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.
i. My algorithm for this part would just be to iterate through the list to find the min value. First, I initialize the variable min to hold the first element in the list. After that, I compare each element with min. If the current element is smaller, we update the min. At the end of the loop, min contains the smallest value

Pseudo-code:

```
min = list[0]
for i = 1 to n-1:
    if list[i] < min:
        min = list[i]
return min
```

ii. The algorithm iterates through the list exactly once so the time complexity should be O(n)

b.
i. For this task, I would use selection sort and access the median after that

Pseudo-code:

```
# Selection sort
function selectionSort(A):
    i <- 1
```

```
        for i <- 1 to n – 1 do
        min_j <- i
        min_x <- A[i]
        for j <- 1 + I to n do
                if A[j] < min_x then
                    min_j <- i
                    min_x <- A[j]
    # Accessing element
    function median(sortedList):
        if n is odd:
                median = sortedList[n/2]
        else:
                median = (sortedList[n / 2 – 1] + sortedList[n/2]) / 2
        return median

    main:
    sortedList = selectionSort(list)
    median = median(sortedList)
    print(median)
```

ii. Selection sort would require O(n^2) runtime complexity then I access the mid index, which runs O(1) time. So my algorithm would require O(n^2) in total

c.

i. My algorithm for this part would to do merge sort on the input list in descending order, then access the 10 largest element afterwards

Pseudo-code:

```
function mergeSort(list):
   if length(list) > 1 then:
      mid <- length(list) / 2
      left <- list[:mid]
      right <- list[mid:]

      mergeSort(left)        // Recursively sort the left half
      mergeSort(right)       // Recursively sort the right half

      // Merge the two sorted halves
      i <- 0  // Index for left
      j <- 0  // Index for right
      k <- 0  // Index for merged list

      while i < length(left) and j < length(right):
         if left[i] > right[j] then:  // Descending order
            list[k] <- left[i]
```

```
            i <- i + 1
        else:
            list[k] <- right[j]
            j <- j + 1
        k <- k + 1

    // Copy remaining elements
    while i < length(left):
        list[k] <- left[i]
        i <- i + 1
        k <- k + 1

    while j < length(right):
        list[k] <- right[j]
        j <- j + 1
        k <- k + 1
    return list

function getTenLargestValues(list):
    return list[0:10]

main:
    list <- [inputList]
    list = mergeSort(list)
    output = getTenLargestValue(list)
    print(output)
```

ii. MergeSort requires O(n log n) and extracting elements requires O(1) so my algorithm requires O(n log n) in total

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 very useful idea because it does not change a lot. This is because:
- In the traditional insertion sort, for each element i (from 1 to n – 1), we use linear search to find the position to insert in the sorted array then shift all larger elements in one position to the right to make room for the element being inserted ahead. Since both the search and shift operations take O(i) time for each i, it adds up to a sum which would create an O(n^2) runtime.

- If we use insertion sort with binary search, binary search reduces the time for finding position to insert the i^{th} element from O(i) to O(log i). However, the shifting elements to make room for the element with changed position still take O(i) time. So this binary search does not help with the shifting part, which dominates the time complexity. Therefore, the total runtime complexity is still O(n^2) if we are considering the big picture.

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?
An insertion sort should be faster in this case. This is because when using merge sort, the algorithm would keep moving and divide the array in half for each step no matter what. Therefore, the runtime would always be O(n log n) no matter in the best case, worst case or average case. Meanwhile, when using insertion sort, the runtime would not be that bad since the list is nearly sorted and we might not have to switch the element in every step. In other words, by using insertion sort, it runs in O(kn) for nearly sorted lists (k is the number of misplaced elements).

b. An unoptimized merge sort or a heap sort?

In terms of runtime complexity, nothing is better than the other. This is because the two algorithms would run O(n log n) time no matter what. So, if we want to deeply consider the efficiency of these two, we might also have to consider some different factors such as space complexity and caching.

In terms of space complexity, heap sort is O(1), meaning it only uses a constant amount of extra memory, as it is considered an "in-place" sorting algorithm, utilizing the input array itself for sorting without requiring additional data structures. Meanwhile, merge sort uses O(n) which means that the amount of memory it requires is proportional to the number of elements in the list. This is because merge sort requires an auxiliary array that is the same size as the input array to temporarily store the merged array.

Regarding caching, Cache locality refers to how well an algorithm takes advantage of the CPU cache. Merge Sort works by dividing the array into subarrays, sorting them, and merging them back. This process accesses elements sequentially, making it more cache-friendly and faster in practice, especially for large datasets. In contrast, Heap Sort reorganizes the array into a binary heap structure, which

involves jumping between non-contiguous elements in memory. This results in poorer cache performance and additional overhead for maintaining the heap property.