Problem 6.1

a)

pseudocode:

```
Bubble_Sort (Array, n)              // n-number of elements in the array
    k = n-2                         // elements in index bigger than k+1 are sorted
    While count != 0                // count- number of swaps
        For i=0 to k                // only need to sort up to k+1 elements
            count=0                 // count needs to be set to 0 in the beginning of every
                                    //    loop
            If Array [ i ] >Array [ i+1 ]
                Swap Array [ i ] with Array [ i+1 ]
                count++             // if any swap is made, it not sure that the array is sorted
                                    // which requires another loop sort from beginning
            i++                     // index of number needs to be compared with next one

        k--                         // after every for loop, one more element is sorted to
                                    // the right place at the end of the array
```

b)

For the worst case in bubble sort, it is a reverse-sorted array, which will have n-1 swaps in the first loop, and then n-2 until 1 swap in the beginning of the array. Therefore, by summing these terms up, we get

(n-1+1)*(n-1)/2,

which is O(big O)(n^2).

For the average case, it is quite similar with the worst case, because the loop stops when there are no more swaps happening. Thus, the sum of comparison between elements is

(n-1+1)*(n-k)/2,

where k is the number of elements in the correct position in the original array.

This gives us O(theta)(n^2)

For the best case, the array has already been sorted, which only needs one loop of element comparison. Thus, this is just O(big Omega)(n).

c)

Insertion Sort: Stable

Insertion Sort works from the beginning of the array to the end, and each element swap places with the previous one if it is smaller until it is larger than or equal to the previous element. Thus, two same elements will not be swapped, and stability is kept.

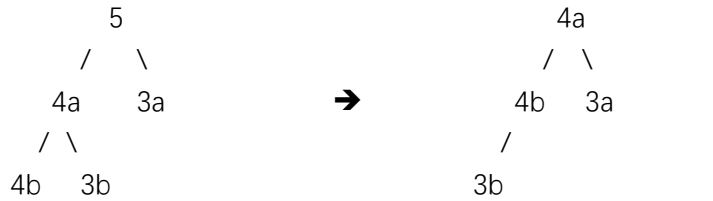
Merge Sort: Stable

Merge Sort uses divide and conquer method that divides base cases into one element first, in which process the order of the elements are not changed, because the comparison of two branches starts from the left branch.

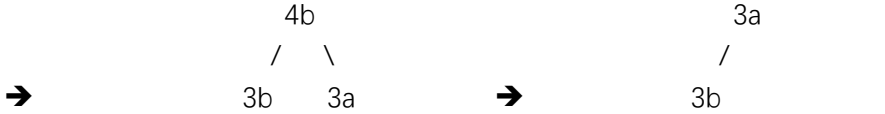
Heap Sort: Unstable

Counter example: Consider a Max Heap (5, 4a, 3a, 4b, 3b)

(5, 4a, 3a, 4b, 3b) remove 5 (4a, 4b, 3a, 3b, 5)



Remove 4a (4b, 3b, 3a, 4a, 5) remove 4b (3a, 3b, 4b, 4a, 5)



Actual sorted array

Remiove 3a (3b, 3a, 4b, 4a, 5) expected sorted array

3b (3a, 3b, 4a, 4b, 5)

➔ This proves Heap sort is unstable.

Bubble Sort: Stable

Bubble Sort compares adjacent elements in loops. In each loop, the largest element or the smallest, depending on sorting order, is going to be sorted in the end (beginning) of the array. When two same value elements are compared, they will not be swapped. Thus, the element has bigger index number remains bigger than its other same-value numbers.

d)

Time Complexity	Worst-case	Average-case	Best-case	Adaptivity
Insertion Sort	O(n^2)	θ(n^2)	Ω(n)	Adaptive
Merge Sort	O(n log(n))	θ(n log(n))	Ω(n log(n))	Not adaptive
Heap Sort	O(n log(n))	θ(n log(n))	Ω(n log(n))	Not adaptive
Bubble Sort	O(n^2)	θ(n^2)	Ω(n)	Adaptive

In order to tell whether a sorting algorithm is adaptive or not, all need to see is to check if time complexity changes for different scenarios cases.

According to the time complexity chart above, Insertion Sort and Bubble Sort are Adaptive, because their worst-case and best-case have different time complexity, which implies they benefit from the pre-sortedness in the input sequence and therefore sort faster.

On the other hand, Merge Sort and Heap Sort do not change time complexity for every scenario. Thus, these two sorting algorithms are not adaptive.

Problem 6.2

a)

Heap_Sort.cpp

b)

Variant.cpp