Problem 4.1

a) Bubble sort pseudocode

```
1.  Swap ← true                 //variable to track if any swap was performed
2.  While Swap = true           //repeat loop until no swaps are required
3.      Swap ← false            //when no more swaps are required
4.      For j ← 1 to (n-1)      //loop (n-1) times where n= size of array
5.          If A[j] > A[j+1]    //check if larger element is in wrong position
6.              Swap_elements(A[j], A[j+1])     //swapping two positions
7.              Swap ← true                     //tracking swap operation
```

b)

1) **Worst Case:**
This means the input array will be in descending order. In the pseudocode above, there are 2 loops (line 2 and 4), and the last element of the array will first be swapped with first element, in (n-1) loops. This swap operation will be done for n loops while loops) and the rest of the other operations have constant timing. Hence, worst case time complexity: $= O\big(n.(n-1)\big) = O(n^2)$. Example array, {5,4,3,2,1}

2) **Best Case:**
Best case is when the input array is already sorted. This means that if condition in line 5 is never satisfied and *for* loop executes *(n-1)* times, and swap becomes false. So, the while loop terminates after the first loop itself. Hence, the time complexity: $O(n-1) = O(n)$.

3) **Average Case:**
We consider a uniformly distributed array(array where drawing each element is equally probable). Taking different examples to perform the analysis, let's do the sorting for 2 arrays {1,3,2} and {4,2,1,3}. Applying the pseudocode to these examples, we can see that while loop iterates n/2 times(in average). Considering equal probability of each possible pair from any array, we get = nC2= n(n-1)/2 possible swap operations. We can say, time complexity is $O\left(\frac{n(n+1)}{2}\right) = O(n^2)$.

c) **Stable sorting algorithms:** These are algorithms that maintain the relative order of records with equal values as of input. For example, in input, we have {apple, peach, marsh, straw, panes} and we are sorting according to first letter, if we get {apple, marsh, peach, panes, straw}, it is stable sorting. This is because the relative order of peach and panes don't change even though the entire word alphabetic would mean we'll have panes before peach. But, the sorting doesn't interchange the positions of same keys (in this case, 'p'). This kind of algorithm is known as stable sorting algorithm.

Now, let's analyze the given 4 algorithms to see if they are stable:
i**) Insertion Sort:** *Stable sorting* as it doesn't change relative order of elements with same key. In pseudocode of insertion sort, we have .. *while j > 0* and *arr[j] > key, swap*…
For an array of *[..,L,R,..]* , let's say current index = L, then it places L in the right position in the left sorted sub-array. Then, if R and L have same values, R will be placed in the position L+1. Ultimately, we get the relative order of L and R unchanged.
Example, {4,3,2,2',10,12,1}, 2 and 2' are the elements with same value but 2' is used to distinguish it. After insertion sort, we get the sorted array in these steps:
{3,4,2,2',10,12,1}
{2,3,4,2',10,12,1}
{2,2',3,4,10,12,1}
{1,2,2',3,4,10,12} //final sorting, relative order of 2 and 2' hasn't changed

ii) **Merge Sort:**
Stable sorting if the merging section of algorithm has this line :
if (L[i] <= R[j])
arr[k] = L[i]
If properly implemented, this merge section will favor the left side value over the right side when the values are equal. This means, left side value will be placed previously in terms of index, implying that the relative order doesn't change as left side value will always be in left relative to the right side value.

Iii) **Heap Sort:**
Unstable(mostly)
When we build a *MaxHeap,* the initial positions of elements are lost. And, we sort by removing/placing the largest element at last. This process is done recursively, and it requires calling *MaxHeap* frequently and the relative order of elements get distorted. However, an exception is that sometimes, there might be instances where the relative order is not distorted. This is determined by where the element lies in the heap. But, in general, we can say that heap sort is unstable because the elements' relative positions are not maintained in all cases. Eg, an array : {21,20',20,12,11,8,7} will result in {7,8,11,12,20,20', 21}.

iv) **Bubble Sort**
In problem 4.1 a, line 5*: if A[j] > A[j+1] … swap*..
It implies that swap occurs only if the current element is strictly greater than the next element. If they are equal, no swap occurs and their original relative positions are maintained. So, it is clearly stable sorting. Eg, an array: {4,4',5,1,2} results in {1,2,4,4',5}.

d) **Insertion Sort:** *Adaptive*. The best case time complexity of insertion sort is $O(n)$, and average case is $O(n^2)$. This is because the while Arr[j] > key operations are mostly skipped for sorted parts in insertion sort. So, having a sorted array reduces the time complexity for insertion sort, hence it is adaptive.

Merge Sort: *Not adaptive*. Irrespective of the fact that the array is sorted or unsorted, in the merging section of merge sort, each comparison is performed, so there is not much difference in time complexity. Additionally, the time complexities for best, average and worst case are the same: $O(nlogn)$.

Heap Sort: *Not adaptive*. The number of times we call *MaxHeap* doesn't change, irrespective of the sortedness of the array. So, it builds a *MaxHeap* every time and the original structure of array is changed anyway. That is also why, the time complexities for best, average and worst case are the same: $O(nlogn)$.

Bubble Sort: *Adaptive*. As already explained in problem 4.1 b, the time complexity for worst case and average case is $O(n^2)$ and for best case is $O(n)$. The number of swaps required decreases due to the sortedness of the array, and the number of total loops performed is also decreased. That's why the time complexity greatly decreases for best case or near to best cases.

Problem 4.2

a) Code in heapsort.c (Makefile1.txt)
b) Code in adj_heap2.c (Makefile2.txt)
c) Online compiler was used to take time computations for unbiased time computations. And, as the values are randomly generated, values for the same n were taken multiple times and averaged it (manually as keeping a loop in the program gave biased values). Raw data for final values of plot are in Excel file "4.2b data". Then, gnuplot was used to draw graphs. The values were taken from n = 1 to 2000, and I have 2 graphs from same data. Graph explained below.

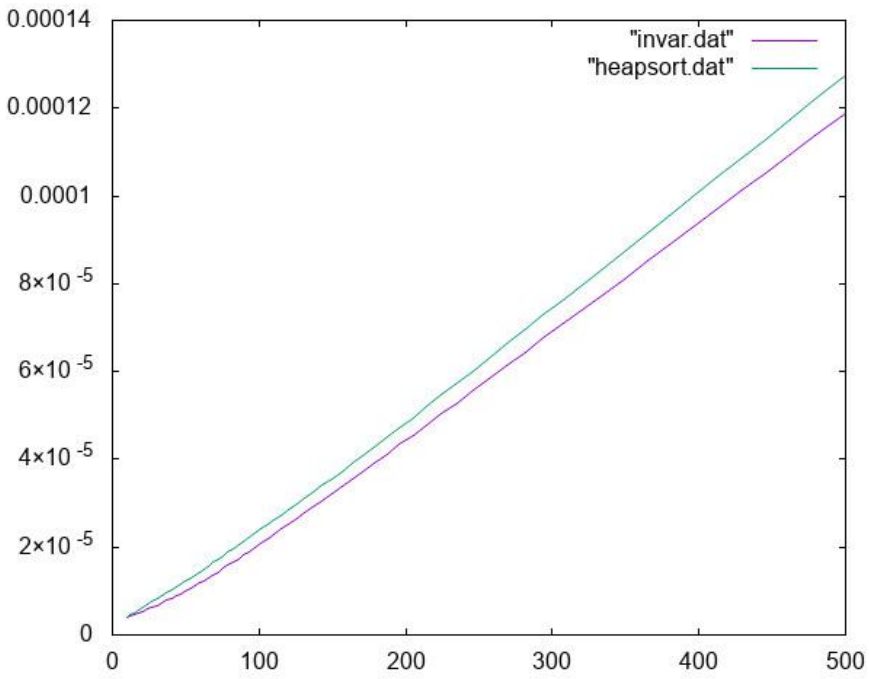

Figure 1: For a closer look, graph from n= 1 to 500.

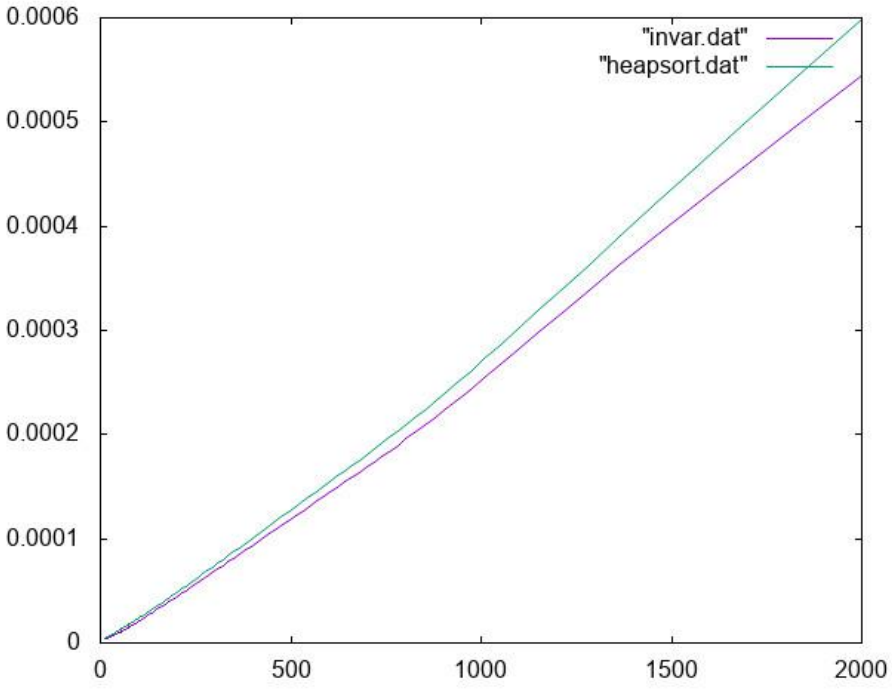

Figure 2: For a larger range, graph was taken for n = 1 to 2000.

From the plot, we can clearly see that the time taken for normal heapsort(green line/curve) is more than the one with heapsort variant(purple line). The time difference can be noticed clearly as n approaches larger values. This is because we don't build a maxheap structure after every loop (starting from second step) in the invariant function. And, sometimes, we have to do lesser operations when the floated root is very small and doesn't disrupt the heap. Even when it disrupts the heap, we don't have to check every sub tree to modify the heap, and we can just follow back the special path as mentioned in the question, and checking heap properties stop when the parent's value is no longer smaller than its children's. So, we can say that the invariant sorting performs relatively better than normal heapsorting (for larger n's) and especially for arrays which have smaller values in new roots after first Maxheap step (which is the same in normal heapsort).