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

a.
```
min_value ← list[0]
for i in list do
  if x < min_value then
    min-value ← x
```

The asymptotic upper bound would be O(n) because we are iterating through the entire list.

b. findMedian(list):
```
insertionSort(list)
if odd then
  return middle element
else
  return average of two middle elements
```

insertionSort(list):
```
  for i ← 1 to list.size() - 1 do
    key ← list[i]
    j ← i
    currentVal ← list[j]
    while j > 0 & list[j] > key
        list [j+1] ← list[j]
        list[j] ← currentVal
        j—
    end while
```

The asymptotic upper bound would be O(n^2) because we are first sorting the array which will go through two loops when we have a list longer than one element.

c. **find10Largest(list):**

```
top10 ← []
insertionSort(list)
for i ← size-10 to size do
    top10.append(i)
    return top10
```

The asymptotic upper bound would be O(n^2) because we are again sorting through the array using insertionSort which has a Big O of n^2. But the for loop that appends the last 10 elements of the array to a list has a constant run time of 10.

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?

Using a binary search can reduce the runtime used to find the proper location at which to insert the ith value from O(i) to O(log i) because we are dividing the search interval in half. However, we would still need to shift the elements which would have a time complexity of O(i) and performing this act on all the elements would give us a Big O of n^2 for the entire algorithm. Since this is the same runtime as our initial implementation of Insertion Sort, I do not think this is necessary.

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 would be faster because its runtime in best-case scenarios where the list is sorted or nearly sorted would be approximately n since we would barely have to do any switching. However, the runtime for merge sort is always O(n log n) since it has to divide the list and then merge it back together in order.

b. An unoptimized merge sort or a heap sort?

   Although both methods has a Big O of n log n, heap sort is more taxing since you need to consistently check and alter the heap structure which can potentially make it slower.