**Algorithms and Data Structures Homework 6**

**Problem 6.1**

1. Bubble Sort pseudocode with explanations

The following algorithm is designed to sort in ascending order.

BubbleSort(arr)

Swapcheck = true

// to make sure that the algorithm runs through the entire array till there is nothing left //to sort and hence to terminate the loop once the array has been sorted and there are no more //swaps to be done or in other words check that a swap has been made and correspondingly //sorting is being done till everything has been sorted

while (Swapcheck)

Swapcheck = false // no swaps yet

for i = 0 to length(arr) do

// iterating and comparing adjacent elements of the array

If (arr[i-1] > arr[i]) then

swap (arr[i-1],arr[i]) // in order to achieve ascending order

Swapcheck = true // so that the loop continues comparing and //sorting since we had a swap

end if

end for

end while

Here, “n” is the array size. The worst and average case time complexities for bubble sort are O(n2) for both cases. For best case, however, the time complexity is O(n). In the best-case scenario, the array is already sorted. The algorithm will run through the array once and then the while loop ends after the for loop has iterated through the array and compared the elements and checked that the array is already sorted. It takes n-1 comparisons during the iteration, so the time complexity is linear hence O(n). Now considering the worst-case scenario(where the array is inversely sorted), the while loop will run n-1 times as each element has to be swapped with its adjacent n-1 times and since the for loop also requires n-1 iterations each for each time the while loop runs giving as O(n\*n) = O(n2) time complexity. For average-case, let us take the scenario of a random permutation of array elements and that half of this array is not yet sorted so the while loop will run n/2 times still resulting in O(n2) time complexity (n/2\*n/2=n2/4, ignore ¼ as it is just a constant number resulting in n2 hence O(n2)).

**C)**

Insertion sort and bubble sort use swapping to sort an array so the order is preserved since only when the elements are different or less or greater are they swapped and not when equal(the algorithms should be implemented to swap when strictly less or strictly greater not equal) so equal elements keep their original order. Merge sort is stable given that it is tweaked meaning it is implemented so that when merging left half elements are preferred over right half when they are equal by using the relationship leftelement<=rightelement. Heap sort is not stable because the order depends on the position of the nodes of the equal elements which is destroyed while building the heap which might lead to a different order. Insertion sort, bubble sort and merge sort are stable sorting algorithms while heap sort is not.

**D)**

Insertion sort and bubble sort are adaptive because they only swap when the elements are not in the right order but in case of presorted sections, they only check whether a swap is needed or not and hence their running times get better. Merge sort can be adaptive in the sense that it is not swapped if part of the array is presorted or copied into an array that holds the sorted version of the original array which depends on implementation but still if it is part presorted array, since either copy or swap implementations are not performed, running time is improved so merge sort is adaptive. The version of Heap sort we learnt is not adaptive because we call max-heapify multiple times during the execution of the heap sort algorithm which disrupts the presorted array and hence destroys any chance of benefitting from the presorted array in improving running time.