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 $O(n^2)$ 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.

a. To find the minimum value in an initially ordered list, we first create a variable to represent the minimum value. Then, we traversed the list to check for each value in the list and whether each of them was smaller than the initial minimum value. We update the initial minimum value with the found one if a smaller value is found.

b. To find the median (middle) value of an initially unordered list, we first sort the list in ascending order using merge sort. Then, we can divide it into 2 cases:
   - 1st case: The list has an odd number of elements: the median value can be computed by the average of the two middle elements
   - 2nd case: The list has an even number of elements: the median is the middle value.

c. To find the 10 largest values of an initially unordered list, we can first sort the list in descending order by using merge sort. Then we return the first 10 elements of the list.

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

a. By traversing through a list, the runtime will depend on the size of the list. We only need to check each element in the list once. Hence, the time complexity is O(n).

b. The process of sorting the elements in ascending order by using merge sort would take O(nlogn) running time. The step of computing the middle element of the in both cases where the number of elements of the list is even or odd

would take O(1) time. Eventually, the whole process would take O(nlogn) time.

c. The step of sorting the list in descending order by using merge sort would take O(nlogn) time. And the step of returning first 10 element of the list after sorting would take O(1). Eventually, the whole process would take O(nlogn) time.

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?

- Because the Binary Search take O(logn) time to execute, so it may sound more efficient than using Linear Search, which takes O(n) time. However, the primary cost in the Insertion Sort is shifting the elements to make space for new values since this algorithm compares each element in an unsorted list with the ones before it and swaps as needed to place it in the correct position among the sorted elements.
- Hence, the overall time complexity remains O(n^2). So using Binary Search is not a useful idea.

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:

a. An unoptimized merge sort or insertion sort?
   - The unoptimized merge sort would run faster because it would take O(nlogn) time at its worst case to execute while the worst case of the insertion sort would take O(n^2) time.

b. An unoptimized merge sort or a heap sort?
   - The Heap Sort may run faster. Although both Unoptimized Merge Sort and Heap Sort have O(nlogn) runtime, the Heap Sort will use less memory than the Merge Sort. To explain, the Merge Sort needs more space for multiple recursive calls and merging since this algorithm performs the same number of comparisons and merges regardless of how sorted the input is.