Assignment 5
Jovan Shandro

Exercise 1

a) The pseudocode for bubble sort is as follows:

```
//bubble sort algorithm to sort in ascending order
Bubble_sort(A)
  swapped = true
  while (swapped)
        swapped = false
        for i = 1 to length(A) - 1 inclusive do:
              //compares every adjacent pair of array elements
              if A[i-1] > A[i] then
                    swap(A[i-1], A[i]) //swap the two elements in order for them to be  asc order
                    swapped = true    //since we had at least one swap then turn swapped to true
                                            //so that the loop continues the checking
              end if
        end for
  end while
```

b) We will show that the asymptotic time complexities are $O(n^2)$ for worst and average cases and O(n) for the best case. We start with the proof for the best case. In the best case, the array is already sorted. So the for loop will run only once to check that the array is sorted and then the while loop ends, so since only n-1 comparisons are made and knowing that each of them requires a constant time the time complexity is linear O(n). Now we consider the worst case. In this scenario the array is inversely sorted. In this case the while loop will run n-1 times as all the first n-1 elements must be switched with the adjacent elements until the i-th element goes to the n-i+1 position so since each for loop will run n-1 times and the for loop itself requires a linear time as it iterates through all the array thus the algorithmic time complexity is $O(n^2)$ as the equation of the complexity becomes $\sum_{i=0}^{n-1}(\sum_{j=0}^{n-1} O(1)) = O(n^2)$ *. Now we consider the average case. In this case we can consider a random permutation of the n elements. In average case we can suppose the only one half is yet unsorted, suppose the first and thus the while loop will run n/2 times and so the algorithmic time complexity would be* $\sum_{i=0}^{n/2-1}(\sum_{j=0}^{n-1} O(1)) = O(n^2)$ *thus again $O(n^2)$.*

c) Insertion sort, bubble sort ,and merge sort are stable whereas heap sort is not.
Insertion sort swaps only adjacent elements thus just adjusting the comparison sign so that the numbers are swapped only when they are different and not equal (for ex using < and not <=) thus equal elements remain in same order; same logic applies to bubble sort since it only swaps adjacent elements and using same way equal elements remain in their initial order compared to each other. Merge sort can become stable if we make it such that when comparing the elements of left and right arrays, the left array has higher priority for equal elements so the elements of equal sizes remain in same order compared to each other. Considering heap sort, we can pick an example in which 2 elements of same size remain in different sides of the root element and the order in which they will be in the sorted array depends in the other 'nodes' of the heap. An example is the entered array [1, 7, 3_1, 4, 3_2, 9] and the sorted version will be [1, 3_2, 3_1, 4, 7, 9] according to the implementation discussed in class.
d) It is very easy to see that insertion sort and bubble sort are both adaptive which can be directly seen from the fact that their time complexities get better when they are already sorted; they are

adaptive as they make swaps only with adjacent elements thus when a certain part is already sorted they only have to check if they do have to make any swap or not. Merge sort can be made adaptive as for example in the implementation I made in java, in case the largest element of the left array is smaller than the smallest element of the right array then no merging is done as they are already 'merged'. This surely does not change the time complexity , but still improves running time. Considering our version of the heap sort, it is not adaptive as the order of the array does not help a lot as whatever it is we will still heapify the array again and again as we take the root and put it in positions from the last in the sorted array. Whenever we take out the root node we call heapify to make sure the largest elem is in the root node and the process is not helped by pre-sortedness. Even if the array is fully sorted, after building the max heap it won't be anymore.

Exercise 2.

c) As we also see from the graph (which checks the time for arrays with random numbers) we see the the algorithms perform similarly but as the size of the array gets bigger the heap sort implementation of point b performs faster as also expected and performs with almost linear time complexity. Since heap sort is not adaptive, I considered only the average case scenario as the time complexity won't change much in the so called 'best ' and 'worst' case scenarios. So we observe that as n grows bigger the second version is faster.