a) Write down the Bubble Sort algorithm in pseudocode including comments to explain the steps and/or actions.

Pseudocode:

```
1   BubbleSort(Array, N)
2   {
3   int check = 0;
        4   for(int i=0; i<N-1; i++)
        5   {
                6   for(int j=0; j<N-1-i; j++)
                7   {
                        8   if(Array[j] > Array[j+1])
                        9   {
                                10  swap(Array[j], Array[j+1]) ;
                                11  check = 1;

                        12  }

                        13  if(check == 0)

                                14  Break;

                15  }

        16  }

17  }
```

Explanation:

1. // There is an *Array* with *N* elements to be sorted

3. // Declaring a variable to check whether the swapping function is running or not. If not then *break* and return the array as it is (already sorted). Using this for the Best Case.

4. // Taking *i<N-1* as the i always sees for the next element to *swap* with, so it will bring an error when we come to the last element as there won't be any element to compare with.

6. // After the first iteration the last element will be in its correct position so we do not have to include/use it again that's why we take *j<n-1-i* or just *j<n-2*

8. // The condition to *swap*. If the previous element is greater than the one after, than *swap* their positions.

10. // The *swap* function to change the positions of elements. It can also be done with the help of another variable *temp* if swap function not included in the language used.

11. // Assign the value 1 to the *check* variable if the *swap* function runs.

13. // If the *check* variable has not changed after the *for* loops, or in other words the *swap* function did not run then *break* the *for* loops as the array is already sorted so no need to continue going through elements of *Array*.

b) Determine and prove the asymptotic worst-case, average-case, and best-case time complexity of Bubble Sort.

Worst Case:

The worst case happens when the elements of the array are in reverse order. So each one of them have to change position. As all the elements have to change positions then both of the two *for* loops will run and we can also assume that the *swap* function will run constant *c* times. As the *for* loop runs *(N-1)* times & also the other for loop runs *(N-1)* times & the function is executed some constant *c* times; we can come up with the time complexity:

T(n) = (N-1) * (N-1) * c = $cN^2 - 2cN + c$ => The worst-case time complexity is ***O(n²).***

Best Case:

The best case happens when the elements are already sorted. This means that the *swap* function will not run as there are no elements which should change positions and the *for* loops will also break. As only one *for* loop will run the time complexity of the best-case is ***O(n).***

Average Case:

The average case happens when the elements are in random order. By having the elements in random order, both of the two *for* loops will have to run and the *swap* function will also run some constant *c* times. So again, as the time complexity of the worst-case, in average-case we will have:

T(n) = (N-1) * (N-1) * c = $cN^2 - 2cN + c$ => The average-case time complexity is ***O(n²).***

Note: ***T(n) = (N-1)*(N-1)*c*** can also be solved with the recursion tree/substitution method but as we can see from the highest exponential coefficient ***N²*** I think it is not necessary to include them.

c) Which of the sorting algorithms Insertion Sort, Merge Sort, Heap Sort, and Bubble Sort are stable? Explain your answers.

A sorting algorithm is stable if – *i<j & array[i] == array[j]* implies *€(i) < €(j)* where € is the sorting permutation (moving *array[i] to €(i)*). Simply equivalent elements retain their relative positions after sorting.

Insertion Sort, *Merge Sort* and *Bubble Sort* maintain stability by ensuring that *array[j]* comes before *array[i]* if *array[j] < array[i]*. Here *i* and *j* are indices and *i<j*. Since *i<j* the relative order is preserved if *array[i] == array[j]*. The condition *swap* if and only if one is strictly less than the other, otherwise let them be as they are makes this three algorithms stable algorithms.

Heap Sort begins by extracting the maximum number from the max heap which is the first element and then putting it on the last position. Any information about the ordering of the items in the input was lost during the heap creation stage. The final output in heap sort comes from removing items from the current heap in an order.

i.e.: *array*[31*, 31**, 19, 27] -> *array*[31**, 19, 27, 31*] -> *array*[19, 27, 31**, 31*]

This is the reason why Heap Sort is not a stable algorithm.

d) Which of the sorting algorithms Insertion Sort, Merge Sort, Heap Sort, and Bubble Sort are adaptive? Explain your answers.

A sorting algorithm is adaptive when the order of elements to be sorted of an array affects the time complexity of the sorting algorithm, so when it takes advantage from the input order.

Insertion Sort and *Bubble Sort* are both adaptive algorithms. If the input is already sorted then the *for* loop only run once and nothing happens (i.e.: no swapping). The time complexity for both algorithms Insertion and Bubble Sort is ***O(n)*** in this case (best-case).

Merge Sort and *Heap Sort* are both non-adaptive algorithms. The input order does not affect the algorithm time complexity at all as it is still be going through all the *for* loops. The time complexity for both algorithms Merge and Heap Sort is ***O(nlgn)***.