COMP2007Assignment2,Liam Jay-Ling Chiang,460368148

by Liam Chiang

Submission date: 11-Sep-2017 06:05PM (UTC+1000)
Submission ID: 844479022
File name: COMP2007Assignment2,Liam_Jay-Ling_Chiang,460368148.pdf (131.25K)
Word count: 1357
Character count: 5922

14 Question1:

No Correctness Proof; Part 1.b: 0

a). *Implementation: 8*

5 **Step1:**
First of all, we put all of the inputs into the reverse priority queue.

Step2:
Secondly, we loop through the reverse priority queue. In each iteration, we get the point with the maximum value. If this value already in the hash map, and then we update the hash map that stores the frequency of the point by incrementing 1. If this point has already exist, the hash map, which stores the importance value will decrement by 1 or we add the new entry with this point and the size of the rest of the reverse priority queue as the importance value.

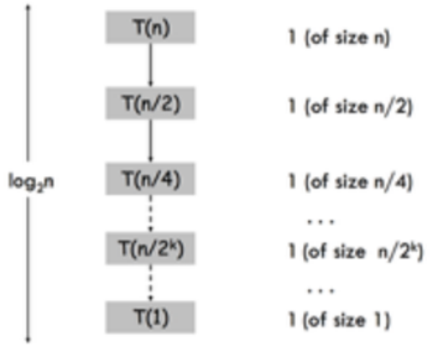
Step3:
Thirdly, we iterate the importance hash map and put those value into the normal priority queue in order to print out the importance value in ascending order.

b).

1 **Time complexity for each step:**
The fist step of the time complexity is O(n log n)
The second step is O(n)
And the last step is O(n)
Overall, the total time complexity is O(n log n) + cn, which is O(n log n)

proof of priority queue time complexity:
Assume T(n) is the number of the comparisons to heap sort an input of size n.
We use the priority queue to do the heap sort (T(n) = O(1) + T(n/2))on the points of left and right parts. The time complexity of inserting the points in the priority queue would be O(n log n) since we have traversed all of the points (n points).

$$T(n) = T(n/2)+O(1)$$



Greedy algorithm V.S. Priority queue (heap sort):
In greedy algorithm, the time complexity of solving this question would be O(n ^ 2) since each point would check whether other points are smaller than the one through traversing all of the points one by one with each point. Instead, the reverse priority queue uses heap sort to sort the points from bigger to smaller points after insert all of the points into the queue, the time complexity of it should be totally O(n log n).

c).
The upper bound is still the same the worst case, which is O(n log n) because the algorithm does not exist the best case and worst case.

15 Question2 a: *Implementation: 10*
i) 2

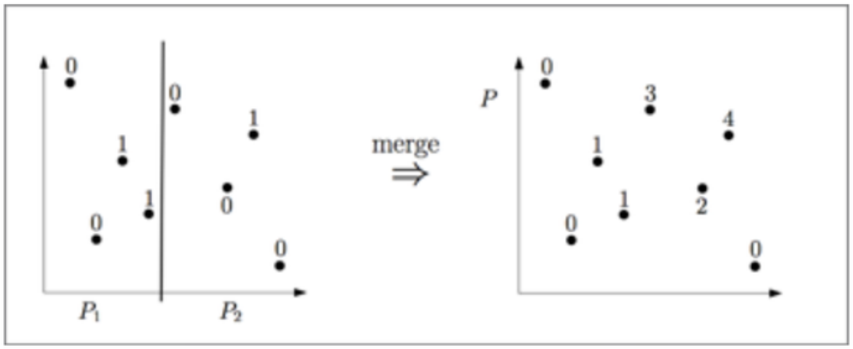
1 **Step1:**
First of all, we store the left parts and right parts into the two different priority queues, which are sorted by y-value. 3

Step2:
Secondly, we get each element out of these two priority queues and store them into two separated lists.

Step3:
The third step, since the left part's x-values are definitely smaller than the right part's x-value, therefore, we just merge these two lists into one lists by y-value ascending order and in order to update the importance-value. 4

Step4:
The last step would put these list into another priority queue, which is sorted by x-value, so that we can output the answers in x-value of ascending order

ii).
3 **Solut 5 dea:**
The question has assumed there are two parts, which are left part and right part and the question gives the 2 point sets input and the importance values in each left and right part.
As the example diagram shows in the assignment, we can tell that this is about 2D rank finding problem and we only need to do merging.



Assume the first set of input is x point (horizontal points) and the second set is y point (vertical points). Since the point has to dominate other points by bigger x value and y value. Hence we can declare 2 core truths: 1). Firstly, the points in left part definitely can not dominate the points in right part since the point has to dominate other points through the x value and y value. 2). Secondly, the points in right part dominates the left points depends on their y value. Therefore, instead of caring about the x value, we should only check whether the points' y value in left part are smaller than each point in the right part. Since the truths are correct. We don't even need to update the importance value of the points in left part. Instead, we should update the importance value of right part since the points in right part have chance to dominate the left-part points. 6

Time complexity for each step:

In step1:

The time complexity will be O(nlogn)

In step2:

The time complexity will be O(n)

In step3:

The time complexity will be O(n)

In the final step:

The time complexity is still O(n log n)

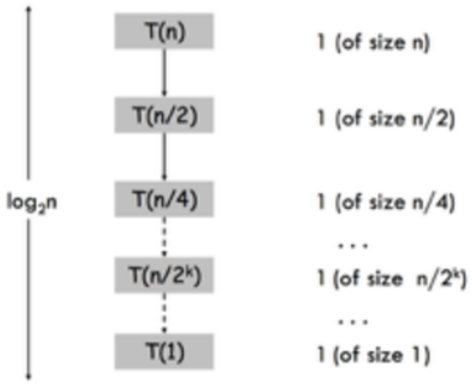
Overall, the total time complexity would be O(cnlogn) + O(cn), which is still O(n log n)

proof of priority queue time complexity:

Assume s the number of the comparisons to heap sort an input of size n.

We use the priority queue to do the heap sort (T(n) = O(1) + T(n/2))on the points of left and right parts. The time complexity of inserting the points in the priority queue would be O(n log n) since we have traversed all of the points (n points).

$$T(n) = T(n/2)+O(1)$$



Greedy algorithm V.S. Priority queue

If we do the question with greedy algorithm, the each point would just traversal every points in the 2 sets(left , right part) to update their importance value. It would cause O(n ^ 2) for traversing all of the points in the two sets. Instead, we use the priority queue to sort the y value of the points in each left and right part would only cause totally O(n log n). Obviously, O(n log n) < O(n ^ 2).

iii).

since we have to put the data into priority queue, hence the upper bound of the time complexity would not be better than O(n log n)

18 Question2 b: *Implementation: 12*

i).

4 **Step1:**

First of all, store the inputs into a priority queue, which is sorted by x-value and then get each element out of this priority queue and store in the list.

Step2:

Secondly, recursively divide the list into a half until the sub-list has 1 element. When it tracing back, we do the merge by sorting the sub-list through y-value in the priority queue.

Step3:

And finally, after the divide-and-conquer has done, we put the final answer into the priority queue, which is sorted by x-value and print out the outputs.

ii).

2 **Solution idea:**

This question's idea is ex[11]the same and has been mentioned (divide and conquer) from question2 part a (2D rank finding).

Time complexity for each step:

In step1:

The time complexity of the step 1 will be O(n log n) + O(n) = O(n log n)

In step2:

The time complexity of this step is O(n log n)

In the la[12]p:

The time complexity of this step will be O(n log n) + O(n) = O(n log n)

Overall, the total time complexity will be O(n log n)

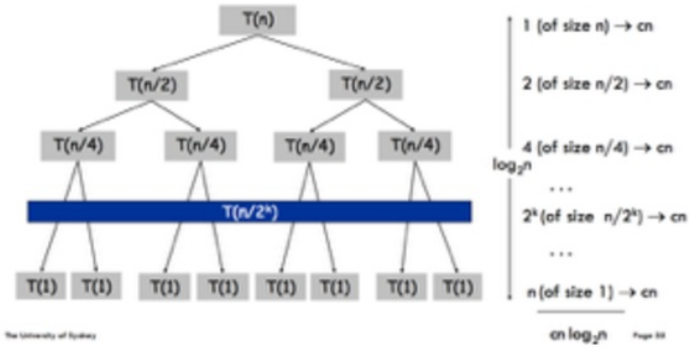
Proof of merge sort time complexity:

Assume T(n) is the number of comparisons to merge sort an input of size n.

The merge sort recurrence formula would be:

$$T(n) \le \begin{cases} 0 & \text{if } n=1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

$$T(n)=\begin{cases} c & \text{if } n=1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$



Therefore, the final time complexity of the merge sort would be T(n) = O(n log n)

Greedy algorithm V.S. Merge sort

If we do the question with greedy algorithm, the each point would just traversal every points in the 2 sets(left , right part) to update their importance value. It would cause O(n ^ 2) for traversing all of the points in the two sets. Instead, we use the priority queue to sort the y value of the points in each left and right part would only cause totally O(n log n). Obviously, O(n log n) < O(n ^ 2).

iii).

The upper bound of the time complexity is still the same as the worst case which is O(n log n), since the array needs to be divided and conquered under any situation.

Reference:

UNIVERSITY OF SYDNEY Lecture slide week 4 Divide and conquer by Joachim

COMP2007Assignment2,Liam Jay-Ling Chiang,460368148

GRADEMARK REPORT

FINAL GRADE

47 /100

GENERAL COMMENTS

Instructor

PAGE 1

Text Comment. 14

Text Comment. No Correctness Proof; Part 1.b: 0

Text Comment. Implementation: 8

Text Comment. 5

Text Comment. 1

Comment 1

These need more justification. Furthermore, your method of printing the values out in reverse order involves resorting them with a priority queue, which should take O(nlogn) time.

In addition, since you use hash maps, your complexity is only an expected time, which you fail to mention.

PAGE 2

Text Comment. Implementation: 10

Text Comment. 15

Comment 2

I don't see how this actually calculates the importance values.

Text Comment. 1

Comment 3

Sorting the halves takes O(nlogn). This makes your merge step too slow. It needs to occur in linear time if the overall divide and conquer algorithm is to take O(nlogn) time.

Comment 4

How is this done?

Comment 5

This is more description of your algorithm. Much of this should be in part i)

Text Comment. 3

Comment 6

How? You're glossing over the entire algorithm here.

Text Comment. 1

Comment 7

This stuff should really be in part iii)

Comment 8

delete-min operations on a priority queue may require rebalancing, requiring O(logn) work, so step 2 should be at least O(nlogn).

Comment 9

Given that you did not describe how to calculate the importance values, it might be as bad as the naïve method, taking quadratic time.

Comment 10

You don't need to prove this. But since you have, why did you not use the result for step 2

above?

PAGE 4

Text Comment. 18

Text Comment. Implementation: 12

Text Comment. 4

Text Comment. 2

Comment 11

And you still haven't really described it. Nor have you proven its correctness.

Comment 12

But why?

Lower down you quote the running time of merge sort, but your algorithm has to differ slightly from merge sort. You can't just quote that running time at me and expect to get marks.

PAGE 5