**COMP232 – Data Structures & Problem Solving**

**Fall 2023**

**Homework #6 – Sorting**

**[50 points]**

1. Implement the insertionSort method in the CS232SortableLinkedList class in the hw06 code. This method should perform an insertion sort on the linked list and must run in O(n2) time. Note: You do not need to change any of the links, swap the elements in the nodes instead. The No1Tests class contains tests that you can use to check your implementation of this functionality.

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

a. Find the minimum value.

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

c. Find the 10 largest values.

For each computation above:

i. Briefly describe, in a few sentences or pseudo code, an efficient algorithm to perform the computation.

ii. Give and briefly justify an asymptotic upper bound on your algorithm’s worst case running time.

For cases b and c, we need to sort the list in ascending order (from the smallest to the largest value) before we can do anything else. This can be accomplished by performing merge sort. Here is the pseudo code for merge sort:

if len(A) > 1 then

half <- length(A) / 2

first <- A[:half]

second <- A[half:]

mergeSort(first)

mergeSort(second)

k <- 0 j <- 0 i <- 0

while i < length(first) and j < length(second):

if first[i] < second[i] then

array[k] <- first[i]

i <- i + 1

else

array[k] <- second[j]

j <- j + 1

k <- k + 1

1. We do not have to sort the list to find the minimum value. We can just initialize the minValue to be the first element in the unsorted list, and iterate through the entire list. For each element, we update the minimum value if the current element in the unsorted list is smaller. After we’re done traversing through the entire list, the value stored in the variable minValue is the smallest value. The worst case running time of this approach is O(n), as it only performs n – 1 comparisons. If we’d sorted the list and then grabbed the value of the first index instead, that would’ve taken nlogn operations.
2. To get the median, we just retrieve the value of the middle index in the sorted list. If the size of the list is odd, the median value is stored in the middle index; if the size is even, the median value is computed by taking the average the the two middle values in the sorted list. This is simply O(1). Sorting takes nlogn operations so the worst case running time is O(nlogn).
3. To find the largest 10 values, simply grab the last 10 values in the sorted list. This takes 10 operations, which is just constant time. Sorting is O(nlogn) so the upper bound time complexity is O(nlogn).

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?

While this idea could be useful in the sense it reduces the number of comparisons in finding the correct position for the new element to be inserted into the sorted portion, the actual insertion process still requires shifting elements to make space for the new element, which still dominates the cost. Shifting elements is O(n) in the worst case scenario, so the overall time complexity would still be O(n^2).

4. Complete the heapSort method in the HeapSort class so that given an array of integers they are sorted into descending order in O(n lg n) time using HeapSort. **You can run the main method to see if your sort works.**

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:

1. An unoptimized merge sort or insertion sort?

Insertion sort. Since the list is nearly sorted and most elements are in the right order, we only need to do a relatively small number of shifting operations. This is roughly O(n). Meanwhile, an unoptimized merge sort always has a time complexity of O(nlogn) regardless of how sorted the list is.

1. An unoptimized merge sort or a heap sort?

Merge sort. While both sorting algorithms have a worst case running time of O(nlogn), heap sort does not take advantage of the nearly sorted feature of the list, it would still build out the heap structure and repeatedly retrieve the max value. This means that heap sort would still swap the values of sorted portions while merge sort would not, as many sublists would already be in sorted order or close to it.