Problem 6.1 Bubble Sort & Stable and Adaptive Sorting

a) Bubble Sort pseudocode

```
BubbleSort (A, n)
   /* Outer loop to go through the elements of the array
      It goes till n-1
      No need to go till n because in the n-th iteration
      there is only one element waiting to be sorted,
      so as it is the only one in the unsorted part
      this means it is already sorted                    */
   for i = 0 to n-1
      /* There is no swapping in the beginning */
      swapped = false

      /* Inner loop for the number of comparisons
         It goes till n-1-i
         When there are n elements, we do n-1 comparisons
         Put also -i to avoid the unnecessary comparisons
         with elements that have already been compared with
         with each other in the previous iterations of
         the outer loop                                  */
         for j = 0 to n-1-i
         /* Compare the adjacent elements
            If the element in the position j is bigger than
            the element in the upcoming position, enter the
            if statement                                 */
         if A[j] > A[j+1]
            /* Swap them */
            swap (A[j], A[j+1])
            swapped = true // Swapping happens
         end if
      end for

      /* If no number was swapped it means
         the array is sorted now so break the loop. */
      if swapped == false
         break
      end if

   end for

end BubbleSort
```

b) Time complexity of Bubble Sort

Best case – The array is already sorted

In this case no swapping is done, so it breaks in the first iteration of the outer loop, which means that the outer loop is performed only once, when i=0. So, everything that is part of the outer loop is performed **1 time** (a constant). The inner loop itself iterates till the end → *n-1-i = n-1-0 = n-1*. As this is a loop part of the outer loop, in the whole code the inner loop will be performed:

1 (n-1) = **n - 1 times**

= O(n) → This is the time complexity for best case

Worst case – The array is in the reversed order

Outer loop iterates till the end → n-1, so everything that is part of the outer loop is performed **n-1 times**. The inner loop itself iterates till the end → n-1-i; 1-i is a constant so *n-1-i = n-c*. As this is a loop part of the outer loop, in the whole code the inner loop will be performed:

(n-1) (n-c) = n^2 - cn - n + 1

= n^2 + n (-c - 1) + 1 times

Now we sum up the times that every statement in the whole code is performed and find out that the time complexity for the worst case is **O(n^2)**.

Average case

This case happens when the elements of the array are sorted in such a way that there are performed half of the maximum swaps, so generally we can say that the outer loop iterates till n - n/2. Hence, everything that is part of the outer loop is performed **n - n/2 times**. The inner loop itself iterates till the end → n-1-i; 1-i is a constant so *n-1-i = n-c*. As this is a loop part of the outer loop, in the whole code the inner loop will be performed:

(n-n/2) (n-c) = n^2 - cn - n^2/2 + cn/2

= n^2/2 - cn/2 times

Now we sum up the times that every statement in the whole code is performed and find out that the time complexity for the average case is **O(n^2)**.

c)

- **Insertion Sort** → stable

The inner loop swaps the elements only if it is greater than the key:

```
while (j >= 0 && arr[j] > key)
    arr[j + 1] = arr[j]
```

It does not perform swapping if the element in position j is less than or equal to key, therefore Insertion Sort is stable.

- **Merge Sort** → stable

When merging the two sorted halves, the if statement is like this:

```
if (L[i] <= R[j])
  merged_arr[k] = L[i]
```

This way, if an element in the left subarray is equal to another element in the right subarray, we favor the element of the left subarray, so it goes first in the merged sorted array. Hence, Merge Sort is stable.

- **Bubble Sort** → stable

The if statement in the inner loop swaps the elements only if the element in position j is greater than the element in position j+1:

```
if A[j] > A[j+1]
  swap (A[j], A[j+1])
```

So, when two elements are equal, they will not be swapped and their order will remain unchanged. Hence, Bubble Sort is stable.

- **Heap Sort** → unstable

When sorting, during heapsort, the root is swapped with the last element (swapping non-adjacent elements). It goes like this: the element that has been picked first stays last, the element that has been picked second stays as the second last element in the sorted list, the third picked element stays as the third last element and so on. During these operations it happens that the relative order of equal elements is changed.

Example: We have this array A = {8, 6a, 6b, 5, 3, 1}

During heapsort the root 8 is swapped and put as the last element, then 6a is picked and placed as the second last element, then 6b is swapped and is positioned as the third last element and so on. In the end, after heapsort, the array looks like this:

A = {1, 3, 5, 6b, 6a, 8}

As we see, the order of the equal elements is changed, hence Heap Sort is unstable.

d)

- **Insertion Sort** → adaptive

If the array is already sorted or partially sorted, insertion sort reduces its total number of steps.

Time complexity for best case: O(n)

Time complexity for worst case: $O(n^2)$

As the time for these cases differ from each other, we can say that this algorithm benefits from the pre-sortedness in the input sequence and sorts faster. Hence, Insertion Sort is adaptive.

- **Merge Sort** → not adaptive

Time complexity for best case: O(n logn)

Time complexity for worst case: O(n logn)

Merge sort divides the array into subarrays recursively until it arrives to the base case which is 1. Regardless of the pre-sortedness of the array, the recursive calls and comparisons of this algorithm are performed logn and nlogn times, hence Merge Sort is not adaptive.

- **Bubble Sort** → adaptive

Time complexity for best case: O(n)

Time complexity for worst case: $O(n^2)$

We see that time complexity for best case and worst case are different from each other. In the best case the array is already sorted, so this algorithm benefits from the pre-sortedness in the input sequence and sorts faster ($O(n) < O(n^2)$). Hence, Bubble Sort is adaptive.

- **Heap Sort** → not adaptive

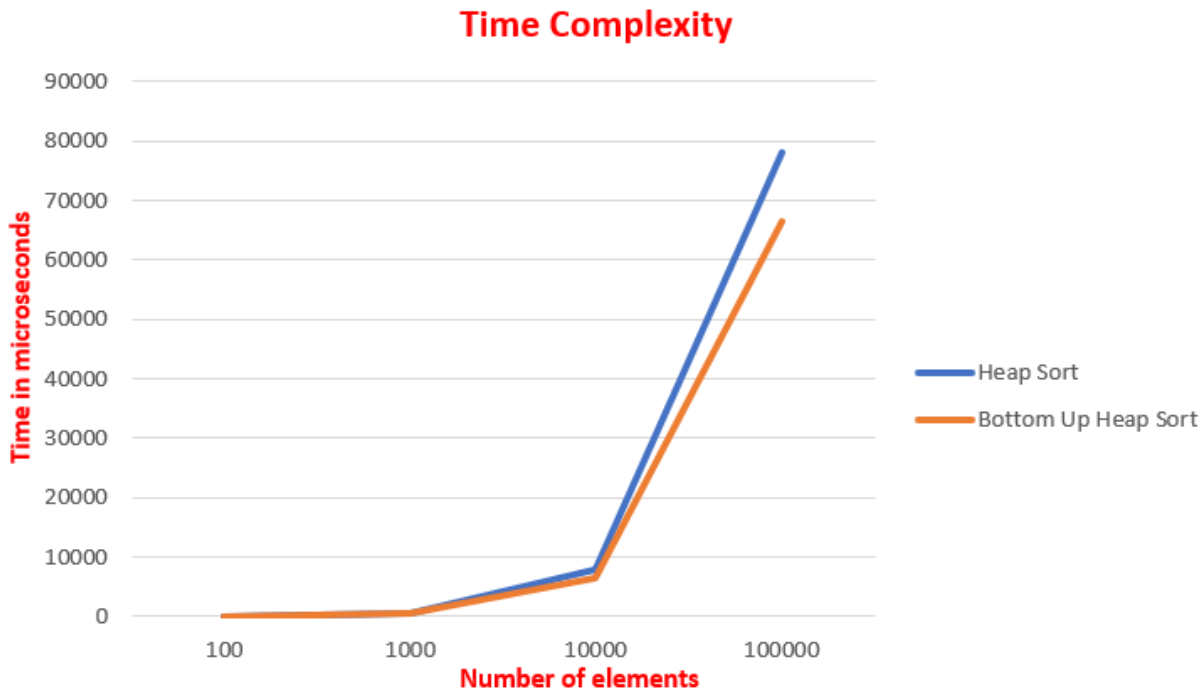
Time complexity for best case: O(n logn)

Time complexity for worst case: O(n logn)

Heap sort does not benefit from the pre-sortedness in the input sequence because Heapify and Build_Max_Heap methods destroy the order. Hence, Heap Sort is not adaptive.

Problem 6.2 Heap Sort

c)



I compared these two algorithms measuring the time that arrays with different number of elements take to be executed. As we can observe from the graph, Bottom-Up Heap Sort performs better than Heap Sort because in order to sort the elements of the array Bottom-Up Heap Sort requires fewer comparisons between the elements.