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.

```
minVal <- list[0]
    for I in range 1 to n(list length) - 1:
        if list[I] < minVal:
            minVal = list[I]
return minVal
```

O(n) because it goes through the list once

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

```
sort(list)
val <- length(list)
if length(val) is odd:
    return list[val/2]
else:
    return (list[(val/2) + 1] + list[val/2]) /2
```

First sort the list. Then find the middle index value by splitting list in half

c. Find the 10 largest values.

```
sort(list)
for I in range 1 to 10:
    list1.add[size-i]
return list1
```

The pre sort takes O(nlogn) and it's O(1) to add each element into the array. The overall run time is O(nlogn)

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.

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?

Binary search is more effective when trying to access elements at arbitrary indicies and takes constant time, but since we are using a linked list, to find the middle element we'd have to traverse through the linked lists which would take linear time.

4. Complete the `heapSort` method in the `HeapSort` class so that given an array of integers they are sorted into <u>descending</u> 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?

Insertion sort because it works better on nearly sorted lists since it requires shifting of elements. It would opperate in O(n) time now since it's doing next to no shifting

b. An unoptimized merge sort or a heap sort?

Merge sort is more efficient and has lower swaps counts. Heap sort doesn't benefit from the nearly sorted strucutre because it always involves building a heap and heapifying it. These takes does more swaps.