Homework 6 by Timofei Podlorytov

6.1

a)

Pseudocode:

```
BUBBLESORT(A)
    swap=1//we need to enter the loop
    while (swap)
        swap=0 //initially there are no swaps, value is 0
        for i=1 to A.length-1 //go through the array A
            if A[i]>A[i+1]//need to swap
                swap=1//we made a swap! This is marked my value 1
                c=A[i]//third variable for swapping
                A[i+1]=A[i]//swap
                A[i]=c
```

We exit the loop when there are no swaps needed when going through it, which can be checked by the value of swap remaining at 0. We go not until the very last element in the for loop since there's no next element for it.

b)

Best case we don't need any swaps and go through the array once and leave while loop. $\rightarrow$ $\Theta(n)$

```
BUBBLESORT(A)                                                          T
    swap=1//we need to enter the loop                                  1
    while (swap)                                                 unclear<(n)
        swap=0 //initially there are no swaps, value is 0              1
        for i=1 to A.length-1 //go through the array A                 n-1
            if A[i]>A[i+1]//need to swap                               1
                swap=1//we made a swap! This is marked my value 1      1
                c=A[i]//third variable for swapping                    1
                A[i+1]=A[i]//swap                                      1
                A[i]=c                                                 1
```

When going through the loop we can shift elements position only by one. Worst case we can have is when each element id the furthest it can be from its position. Reverse order satisfies that.
The longest distance then is n-1. So, we would need to perform at least $(n-1)(n-1)=n^2+2n+1=\Theta(n^2)$

For average case we can assume the average distance between the element and its correct spot is somewhere in the middle n/2. Then we would have $0.5n(n-1)=0.5n^2-0.5=\Theta(0.5n^2)=\Theta(n^2)$

c)

<u>Bubble sort.</u> For two elements to switch order they need to be compared and thus be adjacent at any point in the algorithm. However, the if condition only swaps 2 elements if one is smaller than the other $\rightarrow$ 2 equal element will be in the same order. So, it's Stable.

<u>Insertion Sort.</u> Imagine we have 2 elements one of which is already in the sorted part and the other is the current key, as it is the only case when their order can be changed. We insert the element after the first element which is not larger than it. Thus, we would insert it after the equal element, preserving the order. It's Stable.

Merge Sort. Here we have recursion. Imagine we have 2 equal elements. The switch in their order can only happen during the merging stage of the algorithm, when one element is in L and the other one is in R. Here we would go into the branch L[i]<=R[j] and the case would be L[i]==R[j], but since we choose to put the left element first the order is preserved and then the algorithm would take an element from R. Stable.

Heap Sort. Imagine we have a list [4 2 4’] when we create a max heap first we will get [4 2 4’] at build max heap step. Now we will try to sort by exchanging the first and last elements. [4’ 2, 4]. Now we still have a max heap and exchange again producing [2 4’ 4] != [2 4 4’](← if it was stable). Meaning it’s unstable!

d)

Bubble sort. If some part of the list is sorted or a list as a whole is sorted the already sorted part stays the same. I it’s sorted it’s $\Theta(n)$, meaning it’s adaptive as it takes advantage of some part being sorted.

Ex:

3 1 2 4→ 1 3 2 4→ 1 2 3 4 done-basically the sorted part just moved down

Insertion Sort. Let’s say we have a list that is sorted, then we don’t go into the while loop at all as the value is smaller than key. Meaning linear complexity. So, we can conclude that it’s adaptive.

Merge Sort. If the array is already sorted [1 2 3 4]. → [1 2] [3 4] → [1] [2] [3] [4]→ [1 2] [3 4]→ [1 2 3 4]

The execution time would stay the same as we only have for loops and thus still divide and merge even though it’s not needed. It’s not adaptive.

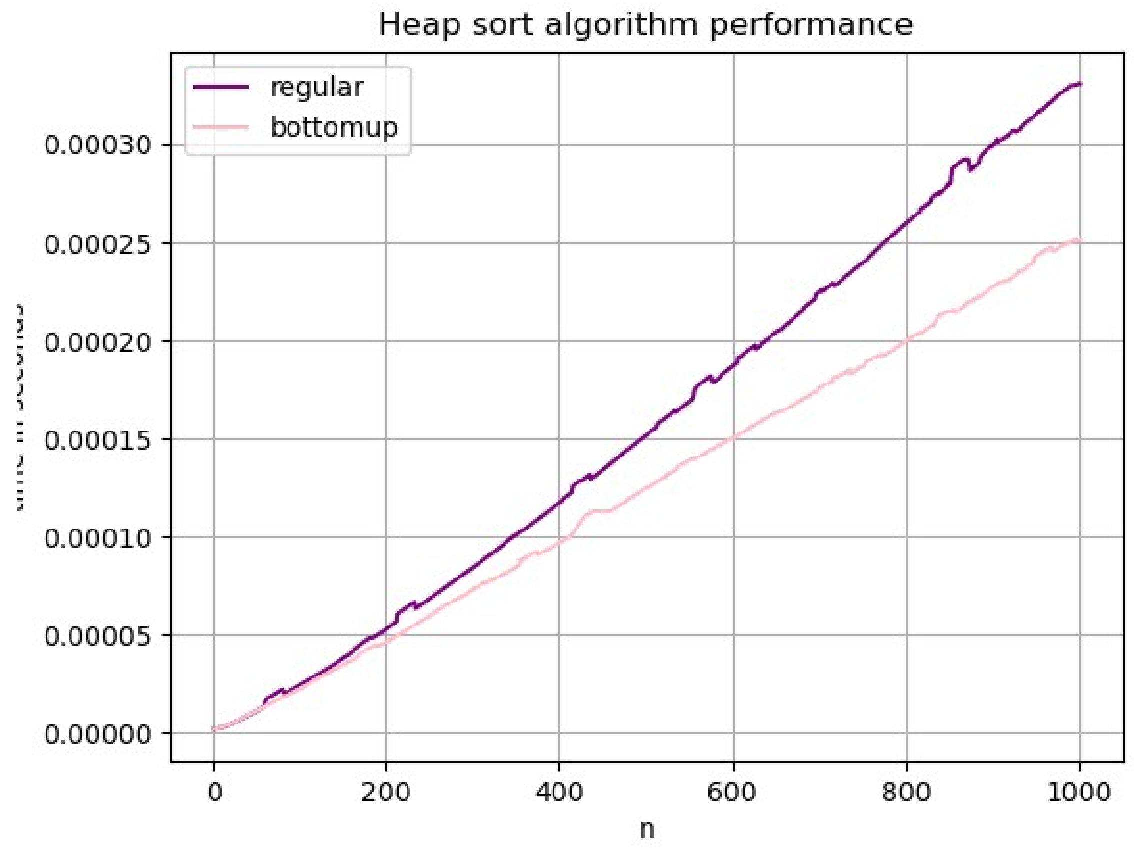
Heap Sort. Imagine we have a list [1 4 5] when we create a max heap we will move the elements even though it’s already sorted. [5 4 1]. And now we will need to sort it 🥲, despite it being sorted in the first place. So, It’s not adaptive.

6.2

a)-b) are implemented in the cpp file and have comments

c) I printed time measurement into txt file and read them in the graph.py file. Also, since the data is noisy, not smooth, due to performance of cpu not being perfectly consistent, I avegaged values near a pint in order to make it more clear. The results are below.

As we see the bottom up heapsort performs significantly better.


Heap sort algorithm performance
regular
bottomup
time in seconds
0.00000
0.00005
0.00010
0.00015
0.00020
0.00025
0.00030
0
200
400
600
800
1000
n