Anastasiia Halchenko

COMP232 – Data Structures & Problem Solving
Fall 2024

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. Find the minimum value in an unordered list

```
minValue ← list[0]
    for i ← 1 to length(list) - 1:
        if list[i] < minValue:
            minValue = list[i]
    return minValue
```

The algorithm iterates through the list once so the time complexity is O(n).

If we use presorting:

```
sort(list)
minValue = list[0]
return minValue
```

The presorting takes O(nlogn) if we are using MergeSort and accessing the value is O(1). The overall time complexity is O(nlogn).

b. Find the median value

```
sort(list)
if length(list) is odd:
    return list[length(list) / 2]
```

```
else:
    return (list[(length(list) / 2) - 1] + list[length(list) / 2]) / 2
```

First we are sorting the list which takes O(nlogn) no matter what sorting algorithm we are using (merge sort, insertion sort, or

c. Find the 10 largest values

```
sort(list) //in descending order
for i←0 to i←10 do
    list2.add[1]
return list2
```

The presorting takes O(nlogn) if we are using MergeSort or heapSort. Then access and add the first 10 elements into a new array, which will take O(10) and it is constant. The overall 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?

Insertion sort takes O(n^2) because, for each element, it requires (n-1) comparisons and (n-1) swaps in the worst-case scenario (reversed order).
Using binary search to find the position where to place the element will reduce the search time to O(logn) instead of O(n). Even though the searching is faster, the shifting still takes O(n) in the worst-case scenario so the overall time complexity is O(n^2). But the average run-time would be improved from Θ(n^2) to Θ(nlogn).

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?
Insertion sort would run faster because Ω(n) for insertion sort requires minimum comparisons and swaps when an unoptimized merge sort always runs at O(nlogn), divides the array in two halves, and takes linear time to merge two halves no matter if the array almost sorted or not.

b. An unoptimized merge sort or a heap sort?

Both an unoptimized merge sort and a heap sort always run at O(nlogn). But if we are looking at the closer bounds, heap sort may take longer because it performs more constant operations such as swapping and trickling down.