COMP232 – Data Structures & Problem Solving
Fall 2023

Homework #6 – Sorting
[50 points]

2. Consider each of the following computations on an initially unordered list of integer values:

a. Find the minimum value.

i. Initialize the minValue variable to the first element in the list and then traverse through the list and compare minValue to each element in the list if an element is smaller than minValue, updating the minValue with the value of this element. Repeating this process, and then after traversing the list, minValue will holds the minimum value in the list.

ii. The asymptotic upper bound on this algorithm's worst case running time will be O(n) since it requires traversing the entire list and updating the minValue if an element with the smaller value is found

b. Find the median (i.e., the middle value).

i. Sorting the list with the ascending order, and then if the list has odd number of the elements, the median will be the middle index. If the list has even numbers of the elements, the median will be the average of two middle elements

ii. The asymptotic upper bound on this algorithm's worst case running time will be O(n log n) since the get method to grab the median will be O(1) and for the sorting algorithm can be n log n.

c. Find the 10 largest values.

i. Using the min heap or priority queue for the initialization and select the first 10 elements in the list. Then for the remaining element in the list, compare it with the root of the min heap. After that, if that element is greater than the root, replacing the root with the new element and re-heapify. At the end, the min heap will consist of 10 largest values of the list. Then sort the list therefore, all the elements will be in the ascending order.

ii. The asymptotic upper bound on this algorithm's worst case running time will be O(n log n) since the initialization takes O(1) and for the insertion

and replacement of an element in a min-heap will be O(log n), for n elements, and then we will sort the list the running time will be O(n log n).

3. In our implementation of Insertion Sort we worked backward from the i-1st position while swapping to find the location at which to insert the ith value. This is essentially a linear search. However, because we know that the first i-1 values are already sorted we could have used a binary search to find the proper location at which to insert the ith value. Is this a useful idea? Why or why not?

It is not an useful idea even if the binary search reduces the time to find the position to O(log n), but the shifting still takes O(n) for each element. As a result, the running time will be total O(n) steps for each insertion. Hence, overally the running time still be O(n^2) which remains the same.

5. In an application where we need to sort lists that we know will already be nearly sorted indicate which sort would you expect to run faster and briefly justify your answer:

a. An unoptimized merge sort or insertion sort?

- The insertion sort is expected to run faster than un optimized merge sort for nearly sorted list since its running time is O(n) in the best case while the merge sort consistently runs O(n log n) regardless of the initial order.

b. An unoptimized merge sort or a heap sort?

- Both sorting algorithms above have the same performance since their running time is O(n log n) and they do not take any advantages of the current order of the data