COMP2007 - Assignment 2

Matthew Watson
SID: 440267858

1. **Compute the importance of each point in P**

   (a) **Description of how your algorithm works**
   My algorithm first reads from stdin to assign n, the number of points, and *points*, an array of the points' x-values. In the importance() method, it first sorts the points in order from smallest to largest using an implementation of the HeapSort algorithm. Then, once it has an array of sorted points, it starts at the first index of 0 where it simply initialises a counter of duplicate importance values and the number of lower/previous points to 0 and then returns the element at index 0 as it's the first point of lowest weight. Then on each iteration it checks if it is at a point of equal x-value to the previous one and if so, increases the counter of duplicate importances and returns the number of lower points (corresponding to its ranked weight. Otherwise, if at the iteration i, it is at a point of greater x-value than the previous point, it increments the number of lower points by one and adds the number of duplicates to it to return the adjusted number of lower points (factoring in duplicates) to achieve the ranked weight of that iteration. It then resets the duplicates to 0. Once it iterates n number of times, the program exits.

   (b) **Argue why your algorithm is correct**
   HeapSort is a well known algorithm invented by J. W. J. Williams in 1964 and has been well proven to sort a list of unsorted integers. With regards to the proof of the importance method, the following proof is observed.
   Once every point has been sorted by x value from smallest to largest, we can begin going over each point (represented by k for simplicity) in the for loop in the importance method. For the first iteration, we are at i==0 and so there can be no previous points and therefore no possibility of a point that exists such that k is superior to it. As such we can set the counter of duplicate importance points to 0, the number of lower points to 0 and print that. Then, for each iteration we will have two potential possibilities, either that the current point k is equal to the previous point *k-1*'s x=value and so we have found a duplicate importance value (point of equal importance); a case which we handle by incrementing a counter of duplicate importance values by 1 and printing the

current k importance. Otherwise we will have a case such that k is superior to *k-1*. To account for duplicate importance values, we do not simply increase the importance of k by 1, but instead increase it by 1 in addition to the counter of duplicate importance values. These importance values will either be 0 (this is the first occurrence of this x-value) or 1 or more in the event we have previously seen several points with the same x-value and as such must increase it by more than 1). This ensures we account for the case in which, for example, we iterate over 3 points of x-value 3 and then reach a point of x-value 4. This may result in all importances of 3 being 2, but without a counter, the importance of x-value 4 would only be 3, when in reality, there are several more points it is superior to which have been counted by this counter; hence the real importance value will be 2+3 = 5.

(c) **Prove an upper bound on the time complexity of your algorithm**
My algorithm first builds a heap from the input priority queue. In general, insertion into a heap requires *O(logn)* work therefore to insert n elements we require *O(n log n)* work. We then iterate through the heap and access elements in the heap in *O(1)* time, n times, thus taking *O(n)* time. Therefore our final algorithm is:

$$O(n \log n) + O(n)$$
$$= O(n \log n)$$

2. **Compute the importance in 2D**

(a) **Combine the solutions**

i. **Description of how your algorithm works**
For your understanding, the point sets p1 and p2 are often referred to as left and right for simplicity.
My algorithm reads from stdin to assign n as the total number of points and l as the number of points in the first set. Then, it reads in the points in *p1* as tuples to two priority queues, where *leftQ* uses *x2* as the key and *leftQA* uses *x1* as the key (*leftQA* is never modified and only used for output as explained later). It then assigns r as the number of points in the second set and reads in the points from *p2* as tuples into *rightQ* using *x2* as the key.
With the data now loaded, we enter our first while loop; while *leftQ* and *rightQ* are not empty. It starts by getting the first point from the left and right sets and extracts the *x2* value from each point as this is the only value we need for comparison (see proof below). It checks if the left value is greater than the right value, in which case we are at a left *x2* point of greater value than the right. This is not desirable for an importance increase, although we may already be somewhat through our iteration and

have finished finding a set of smaller left values in the following loop that has increased our counter and are now ready to write the final update to our output point set *rightQA*. Accordingly, we put the left point back into the left input *leftQ* as we will want to come back to it on the next iteration and we update our output right value in *rightQA* with an updated importance if indeed it has reached this point due to having been through a few left points of smaller value. Otherwise, if the left value is smaller than the right value, this is desirable for an increase in importance and so we increase our counter and discard the left value (i.e. we don't put it back into *leftQ*). However, since there may be a few points like such, we don't want to update the right point just yet until we reach our previous case where left is greater than right (in which case we've found all such points and can update the importance of our right point). Following this while loop, we will have an empty left set of points but may still have one right point where the left value is less than the right value with some counter value ¿ 1 that will mean there are 1 or more points it carries importance over. Hence, we get this last point, update the importance with the addition of the counter value and write this to the output point set *rightQA*.
Finally, to output our results, we simply print the importance value of each element in *leftQA* as it was read into a priority queue with key *x1* and was already provided with the correct importance values as explain ed below. Then, it prints the importance values of each element of our right output priority queue *rightQA* which is already sorted by *x1* values by the nature of priority queues given the key is *x1*.

ii. **Argue why your algorithm is correct**
Due to the fact that any *x1* value in *p2* is greater than all *x1* values in *p1*, we can assume that for every comparison of a point set in *p1* and *p2*, the *x1* value for *p2* will always be greater than that of *p1*. Furthermore, this means that the already computed importance values for points in *p1* will remain the same as they can not improve their importance over *p2* due to them needing to have a higher *x1* **and** *x2* value which is not possible as all *x1* values in *p1* are smaller than those in *p2*. Hence, the deciding factor of importance for a point in *p2* will merely be if there exists point sets in *p1* with a lower *x2* value than that of *p1*. Given the two PQs *leftQ* and *rightQ*, it will call *get()* on each which will return the point in each with the lowest *x2* value and hence iterate through from smallest to largest values of *x2* in each. To catch the case where we may find several left points that are smaller than the right point, we first check if left is greater than right in which case we hold onto the left point but update the importance of right and discard it to move onto the next higher point. Then if we do indeed find a point where left is less than right, we discard left, increase the counter

but hold onto right so that we may keep iterating through the left side and compare to right until we've found all points in left that are smaller than right. Hence we go back to our right is greater than left case where we can update the importance with our counter value and discard our right side to move onto the next point.
Such a case would be where we compare 2<6, 3<6, 4<6, 7>6. In these four iterations, we would come across points 2, 3 and 4 in the left side which are less than the point value of 6 which we keep holding onto, increasing our counter by 1 each time to be 3. Once we reach left 7 and right 6, we are at a point where 7>6 and so we can hold onto left and update our right point with *x2* value 6 to have its importance increased by the counter value of 3. Finally, we may have a case where we've gone through all left points that are less than a point in right. To catch this, we have a loop while the left side is empty and the right side is not to catch this and update the right side. This will finish the iteration and we can then output our values sorted by *x2* handled by the priority queues that are keyed by *x2* to do the sorting for us.

iii. **Prove an upper bound on the time complexity of your algorithm**
My algorithm first builds two priority queues from the input. In general, insertion into a priority queue takes *O(log n)* time. This is done twice, $\frac{n}{2}$ times, thus $O(\frac{n}{2}\ log\ n)$ overall. Then, for the comparison stage, it iterates n times within which it performs a get on a priority queue in *O(log n)* time and access on a tuple in *O(1)* time - thus this stage takes *O(n log n)* time. Finally for printing the results in the correct order, the algorithm again utilises a priority queue which gets values in *O(log n)* time. It does this twice, $\frac{n}{2}$ times, thus $O(\frac{n}{2}\ log\ n)$ overall. Therefore our final algorithm is:

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

$$= O(n \log n)$$

(b) **Divide and conquer**

i. **Description of how your algorithm works**
My algorithm reads from stdin to assign n as the number of points, then initialises a blank priority queue to store them before iterating through n times to read all points into a priority queue of points, with *x1* as the key. It calls *compare(Q)* to run the algorithm on the provided priority q of points. The *compare()* function then recursively splits the provided points into two separate priority queues until it reaches two points of comparison to pass to the *merge()* function and then builds back up merging those set points in each recursion. The *merge()* function will start with a pair set of points,

left and right, comparing if the right is of more importance than the left, then updates the importance accordingly and passes it back to the recursive *compare()* function. On each recursive call, the left and right point sets that are compared grow until eventually it builds back up to the final recursion call where it has a left and right point set like that of *2a* of approximate size $\frac{n}{2}$ which it does a final merge call on and then returns *output* to the priority queue *out*. Finally, to print the output, it goes through each item in the *out* priority queue which will return importance values in order of increasing *x1* value as the priority queue is keyed by *x1*.

The merge function works similar to *2a*, however modified to run as a function to allow recursion. It is provided two sets of points like *2a* which are then iterated through and placed into new priority queues, *leftQ* and *rightQ* to order them by *x2*. A PQ *leftQA* is also created as a duplicate of the given left set which is not modified and only used for output as explained later. With the data now loaded, the remainder of the explanation of the *merge()* algorithm can be observed in *2a*. Because this function is called recursively and will thus start with 2 points and build up from there (in a fashion similar to merge sort), it will follow the same properties as in *2a* with regards to the assumptions of the left and right point sets.

ii. **Argue why your algorithm is correct**

This algorithm re-uses that of *2a* in method form to allow for recursion, as such, much if its proof is applicable for this algorithm. The difference in this algorithm is that it calls on that function, represented as *merge()*, recursively. To allow the logic and proof of *2a* to hold, the recursive function breaks down the provided list of points of length n into two sides of approximately equal size, left and right, which due to this algorithm sorting the input list by *x1* on input, provides two lists of same properties to that of the input to *2a*. These are that any *x1* value in *right* is greater than all *x1* values in *left*, therefore we can assume that for every comparison of a point set in *left* and *right*, the *x1* value for *right* will always be greater than that of *p1*. Hence, the deciding factor of importance for a point in *right* will merely be if there exists point sets in *p1* with a lower *x2* value than that of *p1*. This extends to our recursive function that breaks the list down to a base case of a left and right set of points containing only one point in each side and so it can be observed that right is of more importance if its *x2* value is greater than that of left. As the recursive function builds back up, it is simply calling on the merge function exactly as *2a* where the left and right sides are sorted by *x2* and that all *x1* values in *right* are greater than any in left and so the same comparison based on only *x2* will still hold.

iii. **Prove an upper bound on the time complexity of your algorithm**

My algorithm first builds a priority queue from the input. In general, insertion into a priority queue takes *O(log n)* time. This is done n times, thus *O(n log n)* overall. The compare function which handles the divide stage of the 'divide and conquer' algorithm reads and fills several priority queues. The get and put functions both take *O(log n)* time and are carried out n times resulting in *O(n log n)* time. It then calls on the *merge()* function to handle the conquer stage. For the *merge()* function, it iterates n times within which it performs a get on a priority queue in *O(log n)* time and access on a tuple in *O(1)* time - thus this stage takes *O(n log n)* time. Finally for setting the results in the correct order to *output()*, the algorithm again utilises a priority queue which gets values in *O(log n)* time. It does this twice, $\frac{n}{2}$ times, thus *O($\frac{n}{2}$ log n)* overall. Finally, the output is printed by getting elements from the priority queue where there are n elements. Thus this output stage is also *O(n log n)*
Therefore our final algorithm is:

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

$$= O(n\log n)$$