Assignment 6

Nayan Man Singh Pradhan

Problem 6.1

a.

Bubble Sort Pseudocode

```
int arr[n] //an array of n elements
for (int j = 0; j < n-1; j++) { //a loop that going till n-2 because n-1 will be largest by then
        for (int i = 0; i < n-1; i++) { //a loop that iterates from the 1st elem to the 2nd last elem
                if (arr[i] > arr[i+1] { //if the previous term is greater than the next
                        swap (arr[i], arr[i+1]) //perform a swap
                }
        }
}
```

b.

Worst-case

The worst case for a bubble sort algorithm is when the array is sorted in reverse order. This is because the if condition is always going to be true for each iteration of the loop and a swap will always be performed in each iteration.
Therefore, an array sorted in reverse order is the worst case for a bubble sort algorithm.

Best-case

The best case for a bubble sort algorithm is when the array is already sorted. This is because the if condition is always going to be false, i.e. the next term is always going to be greater than the previous term, and never execute. Hence, the swap is never going to perform.
Therefore, an already sorted array is the best case for a bubble sort algorithm.

Average-case

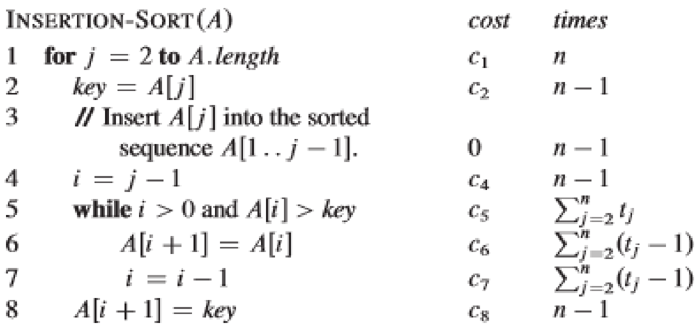
The average case for a bubble sort algorithm is when the average of 'n' number of randomly generated arrays is sorted. The arrays should be randomly generated as sometimes the array might be favoring the best-case scenario, while, in other cases, it might be favoring the worst-case scenario.
Therefore, the average of a randomly generated array is the average case for a bubble sort algorithm.

I have attached a c++ file called "bubbleSort.cpp" that contains an example for the three cases.

c.

Stable sorting algorithms maintain the relative order of records with equal keys. Thus, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list. For example, Let us assume an array of elements: { 4, 1, 3, 2, 5, 3, 6, 9, 7 } with number 3 repeating twice. I have represented the two occurrences with red and blue color to show my example. A stable sorting algorithm is going to sort the array as : { 1, 2, 3, 3, 4, 5, 6, 7, 9 } with the red three in front of the blue three, while an algorithm that is not stable may sort the array as : { 1, 2, 3, 3, 4, 5, 6, 7, 9 }. Therefore, the order of occurrence is maintained by the stable sorting algorithm.

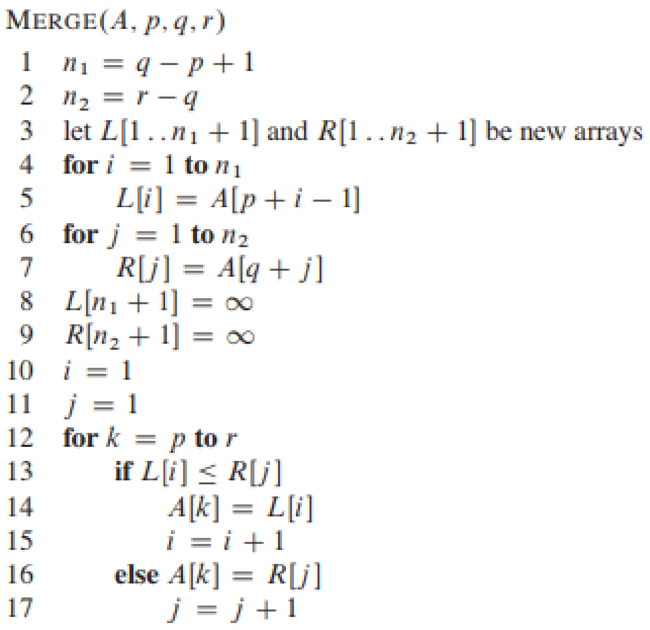
An insertion sort algorithm **is a stable sorting algorithm**. Image l represents a general pseudo code for the insertion sort algorithm. In line 5, we can see that a swap (in line 6) takes place only if the element that is being compared is greater than the key. If the element and the key are the same, they are not swapped and hence, the order of occurrence is maintained.



```
INSERTION-SORT($A$)                                      cost      times
1  for $j = 2$ to $A.length$                              $c_1$     $n$
2      $key = A[j]$                                       $c_2$     $n-1$
3      // Insert $A[j]$ into the sorted
           sequence $A[1..j-1]$.                          $0$       $n-1$
4      $i = j-1$                                          $c_4$     $n-1$
5      while $i > 0$ and $A[i] > key$                     $c_5$     $\sum_{j=2}^{n} t_j$
6          $A[i+1] = A[i]$                                $c_6$     $\sum_{j=2}^{n} (t_j - 1)$
7          $i = i-1$                                      $c_7$     $\sum_{j=2}^{n} (t_j - 1)$
8      $A[i+1] = key$                                     $c_8$     $n-1$
```

Image 1. Pseudo code of insertion sort

A merge sort algorithm **is also a stable sorting algorithm** (if implemented correctly). Image 2 represents a general pseudo code for a merge sort algorithm. In line 13, the if condition runs only if L[i] <= R[j] (if L is smaller or equal to R). This shows us the when the algorithm is merging the two halves of the list, the left part is given priority than the right part, hence the order of occurrence is maintained. Note that, in this exact case, implementing the algorithm as L[i] < R[j] (strictly smaller than) would result in an unstable algorithm.



```
MERGE($A, p, q, r$)
1   $n_1 = q - p + 1$
2   $n_2 = r - q$
3   let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
4   for $i = 1$ to $n_1$
5       $L[i] = A[p + i - 1]$
6   for $j = 1$ to $n_2$
7       $R[j] = A[q + j]$
8   $L[n_1 + 1] = \infty$
9   $R[n_2 + 1] = \infty$
10  $i = 1$
11  $j = 1$
12  for $k = p$ to $r$
13      if $L[i] \leq R[j]$
14          $A[k] = L[i]$
15          $i = i + 1$
16      else $A[k] = R[j]$
17          $j = j + 1$
```

Image 2. Pseudo code for merge sort

A Heap Sort algorithm **is not a stable sorting algorithm**. This is because when the heap sort algorithm is being implemented, it takes the largest element (root) and places it into the last index of the array. Placing the largest element in the array destroys the order of occurrence while sorting the algorithm. For example, consider an array containing elements { 4, 3a, 2, 1, 3b}. After using the maxHeap, we get { 4, 3a, 3b, 2, 1 }. We are using 3a and 3b to show the order of elements. When heapsort is called, we first take the root, in our case 4 and put it at the end of our list, then we take 3a in the second last, and 3b in the third last. Our final sorted array looks like: { 1, 2, 3b, 3a, 4 } in the final sort. Therefore, a heap sort algorithm does not maintain the order of occurrence of the array, and hence, is not stable.

A Bubble Sort algorithm **is a stable sorting algorithm**. We can use the pseudo code in **problem 6.1a** to prove this (also written below with line number for convenience). The if statement in line 4 (highlighted) runs only if arr[i] is strictly greater than arr[i+1]. That means, if two terms are equal, they are not swapped with one another and hence, the order of occurrence of the array is maintained. Therefore, bubble sort is a stable sorting algorithm.

```
1. int arr[n] //an array of n elements
2. for (int j = 0; j < n-1; j++) { //a loop that going till n-2 because n-1 will be largest by then
3.      for (int i = 0; i < n-1; i++) { //a loop that iterates from the 1st elem to the 2nd last elem
4.           if (arr[i] > arr[i+1] { //if the previous term is greater than the next
5.                swap (arr[i], arr[i+1]) //perform a swap
6.           }
7.      }
8. }
```

d.
A sorting algorithm is adaptive, if it takes advantage of existing order in its input. Thus, it benefits from the pre-sortedness in the input sequence and sorts faster.

Insertion Sort is an adaptive algorithm as it does not swap the elements if the existing order of the input is already sorted.

Merge Sort is not an adaptive algorithm as it has to go through the whole breaking the array down into two parts and merging them in order to sort the elements in all cases.

Heap Sort itself is not adaptive, BUT an enhanced version of the heapsort algorithm exists that uses the idea of randomized binary search trees to place the input according to any preexisting order. This method is used to place the elements into the heap such that the heap is not responsible in maintaining the order of elements.

Bubble Sort is an adaptive algorithm as it checks whether the array is already sorted or not on every swap (step).

Problem 6.2
The c++ file “heapSort.cpp” is created for this problem.