CSOR 4231: Analysis of Algorithms (sec.001) - Problem Set #2

Hongmin Zhu (hz2637) - hz2637@Columbia.edu

October 2, 2019

Problem 1 Solution:

(a) assume n_k is the number of x_i that is smaller than x_k, we have:

$$\begin{cases} \sum_{x_i<x_k} w_i = \frac{n_k}{n} < \frac{1}{2} \\ \sum_{x_i>x_k} w_i = \frac{n-1-n_k}{n} \le \frac{1}{2} \end{cases}$$

$\frac{n_k}{n} < \frac{1}{2} \Rightarrow n_k < \frac{n}{2}$

$\frac{n-1-n_k}{n} \le \frac{1}{2} \Rightarrow n_k \geq \frac{n}{2} - 1$

to satisfy both inequalities, $n_k = \frac{n}{2} - 1$ and n_k is an integer, so $n_k = \lfloor \frac{n}{2} \rfloor - 1$, which means there are equal numbers of x between $x_i < x_k$ and $x_i > x_k$, therefore the median is equal to the weighted median

(b) assume the class Node { value, weight }

```
ComputeWeighedSum ( A, n )
1. MergeSort (A) based on values
2. Sum ← A[0].weight
3. i ← 1
4. while (i<n) do
5.     newSum ← Sum + A[i].weight
6.     if (newSum >= 1/2 && Sum < 1/2) then
7.         return A[i].value
8.     Sum ← NewSum
9.     i ← i+1
10. return A[i].value
```

EXPLANATION:

first merge sort the array in $O(n \lg n)$ running time, then let Sum_i be the sum of the first i weights of x_i, use while-loop to update Sum_i until we find $Sum_k >= \frac{1}{2}$ and $Sum_{k-1} < \frac{1}{2}$, the weighted median is x_k

the merge-sort uses $O(n \lg n)$ time, while-loop takes $O(n)$ time, updating Sum_i takes $O(1)$ time. So the total running time $T(n) = O(n \lg n)$

(C) this problem is similar to the SELECTION problem mentioned in the lecture, which is the $O(n)$ running time, So based on that algorithm, below is the way to compute weighted median

```
ComputeWeightedMedian (A, i, n)
1.  if (i == n) then                                   // base case = number == 1
2.      return A[i].value
3.  if (n-i == 1) then                                 // base case = number == 2
4.      if (A[i].weight == A[n].weight) then
5.          return (A[i].value + A[n].value) / 2
6.      else if (A[i].weight > A[n].weight] then
7.          return A[i].value
8.      else then
9.          return A[n].value
10. pivot ← FindClosestToMedian (A)                    // find a number that is close to median
11. wleft ← sum the weights of A[i, pivot-1]
12. wright ← Sum the weights of A[pivot+1, n]
13. if (wleft == wright) then                          // compare sum weights between left part and
14.     return A[pivot].value                          right part of pivot
15. else if (wleft > wright) then
16.     A[pivot].weight ← A[pivot].weight + wright
17.     ComputeWeightedMedian (A, i, pivot)
18. else then                                          // change the weight of pivot, and do recurssion
19.     A[pivot].weight ← A[pivot].weight + wleft
20.     ComputeWeightedMedian (A, pivot, n)
```

EXPLANATION:

line 10-20 do the following things: find the pivot and calculate the sum of weights between the left part and the right part of pivot, if one of them is smaller, add this sum weight to A[pivot]. weight, which means to shorten the array, because we know the weighted median is in the part that has bigger sum weight. Then we do the recursion.

Total running time = $T(n) = T(n/2) + M(n) + O(n)$, $M(n)$ is the time to find pivot and $O(n)$ is the time to split array into 2 parts

we learned from the lecture, we can make $M(n)$ equals to $O(n)$, therefore the total running time $T(n) = \Theta(n)$ in the worst case

1(d) assume p_k is the weighted median, P_x is the point where the post office should be and $P_x = P_k + \varepsilon$, ε is a small number, assume $\varepsilon \geq 0$

$$\sum_{i=1}^{n} w_i d(p_k, p_i) = \sum_{i=1}^{n} w_i |p_k - p_i| = \sum_{p_i < p_k} w_i (p_k - p_i) + \sum_{p_i > p_k} w_i (p_i - p_k)$$

note that when $p_i = p_k$, $d(p_i, p_k) = 0$

$$\sum_{i=1}^{n} w_i d(p_x, p_i) = \sum_{i=1}^{n} w_i |p_x - p_i| = \sum_{i=1}^{n} w_i |(p_k+\varepsilon) - p_i| = \sum_{p_i < (p_k+\varepsilon)} w_i |(p_k+\varepsilon) - p_i| + \sum_{p_i > (p_k+\varepsilon)} w_i |(p_k+\varepsilon) - p_i|$$

$$= \sum_{p_i < p_k} w_i (p_k - p_i) + \sum_{p_i \leq p_k} w_i \varepsilon + \sum_{p_k < p_i < p_k+\varepsilon} w_i (p_k + \varepsilon - p_i) + \sum_{p_i > p_k} w_i (p_i - p_k)$$

$$+ \sum_{p_k < p_i < p_k+\varepsilon} w_i (p_k+\varepsilon - p_i) - \sum_{p_i > p_k} w_i \varepsilon$$

$$= \sum_{p_i < p_k} w_i (p_k - p_i) + \sum_{p_i > p_k} w_i (p_i - p_k) + 2 \sum_{p_k < p_i < p_k+\varepsilon} w_i (p_k + \varepsilon - p_i)$$

$$+ \sum_{p_k \geq p_i} w_i \varepsilon - \sum_{p_k < p_i} w_i \varepsilon$$

if $\text{Sum}_k = \sum_{i=1}^{n} w_i d(p_k, p_i)$

then $\sum_{i=1}^{n} w_i d(p_x, p_i) = \text{Sum}_k + 2 \sum_{p_k < p_i < p_k+\varepsilon} w_i (p_k + \varepsilon - p_i) + \varepsilon \left(\sum_{p_i \leq p_k} w_i - \sum_{p_i > p_k} w_i \right)$

$\because p_k < p_i < p_k+\varepsilon \quad \therefore 2 \sum_{p_k < p_i < p_k+\varepsilon} w_i (p_k + \varepsilon - p_i) > 0$

$\because p_k$ is the weighted median

$\therefore \sum_{p_i \leq p_k} w_i \geq \frac{1}{2}$, $\sum_{p_i > p_k} w_i < \frac{1}{2} \Rightarrow \varepsilon \left(\sum_{p_i \leq p_k} w_i - \sum_{p_i > p_k} w_i \right) > 0$

$\therefore \sum_{i=1}^{n} w_i d(p_x, p_i) > \text{Sum}_k$

$\therefore$ the weighted median must be the best Solution for 1-dimensional post-office problem

1(e) $$\left\{ \begin{array}{l} \sum_{i=1}^{n} w_i d(p, p_i) \\ d(p, p_i) = |x_1 - x_i| + |y_1 - y_i| \end{array} \right\} \Rightarrow \sum_{i=1}^{n} w_i (|x_1 - x_i| + |y_1 - y_i|) = \sum_{i=1}^{n} w_i (|x_1 - x_i|) + \sum_{i=1}^{n} w_i (|y_1 - y_i|)$$

in order to find the best solution, we need to minimize both $\sum_{i=1}^{n} w_i |x_1 - x_i|$ and $\sum_{i=1}^{n} w_i |y_1 - y_i|$, these two subproblems are identical to the problem in problem (d), So we know the weighted median of x_i will make $\sum_{i=1}^{n} w_i |x_1 - x_i|$ the smallest, and ~~x~~ the weighted median of y_i will make $\sum_{i=1}^{n} w_i |y_i - y_1|$ smallest

So in conclusion, the weighted median of x coordinates and the weighted median of y Coordinates will be the best pair for 2-dimension post office problem.

Problem 2 Solution:

(a) Let X_i be the indicator random variable associated with the event in which the i^{th} step is correct, $X_i = I\{$the i^{th} ~~pa~~step is correct$\}$

Because we only consider the numbers that have not appeared so far, so

$$Pr(X_i = 1) = \frac{1}{n-i+1}$$

(b) Let X be the random variable denoting the total number of points

$X = \sum_{i=1}^{n} X_i$

we take expectation of both sides: $E(X) = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} Pr(X_i=1)$

$E(X) = \sum_{i=1}^{n} \frac{1}{n-i+1} = \frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \dots + \frac{1}{2} + \frac{1}{1}$

$\because E(X)$ is Harmonic Series $\therefore E(X) = \ln n + O(1) \approx \ln n$

$\therefore$ the random strategy is $\Theta(\log n)$

Problem 3 Solution:

(a) points in P are r-spread, so in order to maximize the number of points, we partition the x-y area by squares with side length r, therefore the four corners of these squares can be the targeted points, and the minimum distance is the side length r

So assume there are m numbers of x_i, $x_i \in [\ell_x, \ell_x + 10r]$, $x_{i+1} - x_i = r$

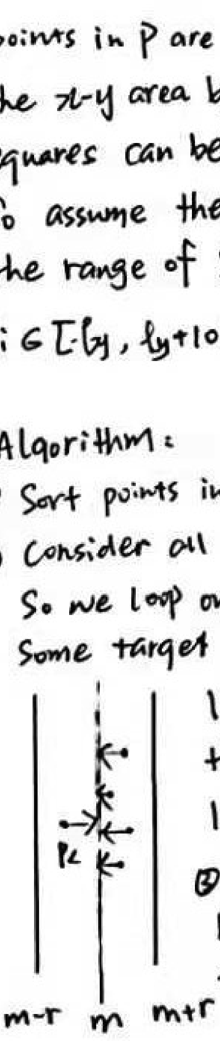
So the range of y axis is the same as x axis, so there are also m numbers of y_i, the asymptotic upper bound is $O(m^2)$, m is a constant

$y_i \in [\ell_y, \ell_y + 10r]$, $y_{i+1} - y_i = r$, so the asymptotic upper bound is $O(m^2)$, m is a constant

(b) Algorithm:

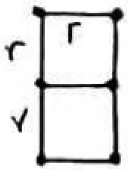
① Sort points in L and R by the y cordinate, which takes $O(n\log n)$ time

② Consider all points in L, and find if there is a point in R that meets the requirement. So we loop over the points in L, and for every point in L, P_L, we only need to check some target points in R, for example, we project P_L and points in R on to the middle line of the band, we know that if the projection of some point from R is within the distance r of the projection of P_L, then it is possible that this pair has less distance than r. So those are the points we need to look at.



③ now we know which points to look at, we simply calculate the distance between P_L and them to get the minimum distance, this takes $O(n)$ time, which ~~is~~ is the time to loop over all points in L.

④ the tricky part is how many points exactly are we looking at in R
for every point in L, points of R within the distance r of P_L must lie in
a $2r \times r$ rectangle, so it's safe to check 6 points around P_L's progection.



Pseudo-code:

```
   findMinimal(L, R, ~~D~~, r)
1. A ← Sort(L, R) by y coordinate, minimal ← r
2. for (i=0; i< A.length; i++) do
3.     if (A[i].x ≤ m) then
4.         count ← 0
5.         j ← i-1,
6.         while (j>=0 && count<3) do
7.             if (A[j].x > m) then
8.                 minimal ← min(distance(A[i], A[j]), r)
9.                 count ~~to~~ ← count +1
10.        count ← 0
11.        J ← i+1
12.        while (count<3 && j<A.length) do
13.            if (A[j].x > m) then
14.                minimal ← min(distance(A[i] A[j]), r), result ← (A[i], A[j])
15.                count ← count +1
16. return ~~minimal~~ result
```

Time complexity: Sorting takes $O(n\log n)$ time and for-loop takes $O(n)$ time
therefore the total running time is $O(n\log n)$

Proof: Loop-invariant: at the start of iteration with i of the loop, the variable minimal should contain the minimal distance between ~~of~~ ~~any~~ points in L and points in R

Initialization: Prior to ~~at~~ the start of the first loop, we have i=0, the variable should contains minimal distance, since L, R ~~are~~ r-spread, the minimal should be r, which is what minimal has been set to.

Maintenance: Assume loop invariant holds at the start of iteration i, then it containg the minimal distance between points in L and points in R and it is stored in variable minimal. There are two cases: (1) compute distance between A[i] in L and 3 points in R that are positioned before A[i] in the array A, so for point A[j] that has smaller y coordinate than A[i], we calculate the distance and take the minimum between the distance and r, and store it in minimal, thus in this case, the loop invariant holds (2) compute distance between point A[i] in L and 3 points in R that have bigger y coordinate than A[i], so for point A[j] in R, we calculate the distance and assign the minimum between distance(A[i], A[j]) and r, thus in this case, the loop invariant holds

Termination: when the loop terminates $i = (n-1)+1 = n$, Now the loop invariant gives: The variable minimal of all points in L with potential points in R. this is exactly what the algorithm should output. Therefore the algorithm is Correct

(C) Algorithm:

① Sort P based on x coordinate, which takes $O(n\log n)$ time

② find the median of sorted array, m

③ divide the array P into 2 halves, the first half $P_x \le m.x$, Second half $P_x > m.x$

④ recursively find the smallest distance r_1 in first half and r_2 in Second half. use base cases: 1) if only have one point, $r_1 = \infty$ 2) if have 2 points, $r_1 = |P_2 - P_1|$

⑤ r_1 is the smallest distance if 2 points are in the first part
r_2 is the smallest distance if 2 points are in the Second part
$r = \min(r_1, r_2)$, now we need to find smallest distance when ~~one~~ points are in different parts, which is a similar question compare to (b)

⑥ suppose we find a vertical line in the middle and get a band with width of $2r$, L is a set of points (x,y) such that $m-r \le x \le m$, R is a set of points (x,y) such that $m < x \le m+r$, the points within this band are $(x,y) \in A$

⑦ the question is the same as problem (b), so we know this part takes ~~$O(n\log n)$~~ $O(n)$ time

⑧ finally we return the smallest distance find from above steps

Pseudo-Code:

we will use algorithm from problem (b), findMinimal(L, R, r) which returns global smallest distance, the inputs are points set L and points set R, the smallest distance r of points that are in the same side. The input P is sorted.

```
smallestDistance (P)
1. if |P| == 1, return ∞
2. if |P| == 2, return distance(P_2, P_1)
3. else then
4. m ← median(P)
5. L ← (x,y) ∈ P && x ≤ m
6. R ← (x,y) ∈ P && x > m
7. r_1 ← smallestDistance (L)
8. r_2 ← smallestDistance (R)
```

```
9. r ← min (r1, r2)
10. result ← findMinimal (L, R, r)
11. return result
```

Time complexity:

Sorting P takes $O(n\log n)$

algorithm uses divide-and-conquer with running time $T(n) = 2T(n/2) + O(n)$, $O(n)$ is the time complexity of the algorithm of finding smallest distance which points are in 2 sides

$T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n\log n)$

thus, the overall running time is $O(n\log n)$

~~Proof~~ Problem 4 Solution:

Algorithm:

① ~~if~~ consider the min-heap in the form of tree, if the root value is bigger than x, then the number is zero

② if root value is smaller than x, then we consider the left leaf and right leaf seperately and compare the value with x

③ recursively consider all the nodes of this tree, until we find a node whose value is bigger than x, we stop finding and return the ~~current number of~~ nodes that are smaller than

Pseudo-code:

```
findElements (A, x, index, result)
input: min-heap A, target number x
       index: current index of array, initailized to 1
       result: array list, initialized to empty list
1. if (i > A.length || A[i-1] >= x) then
2.     return
3. els then
4.     ~~result.add(A[i-1])~~ if (A[i-1] < x) then result.add(A[i-1])
5.     findElements (A, x, index*2, result)  // left leaf
6.     findElements (A, x, index*2+1, result)  // right leaf
```

Time complexity:

comparing A[i] and x takes $O(1)$ time, if a subtree's root ~~value~~ has a value that is ~~smaller than~~ greater than or equals to x, then by the definition of min_heap, all of its desendents will have values greater than or equal to x. The ~~φ~~ algorithm need not explore deeper than the items it's traversing, hence the running time is $O(k+1)$

Problem 5 Solution:

(a) Algorithm:

① Since each row and column is sorted in ascending way, we start with the point A[i][j], i=0, j= columnNumber -1

② if A[i][j] is smaller than ~~or~~ or equal to x, we take the entire number of this raw into count, i.e. count = count + (j+1), and we do i++ to consider next row

③ if A[i][j] is bigger than x, which means we need to move current point to the left ~~④ if~~ to make the value smaller

④ when i or j hits the boundary, we stop and return count

Psudo-code:

```
~~find~~ CountNumber (A, x)
1.  if (x < A[0][0] || x > A[rowNum -1][ColNum -1])
2.        return 0
3.  i ← 0 , j ← ColNum -1 , count ← 0
4.  while ( i < A.length && j >= 0) do
5.        if (A[i][j] ≤ x) then
6.              count ← count + (j+1)
7.              i++
8.        else then
9.              j--
10. return count
```

Time Complexity:

the while-loop will stop if i or j hits the boundary, so in the worst case, we go over an entire row and an entire column, so the running time is O(2n), which can be simplified to O(n)

(b) Algorithm:

① the goal ~~[illegible]~~ is to use binary search and algorithm proposed in part A to find median

② we use binary search to find a number X and we use CountNumber (A,X) from part A to count the number smaller than or equal to X, if the number equals to ~~[illegible]~~ half size of A, then this number is the median, if not, continue binary search

③ if median is found, we need to make sure it's the number from A, so we will use similar approach as algorithm from part A to find the number in A that is closest to median

Pseudo-Code:

```
    findMedian (A, n)
1.  low ← A[0][0]
2.  high ← A[n-1][n-1]
3.  if (n%2 == 0) then
4.      halfsize ← n^2/2
5.  else then
6.      halfsize ← (n^2+1)/2
7.  while (low < high) do                          // binary search
8.      mid ← low + (high-low)/2
9.      count ← countNumber (A, mid)               // algorithm from part A
10.     if (count == halfsize) then
11.         break;
12.     else if (count < halfsize) then
13.         low = mid+1
14.     else then
15.         high = mid-1
16. i ← 0, j ← n-1, median ← A[0][0]               // find median in A
17. while (i < n && j >= 0) do
18.     if (A[i][j] ≤ mid) then
19.         median ← (A[i][j] > median) ? A[i][j] : median
20.         i++
21.     else then
22.         j--
23. return median
```

Time Complexity:

the algorithm contains 2 parts, part 1 uses binary search to find a number X, and countNumber () is used to count ~~number of~~ number of values that are smaller than or equal to that number X, so the running time is $O(\log n^4 * n) = O(4 n\log n)$, which can be simplified to $O(n\log n)$

part 2 tries to find a number in A to be the median, this part uses same logic as countNumber, so the running time is $O(n)$

So total running time is $O(n\log n) + O(n)$, which is $O(n\log n)$