COMP2007 Assignment 2 Report

Jiashu Wu 460108049 jiwu0083

Question 1

1.1 Description of how the algorithm works (In plain English)

After taking all the inputs and storing them in a list, my algorithm performs a merge sort to sort every point by their X coordinate in ascending order (Since the question is in 1 dimension thus each point will be represented by its X coordinate only). Base on the definition of "superior", after performing the sorting operation, now each point Pi in the list is superior to all the points before it (except these points who have the same X coordinate as Pi). And then for each point Pi, my algorithm count the number of points which has the X coordinate smaller than the X coordinate of point Pi, and the counting result is the importance of point Pi.

1.2 Argue the correctness of the algorithm

Since Merge-Sort is covered in the lecture thus we assume that it is correct without proving.

Then we need to prove that after sorting X coordinate in ascending order, for each point Pi, it is superior to all the points before it (except these points who have the same X coordinate as Pi)

Definition of "Superior" and "Inferior" in 1 dimension: A point Pi is said to be superior to a point Pj if and only if Pi(X) > Pj(X) (Strictly greater than). And we say that Pj is inferior to Pi.

Thing needs to be proved: After sorting X coordinate in ascending order, for each point Pi, it is superior to all the points before it (except these points who have the same X coordinate as Pi)

Prove by contradiction:

Let's suppose that after sorting X coordinate in ascending order, there exists a point Pi that is inferior to a point Pj, where Pj is before Pi and they have don't have the same X coordinate (Pi(X) != Pj(X)).

Since Pj is before Pi, we have Pi(X) > Pj(X). Since Pi is inferior to Pj and they don't have the same X coordinate, based on the definition of inferior, we have Pi(X) < Pj(X). Hence there is a ***contradiction:*** Pi(X) > Pj(X) and Pi(X) < Pj(X) can't hold together. Therefore, we can draw the conclusion that after sorting X coordinate in ascending order, for each point Pi, it is superior to all the points before it (except these points who have the same X coordinate as Pi).

Therefore, after sorting, the algorithm can correctly calculate the importance value of each point.

1.3 Prove the upper bound on the time complexity of the algorithm

1.3.1 Upper bound of the time complexity

The upper bound on the time complexity of the algorithm is O(n * log(n)), where n is the number of points.

1.3.2 Prove the correctness of the time complexity

Proof:

Input all the data: At the beginning, we need to store the value of n. We also need to store n points into an array. Each storing operation takes O(1). Therefore the input process takes O(1) + O(n) = O(n).

Merge-Sort: The time complexity of the merge sort is O(n * log(n)).

Counting the importance: In this process, I used a counter to keep track of the largest index of points Pj where Pj < Pi, thus we don't need to traverse the array n times, instead we only need to traverse the sorted X coordinate list once to count the importance of all points.

To explain this clearer, I will use the following example.

Sorted List (X coordinate): 0 1 2 2 2 3 5 5 7 Counter: 0 , 1 , 2 , 2 , 2 , 5 , 6 , 6 , 8

After we visit the n^{th} element in the sorted array, we update the counter to store the (largest index of Pj + 1) where Pj(X) < Pn(X), then we visit the $n+1^{th}$ element. Then if the $n+1^{th}$ element has the same X coordinate as n^{th} element, the importance of $n+1^{th}$ element is the counter, and if the X coordinate of $n+1^{th}$ element is greater than the X coordinate of n^{th} element, then the importance value of $n+1^{th}$ element is index of n plus one, and then we update the counter to the index of n plus one. Therefore, by using this counter, we only need to traverse the array once. Since the length of the list is n, thus this requires O(n) work.

Totally, the algorithm requires O(n) + O(n * log(n)) + O(n) = O(n * log(n)) work.

Question 2A (Merge step)

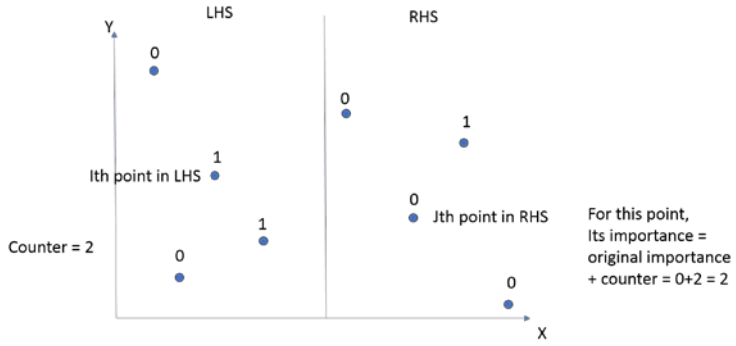
2.1.1 Description of how the algorithm works (In plain English)

Basically, the merge algorithm will keep comparing elements from left side and right side, and add them into the merge result array with merged importance value.

Firstly, we create a counter to keep track of the number of points in LHS (Left Hand Side) which is inferior to (in this case is strictly below) the point Pi in the RHS (Right Hand Side). Initially the counter is 0. Also, we create two variables i and j to keep track of the index of points which we are currently working on in LHS and RHS respectively. Initially i = j = 0, which means we will starting comparing the lowest point of both side.

Since for both side the points, they are sorted by Y coordinate in ascending order, thus if the Y coordinate of point with index i in LHS is less than the Y coordinate of the point with index j in RHS, we add that point of LHS into the merge result array with its original importance value unchanged, and then increase the counter by one, and then increment the index i by one. And if the Y coordinate of the point with index i in the LHS is greater than or equal to the Y coordinate of the point with index j in the RHS, then we add that point of RHS into the merge result array with its original importance value plus the counter, since there are

counter number of points in the LHS which is inferior to this point and originally there are imp number of points which is inferior to this point in RHS. Thus, the merged importance value is the original importance value plus the counter. And then we increment the index j by one to make it points to the next element of RHS. We will keep doing so until either side exhausted.



If the RHS exhausted, we then add all points which are unvisited in LHS into the merge result array with the original importance value unchanged. And if the LHS exhausted, we then add all points in RHS which are unvisited into the merge result array with the importance value equals to the original importance value plus the counter.

Finally, we return the merge result array.

2.1.2 Argue the correctness of the algorithm

Note: The following definition will be used during the proof.

Definition of "Superior" and "Inferior" in 2 dimensions: A point Pi is said to be superior to a point Pj if and only if Pi(X) > Pj(X) and Pi(Y) > Pj(Y) (Strictly greater than on every coordinate). And Pj is said to be inferior to point Pi.

Proof the correctness of the algorithm:

LHS:

For all points in LHS, the merge algorithm ends up adding all of them into the merge result array with their original importance value unchanged. This is obviously correct because all the points in RHS has X coordinate greater than all points in LHS, thus by definition all points in RHS are at least not inferior to all points in LHS, thus the importance value of all points in LHS will not be affected, thus the importance value of all points in LHS remain unchanged. Therefore, the algorithm done the LHS correctly.

RHS:

For each points Pi in RHS, the merge algorithm changed the importance value of Pi into its original importance value plus the number of points in LHS which has Y coordinates strictly less than the Y coordinate of Pi. The reason is that for all these points in LHS whose Y coordinate is strictly less that the Y coordinate of Pi, they also have X coordinate strictly less

than the X coordinate of Pi since they stays at the LHS, thus by definition they are all inferior to Pi, therefore the importance value of Pi should be its original importance value (Originally the number of points in RHS which is inferior to Pi) plus the counter (The number of points in LHS which is inferior to Pi). Therefore, the algorithm done RHS correctly.

Hence, the merge algorithm done both sides correctly, which means that the merge algorithm is correct.

2.1.3 Prove the upper bound on the time complexity of the algorithm

2.1.3.1 Upper bound of the time complexity

The upper bound of the time complexity if O(n), where n represents the number of points.

2.1.3.2 Prove the correctness of the time complexity

Input all the data: Since for this algorithm, we need to input the value of n, l, and r, and also n points (stored in an array, each storing operation costs O(1) work), therefore the input process takes O(3) + O(n) = O(n)

Merge algorithm: Since for the merge algorithm, we only need to traverse the LHS and RHS once, and for both LHS and RHS the maximum number of points is n, thus traversing requires O(n) work. And also we need to add all points into the merge result array, since we have n points in total and each adding operation takes O(1) time, therefore, adding all points takes O(n) time.

Thus, the whole merge algorithm takes O(n) + O(n) + O(n)= O(n) time.

Question 2B (Divide and Conquer Algorithm)

2.2.1 Description of how the algorithm works (In plain English)

For this algorithm, firstly I use merge sort to sort all points by their X coordinate. After sorting, the algorithm keep dividing the points array into two halves recursively (divide by middle index = (start + end) / 2), until each part only have one point in it (length = 1). After dividing, we merge all lists together using the merge algorithm to get the importance of all points.

2.2.2 Argue the correctness of the algorithm

Note: The merge algorithm is correct based on Question 2A.

Proof:

I will prove the correctness of this algorithm using **Prove by Induction**.

a) Inductive Claim

The algorithm can correctly calculate the importance value of all points at each step.

b) Base Case

For a list which contains only one point, we don't need to divide it. The algorithm will correctly calculate its importance value, which is 0, and then the merge algorithm will merge it. So obviously, this will produce the correct result.

c) Inductive Hypothesis

At K^{th} divide and merge, the algorithm can correctly calculate the importance value of each points.

d) Inductive Step

At $K+1^{th}$:

Since the inductive hypothesis holds, the algorithm correctly calculates the importance value of two halves L to M and M + 1 to N. And by Question 2A we have already proved that the merge algorithm is correct, therefore, after executing the $K+1^{th}$ step, the algorithm can correctly calculate the importance value between L and N.

e) Corollary to Inductive Claim

We can draw the conclusion that the algorithm can correctly calculate the importance value of all points at each step.

Therefore, we can draw the conclusion that our algorithm is correct.

2.2.3 Prove the upper bound on the time complexity of the algorithm

2.2.3.1 Upper bound of the time complexity

The time complexity of this algorithm is O(n * log(n)), where n is the number of points.

2.2.3.2 Prove the correctness of the time complexity

Note: By Question 2A, the time complexity of merge algorithm is O(n).

Proof:

Input all the data: Since we need to store the value of n and also n points, the total time of input is O(1) + O(n) = O(n).

Merge-Sort: Using merge-sort to sort all points by their X coordinate takes O(n * log(n)) since there are n points in total, and the time complexity for merge-sort is O(n * log(n)). Therefore the sorting process requires O(n * log(n)) work.

Divide and merge: Since for each divide operation, the list will be divided into two halves, and the algorithm keep doing this recursively until reaching the base case, which is a list with length 1. Thus by Master Theorem, the time complexity for this is T(n) = 2*T(n/2) + O(n) = O(n * log(n)).

Therefore, the time complexity of the whole algorithm is O(n) + O(n * log(n)) + O(n * log(n))= O(n * log(n))