  $X = \{x_1, x_2, \dots, x_n\}$ 
8      partition the set  $X$  around  $x_k$ 
9      compute  $W_L = \sum_{x_i < x_k} w_i$  and  $W_G = \sum_{x_i > x_k} w_i$ 
10     if  $W_L < 1/2$  and  $W_G < 1/2$ 
11         return  $x_k$ 
12     elseif  $W_L > 1/2$ 
13          $w_k = w_k + W_G$ 
14          $X' = \{x_i \in X : x_i \leq x_k\}$ 
15         return WEIGHTED-MEDIAN( $X'$ )
16     else  $w_k = w_k + W_L$ 
17          $X' = \{x_i \in X : x_i \geq x_k\}$ 
18         return WEIGHTED-MEDIAN( $X'$ )

```

The recurrence for the worst-case running time of WEIGHTED-MEDIAN is $T(n) = T(n/2 + 1) + \Theta(n)$, since there is at most one recursive call on half the number of elements, plus the median element x_k , and all the work preceding the recursive call takes $\Theta(n)$ time. The solution of the recurrence is $T(n) = \Theta(n)$.

4. Let the n points be denoted by their coordinates x_1, x_2, \dots, x_n , let the corresponding weights be w_1, w_2, \dots, w_n , and let $x = x_k$ be the weighted median. For any point p , let $f(p) = \sum_{i=1}^n w_i |p - x_i|$; we want to find a point p such that $f(p)$ is minimum. Let y be any point (real number) other than x . We show the optimality of the weighted median x by showing that $f(y) - f(x) \geq 0$. We examine separately the cases in which $y > x$ and $x > y$. For any x and y , we have

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^n w_i |y - x_i| - \sum_{i=1}^n w_i |x - x_i| \\ &= \sum_{i=1}^n w_i (|y - x_i| - |x - x_i|). \end{aligned}$$

When $y > x$, we bound the quantity $|y - x_i| - |x - x_i|$ from below by examining three cases:

- (a) $x < y \leq x_i$: Here, $|x - y| + |y - x_i| = |x - x_i|$ and $|x - y| = y - x$, which imply that $|y - x_i| - |x - x_i| = -|x - y| = x - y$.

(b) $x < x_i \leq y$: Here, $|y - x_i| \geq 0$ and $|x_i - x| \leq y - x$, which imply that $|y - x_i| - |x - x_i| \geq -(y - x) = x - y$.

(c) $x_i \leq x < y$: Here, $|x - x_i| + |y - x| = |y - x_i|$ and $|y - x| = y - x$, which imply that $|y - x_i| - |x - x_i| = |y - x| = y - x$.

Separating out the first two cases, in which $x < x_i$, from the third case, in which $x \geq x_i$, we get

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^n w_i(|y - x_i| - |x - x_i|) \\ &\geq \sum_{x < x_i} w_i(x - y) + \sum_{x \geq x_i} w_i(y - x) \\ &= (y - x) \left(\sum_{x \geq x_i} w_i - \sum_{x < x_i} w_i \right). \end{aligned}$$

The property that $\sum_{x_i < x} w_i < 1/2$ implies that $\sum_{x \geq x_i} w_i \geq 1/2$. This fact, combined with $y - x > 0$ and $\sum_{x < x_i} w_i \leq 1/2$, yields that $f(y) - f(x) \geq 0$.

When $x > y$, we again bound the quantity $|y - x_i| - |x - x_i|$ from below by examining three cases:

(a) $x_i \leq y < x$: Here, $|y - x_i| + |x - y| = |x - x_i|$ and $|x - y| = x - y$, which imply that $|y - x_i| - |x - x_i| = -|x - y| = y - x$.

(b) $y \leq x_i < x$: Here, $|y - x_i| \geq 0$ and $|x - x_i| \leq x - y$, which imply that $|y - x_i| - |x - x_i| \geq -(x - y) = y - x$.

(c) $y < x \leq x_i$. Here, $|x - y| + |x - x_i| = |y - x_i|$ and $|x - y| = x - y$, which imply that $|y - x_i| - |x - x_i| = |x - y| = x - y$.

Separating out the first two cases, in which $x > x_i$, from the third case, in which $x \leq x_i$, we get

$$\begin{aligned} f(y) - f(x) &= \sum_{i=1}^n w_i(|y - x_i| - |x - x_i|) \\ &\geq \sum_{x > x_i} w_i(y - x) + \sum_{x \leq x_i} w_i(x - y) \\ &= (x - y) \left(\sum_{x \leq x_i} w_i - \sum_{x > x_i} w_i \right). \end{aligned}$$

The property that $\sum_{x_i > x} w_i \leq 1/2$ implies that $\sum_{x \leq x_i} w_i > 1/2$. This fact, combined with $x - y > 0$ and $\sum_{x > x_i} w_i < 1/2$, yields that $f(y) - f(x) > 0$.

5. We are given n 2-dimensional points p_1, p_2, \dots, p_n , where each p_i is a pair of real numbers $p_i = (x_i, y_i)$, and positive weights w_1, w_2, \dots, w_n . The goal is to find a point $p = (x, y)$ that minimizes the sum

$$f(x, y) = \sum_{i=1}^n w_i (|x - x_i| + |y - y_i|) .$$

We can express the cost function of the two variables, $f(x, y)$, as the sum of two functions of one variable each: $f(x, y) = g(x) + h(y)$, where $g(x) = \sum_{i=1}^n w_i |x - x_i|$, and $h(y) = \sum_{i=1}^n w_i |y - y_i|$. The goal of finding a point $p = (x, y)$ that minimizes the value of $f(x, y)$ can be achieved by treating each dimension independently, because g does not depend on y and h does not depend on x . Thus,

$$\begin{aligned} \min_{x,y} f(x, y) &= \min_{x,y} (g(x) + h(y)) \\ &= \min_x \left(\min_y (g(x) + h(y)) \right) \\ &= \min_x \left(g(x) + \min_y h(y) \right) \\ &= \min_x g(x) + \min_y h(y) . \end{aligned}$$

Consequently, finding the best location in 2 dimensions can be done by finding the weighted median x_k of the x -coordinates and then finding the weighted median y_j of the y -coordinates. The point (x_k, y_j) is an optimal solution for the 2-dimensional post-office location problem.

2 Problem 2

- (a) We note that in the i th step, we have seen $i - 1$ cards, so there remains $n - i + 1$ cards unseen, and thus the probability that we guess correctly is $\frac{1}{n-i+1}$.
- (b) We apply linearity of expectation:

$$E[\text{Correct}] = \sum_{i=1}^n \frac{1}{n-i+1} = \sum_{i=1}^n \frac{1}{n} = \Theta(\log n),$$

where in the second to last step, we summed values in the reverse direction.

3 Problem 3

- (a) We claim that there is a large constant $C > 0$ such that the number of points is at most C . The claim is the following: suppose we have a set of r -spread points

in a square of side-length $10r$, then consider making a circle of radius $\frac{r}{2}$ around each point. We have that since the points are r -spread, none of the circles will intersect. Therefore, the total area contained within circles is a lower bound for the area of the rectangle of side-length $10r + r$ (note that points on the boundary will have part of the circle being outside the square of side-length $10r$, but fully inside the square of side-length $11r$). Thus, we have that the number of points is at most $\frac{4 \cdot 11^2 r^2}{\pi r^2} \leq C$ for a large enough C , independent of r .

- (b) Consider sorting the points in L and R according to their y -value. Now, consider the following algorithm:

1. We let $i = 0$ and $j = 0$. We look at $L[i]$ and $R[j]$ and consider the smaller y -value, suppose without loss of generality that it is $L[i]$ comes first (otherwise, we reverse the roles).
2. Consider the points in $R[j : j + C]$ and compare all of them. Maintain the closest pair.
3. Increment i and continue (if $R[j]$ was the smallest y -value, then we will increment j in this step).

First, we show the algorithm finds pairs between L and R which are closer than r . This is because if two points are within distance r , then when we analyze the smaller point, the larger point must be contained in a box going up from the bottom point of side-length $10r$, and there can be at most C of these from R the list (recall both R and L are r -spread). In addition, we will find the closest pair among L and R since we check all points within distance r .

Finally, note that the algorithm takes C time to compare all one point to all C points, and goes through each list once, which makes the algorithm run in time $O(n)$ once it is sorted, since sorting takes $O(n \log n)$ time, the algorithm takes $O(n \log n)$ time.

- (c) We will solve the problem in the following way, where we assume points are sorted along y -value:

1. We split the points according to the median x -value, suppose this occurs at m . Note that this requires $O(n)$ time to find the median, and $O(n)$ time to split the data set, additionally, this procedure does not change the order of the y -elements.
2. We now recurse on the left and on the right, and obtain two distances r_l and r_r corresponding to the closest pair of points on the left and right-hand sides, respectively.
3. We then consider $r = \min\{r_l, r_r\}$ and get rid of points which are less than $m - r$ in L and points which are greater than $m + r$ in R , according to the

x -value. On these remaining two lists, L' and R' , we run the algorithm from part (b).

If we sort naively, at each step, this algorithm takes $O(n \log^2 n)$ time; however, we claim that after the first sort operations, the subsequent sorts can be made in $O(n)$ time. In particular, splitting a sorted array takes $O(n)$ time and maintains sorted order. Thus, the sorting step in part (b) will take $O(n)$ time, which brings down the time complexity to $O(n \log n)$.

4 Problem 4

At a high level, the algorithm works as follows:

1. Begin at the root of the tree and compare the root value to x .
2. If the root value is less than x , then print the root value, and recurse down both child nodes.
3. If the root value is greater than x , stop.

In order to see that this algorithm is correct, note that we only return elements which are smaller than x , and in addition, if there exists an element which is less than x , then all ancestors will be less than x , so the recursive algorithm will reach the node.

In order to see the runtime of this algorithm is $O(\max\{k, 1\})$, we note that after comparing the root, every comparison of elements can be charged to the root node which was less than x . Thus, for each element returned, there were two comparisons (since the implementation is a binary heap).

5 Problem 5

- (a) We design an algorithm which maintains a counter, initially 0, and pointers i and j , where $i = 1$ and $j = n$. In each step, we proceed as follows:

1. We compare the element $A[i, j]$ to x .
2. If $A[i, j] > x$, then we decrement j .
3. If $A[i, j] \geq x$, then we increment i and add j to the counter.

If we ever have $i > n$ or $j < 0$, then we stop. We note that the algorithm takes $O(n)$ time, since we always either decrement j or increment i , which can happen at most n times each.

In order to see the algorithm works correctly, if we draw the array as a square, where columns are indexed by j and rows are indexed by i , then we have that the algorithm begins at the upper-right corner, and snakes its way down the array

moving left or down. We note that whenever we move down, it is because we are the biggest element which is less than or equal to x in that row. Therefore, when we reach the bottom, we will have accumulated all such elements.

- (b) The idea is now to binary search for the median, by picking appropriate values of $x \in [n^4]$. Thus, we execute the algorithm from part (a) a total of $\log(n^4) = \Theta(\log n)$ times. Thus, the algorithm takes $O(n \log n)$ time.