Problem 6.1

a)

```
    BubbleSort(A, n)
// A is the array to be sorted, n is the number of elements in A
repeat
   swapped ← false  // Initialize swapped flag to track swaps
   for i ← 0 to n-2 do  // Iterate through the array
      if A[i] > A[i+1] then  // Compare adjacent elements
         swap(A[i], A[i+1])  // Swap if they are in the wrong order
         swapped ← true  // Set swapped flag to true
   end for
until swapped = false  // Stop if no swaps occurred in the last pass
```

b)

Worst-case time complexity O(n^2) occurs when the array is sorted in descending order.

Average-case time complexity O(n^2) since half of the elements have to be swapped.

Best-case time complexity O(n) occurs when the array is already sorted.

c)

Insertion Sort is stable since it only swaps elements when necessary, and equal elements remain in their original order.

Merge Sort is stable because it maintains the relative order of equal elements by merging them in the same sequence as they appeared in the original lists.

Heap Sort is unstable since the heap structure does not preserve the relative order of equal elements due to its tree-based reordering.

Bubble Sort is stable since it swaps adjacent elements without changing the order of equal elements, making it stable.

d)

Insertion Sort is adaptive. If the input is nearly sorted, it runs in O(n) because it only requires minimal swaps.

Merge Sort is not adaptive since it always divides the array and merges it back, regardless of whether it was initially sorted, making its time complexity always O(n log n).

Heap Sort is not adaptive since it builds a heap and performs heapify operations, which do not take advantage of any existing order in the input.

Bubble Sort is adaptive since if the input is already sorted, it runs in O(n), as it will make only one pass.

Problem 6.2

d)

Bottom-up Heap Sort is slightly faster as it avoids unnecessary checks when moving elements downward but in cases where an element need to be moved back up frequently the time difference is very minimal.

Both algorithms have a O(n log n) time complexity.

The difference of the two algorithms becomes noticeable when the input size grows. The bottom-up version benefits when elements naturally move downward without too many adjustments.