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)
Assign the first element of the list to a variable A. Iterate through the list, if the current element is smaller than the A, change the value of A to that element. Return A after the completion of the iteration.

We iterate through the list, and for each iteration, we do a constant number of computational steps. Hence the upper bound of the function's runtime is O(n)

b)
We short the array using merge sort, heap sort …. (any thing that takes O(nlgn)). Then if the array length is 2K (K is non-negative integer), return the average value of the K^th^ and (K+1)^th^ element. Else, if the array length is 2K+1, return the value of K^th^ element.

We sorted the list, which takes O(nlgn). Assume we use array based list, then accessing K^th^ element is O(1), calculating the median take a constant amount of computational step. Hence the upper bound of the function's runtime is O(nlgn). Even if the list is linked list based, the upper bound is still O(nlgn) since accessing K^th^ element is O(n).

c)
We short the array using merge sort, heap sort …. (anything that takes O(nlgn)) in descending order. Then just grab the first 10 elements in the list.

We sorted the list, which takes O(nlgn). Getting Kth element is O(1) for array based, O(n) for linked list based list. Hence the upper bound of the function's runtime is O(nlgn).

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?

This is not a useful idea

If the list used linked list as the backing store, using binary search yield no advantage. Binary search is O(lgn) if and only if we can access an element at a constant time. Since accessing an element is O(n) in linked list, binary search in is case is no better than linear search.

If the list uses array backing store, using binary search won't benefit either. Even though we can find the correct spot for an element in O(lgn), moving that element to the spot takes linear time, which is proportional to the distance from that element to the spot. In short, there is no effective way, that takes O(1), to "insert" an element between 2 other elements in an array, as it requires shifting elements to create space. Insisting on using binary search on this case resulting in a running time of n*(O(lgn)+O(n)) = O(n^2)

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 will run faster, as if the list is nearly sorted, it will only iterate backward and swap the element a few times, so the runtime should be nearly O(n). On the other hand, merge sort will still break down and merge the lists as normal, taking O(nlgn).

b. An unoptimized merge sort or a heap sort?

They will have approximately the same runtime. Both unoptimized merge sort or a heap sort do not benefit from the nearly-sorted array. One still breaks the list down and merges it lgn times, resulting in lgn*O(n) = O(nlgn) runtime. One still builds the heap and do trickle down n times, resulting in n*O(lgn)= O(nlgn) runtime. However, in cases where a nearly sorted array has most of its elements identical, heap sort tends to be faster. That is because trickledown() run time is nearly Θ(1) for heaps that contain identical elements. In this special cases the run time of heap sort can be approximately n*O(1)= O(n).