Gebze Technical University

Department Of Computer Engineering

CSE 222/505 - Spring 2023

Data Structures and Algorithms

Homework #7

Zeynep Çiğdem Parlatan

1901042705

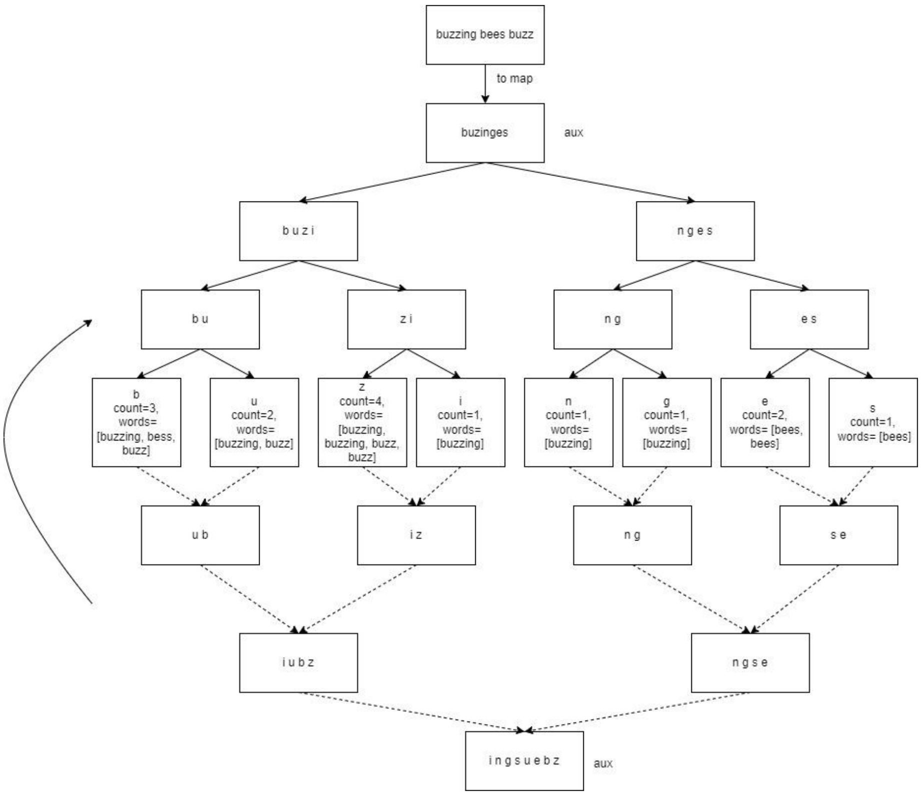
Part 2

a.Best, average, and worst case time complexities analysis of each sorting algorithm.

1.MergeSort Algorithm;

In the merge sort algorithm, we employ the divide and conquer approach. That is, we first divide the given unordered list into smaller parts, perform the sorting operation on each part, and then combine these sorted parts to obtain the entire list.

Let's we demonstrate the working of the merge sort algorithm with an example.


buzzing bees buzz
to map
buzinges
aux
b u z i
n g e s
b u
z i
n g
e s
b count=3, words= [buzzing, bess, buzz]
u count=2, words= [buzzing, buzz]
z count=4, words= [buzzing, buzzing, buzz, buzz]
i count=1, words= [buzzing]
n count=1, words= [buzzing]
g count=1, words= [buzzing]
e count=2, words= [bees, bees]
s count=1, words= [bees]
u b
i z
n g
s e
i u b z
n g s e
i n g s u e b z
aux

Now let's examine the time complexity of merge sort;

- Merge sort is recursive algorithm. mergeSort algorithm => T(n) = 2T(n/2) + n => O(nlogn)

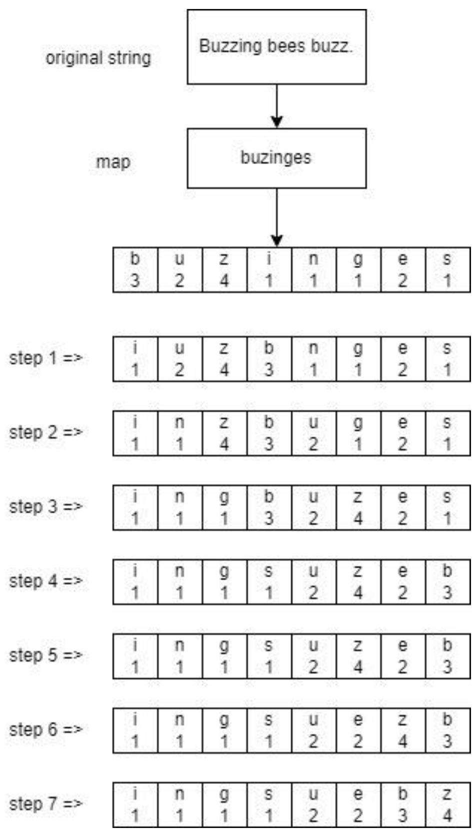
- In summary, in the merge sort algorithm, we recursively divide the array, sort the divided parts, and then combine them back together. This process is same for all scenarios. Therefore, the worst case, best case, and average case time complexity values are the same, which is **O(nlogn)**.

	Best case	Average Case	Worst Case
merge Sort	O(nlogn)	O(nlogn)	O(nlogn)

2.Selection sort;

In the selection sort algorithm, we first find the smallest element in an array (thus traversing the entire array) and swap it with the element at the beginning. In other words, we put the smallest element of the array at the front and we no longer look at the beginning. Then, we find the second smallest element in the remaining array and put it in the second position. We continue this process until the entire array is sorted.

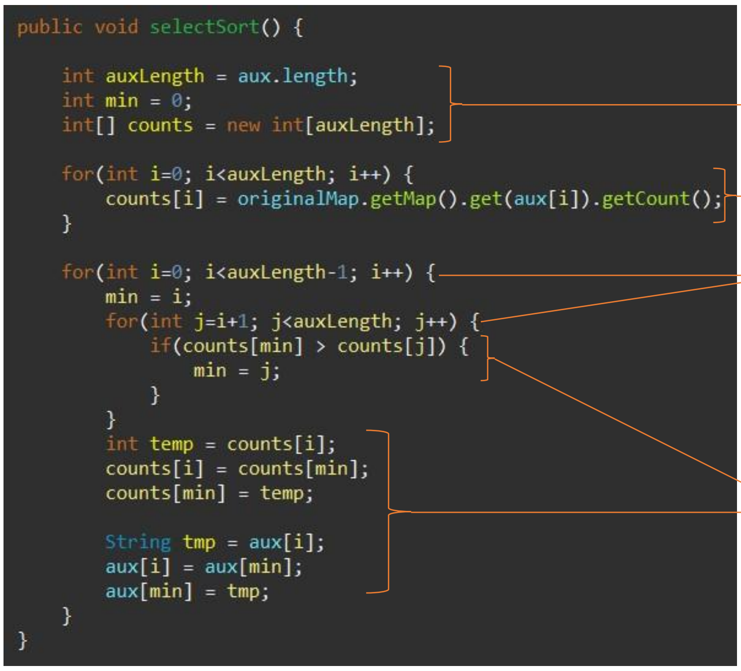
Let's we demonstrate the working of the selection sort algorithm with an example.



	b 3	u 2	z 4	i 1	n 1	g 1	e 2	s 1
step 1 =>	i 1	u 2	z 4	b 3	n 1	g 1	e 2	s 1
step 2 =>	i 1	n 1	z 4	b 3	u 2	g 1	e 2	s 1
step 3 =>	i 1	n 1	g 1	b 3	u 2	z 4	e 2	s 1
step 4 =>	i 1	n 1	g 1	s 1	u 2	z 4	e 2	b 3
step 5 =>	i 1	n 1	g 1	s 1	u 2	z 4	e 2	b 3
step 6 =>	i 1	n 1	g 1	s 1	u 2	e 2	z 4	b 3
step 7 =>	i 1	n 1	g 1	s 1	u 2	e 2	b 3	z 4

First, I assigned the key values from my created map to my aux array. Then, to implement the selection sort algorithm, I created an int array and assigned the count values of the keys from the aux array to this array. Then, I iterated through the loop to find the smallest element and performed a swap operation. After this process, I obtained the sorted auxiliary array.

Now let's examine the time complexity of selection sort;



```java
public void selectSort() {

    int auxLength = aux.length;
    int min = 0;
    int[] counts = new int[auxLength];

    for(int i=0; i<auxLength; i++) {
        counts[i] = originalMap.getMap().get(aux[i]).getCount();
    }

    for(int i=0; i<auxLength-1; i++) {
        min = i;
        for(int j=i+1; j<auxLength; j++) {
            if(counts[min] > counts[j]) {
                min = j;
            }
        }
        int temp = counts[i];
        counts[i] = counts[min];
        counts[min] = temp;

        String tmp = aux[i];
        aux[i] = aux[min];
        aux[min] = tmp;
    }
}
```

O(1)

O(n) Because it loops as much as the number of elements in the array.

We have two nested loops. Therefore, the time complexity becomes O(n^2).

Time complexity of swap operations is O(1)

So,

- **Worst case time complexity=>** The worst-case scenario for selection sort occurs when the elements in the array are given in reverse order (i.e., from largest to smallest). The time complexity becomes **O(n^2)**.

- **Best case time complexity=>** We can consider the best case scenario as when the array is already sorted. However, in the selection sort algorithm, even if the array is sorted, it still needs to traverse the entire array until it confirms that the array is sorted (i.e., it still needs to examine all 'n' elements). Therefore, the time complexity is **O(n^2).**

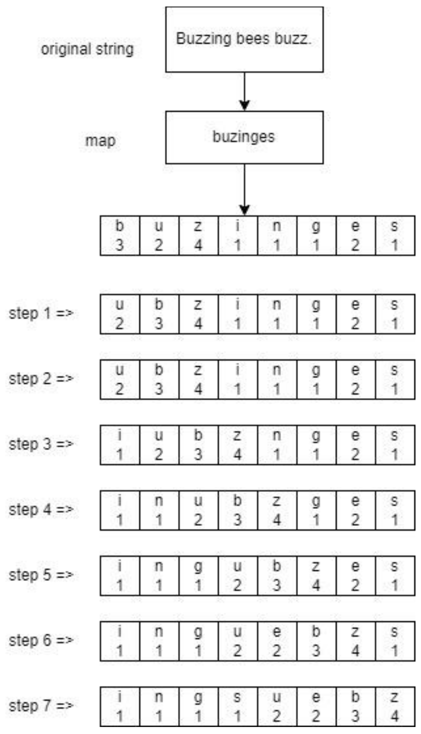
- **Average case time complexity =>** According to best case and worst case time complexities, the average time complexity is **O(n^2)** too.

	Best case	Average Case	Worst Case
Selection Sort	O(n^2)	O(n^2)	O(n^2)

3.Insertion sort;

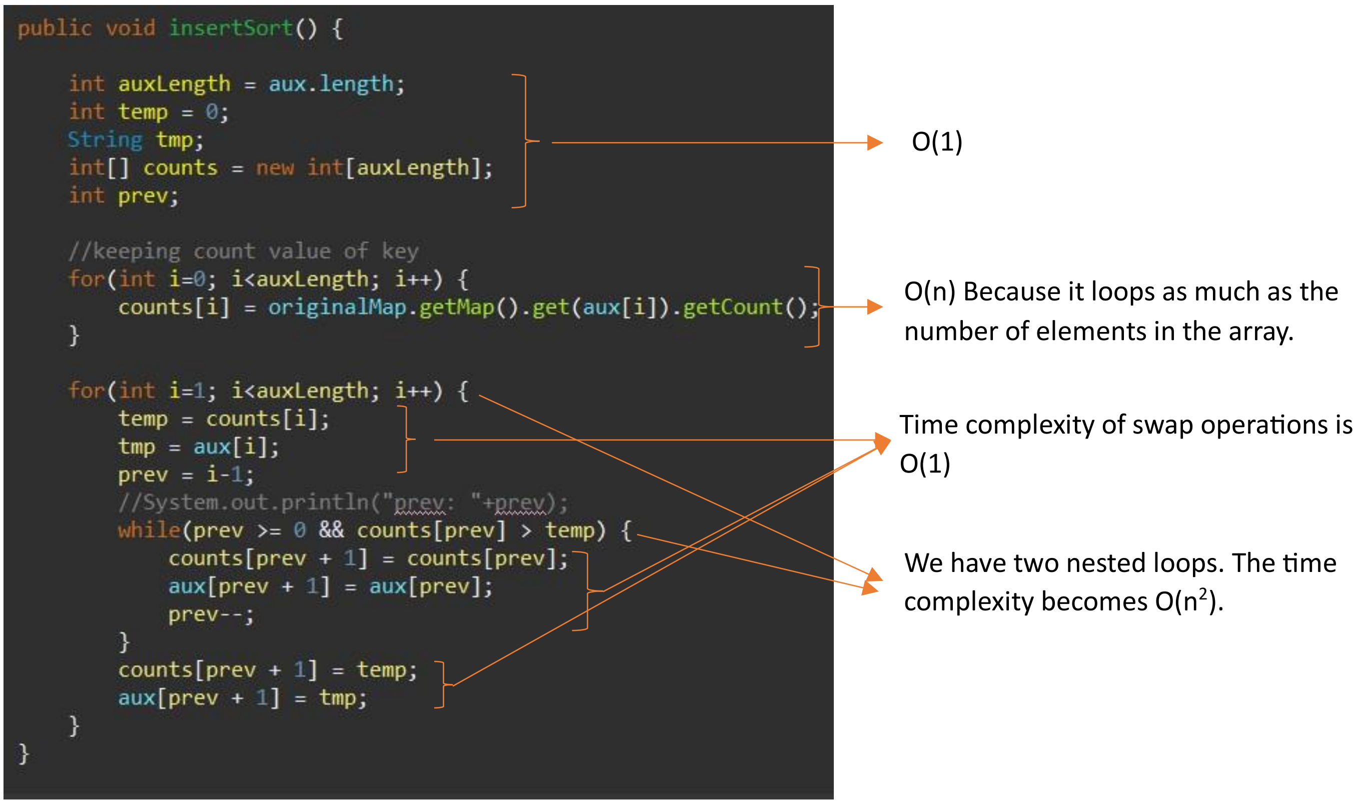
It is a linear sorting algorithm. In this algorithm, in the loop, we compare the element at the current index with the previously sorted elements front it and perform a swap operation if necessary. In other words, we shift it forward (swap with the previous element) until it is larger than the previous element. To explain it better, we can provide the following example.

Let's we demonstrate the working of the insertion sort algorithm to explain it better with an example.


original string
Buzzing bees buzz.
map
buzinges
b 3 u 2 z 4 i 1 n 1 g 1 e 2 s 1
step 1 => u 2 b 3 z 4 i 1 n 1 g 1 e 2 s 1
step 2 => u 2 b 3 z 4 i 1 n 1 g 1 e 2 s 1
step 3 => i 1 u 2 b 3 z 4 n 1 g 1 e 2 s 1
step 4 => i 1 n 1 u 2 b 3 z 4 g 1 e 2 s 1
step 5 => i 1 n 1 g 1 u 2 b 3 z 4 e 2 s 1
step 6 => i 1 n 1 g 1 u 2 e 2 b 3 z 4 s 1
step 7 => i 1 n 1 g 1 s 1 u 2 e 2 b 3 z 4

First, I assigned the key values from my created map to my aux array. Then, to implement the insert sort algorithm, I created an int array and assigned the count values of the keys from the aux array to this array. Then, throughout the loop, I compared the element in the first index of the count array with the elements before it, and swapped if necessary. According to the replacements in the Count array, I also made the swap operation in the aux array. And by repeating these operations, I got the sorted array.

Now let's examine the time complexity of insert sort;



```java
public void insertSort() {

    int auxLength = aux.length;
    int temp = 0;
    String tmp;
    int[] counts = new int[auxLength];
    int prev;

    //keeping count value of key
    for(int i=0; i<auxLength; i++) {
        counts[i] = originalMap.getMap().get(aux[i]).getCount();
    }

    for(int i=1; i<auxLength; i++) {
        temp = counts[i];
        tmp = aux[i];
        prev = i-1;
        //System.out.println("prev: "+prev);
        while(prev >= 0 && counts[prev] > temp) {
            counts[prev + 1] = counts[prev];
            aux[prev + 1] = aux[prev];
            prev--;
        }
        counts[prev + 1] = temp;
        aux[prev + 1] = tmp;
    }
}
```

O(1)

O(n) Because it loops as much as the number of elements in the array.

Time complexity of swap operations is O(1)

We have two nested loops. The time complexity becomes O(n^2).

So,

- **Worst case time complexity =>** The worst-case scenario for insertion sort occurs when the elements in the array are given in reverse order (i.e., from largest to smallest). The time complexity becomes **O(n^2).** The outer loop iterates 'n-1' times, and the inner loop iterates based on the number of elements until the condition is met. In the worst case, the smallest element will be at the end, resulting in a time complexity of O(n^2).

- **Best case time complexity =>** The best-case scenario for insertion sort is when the elements in the array are already sorted in ascending order. In this case, it does not enter the inner while loop because the elements are already in the correct positions. The outer for loop will iteratenumber of elements-1 times. Therefore, the time complexity becomes O(n).

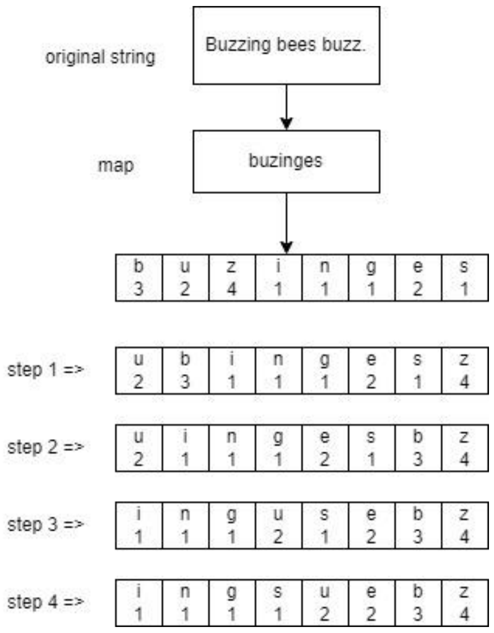
- **Average case time complexity =>** In the Average case, the time complexity of insertion sort O(n^2), depending on the best and worst case time complexities. For example, an array can be partially ordered or partially unordered.

	Best case	Average Case	Worst Case
Insertion sort	O(n)	O(n^2)	O(n^2)

4.Bubblesort;

In the bubble sort algorithm, we start from the beginning and compare consecutive pairs of elements until we reach the end of the array. When we compare two consecutive elements, if the element at the front index is smaller, we don't do anything because it is already sorted. However, if the element at the front index is larger, we swap these two elements (perform a swap operation). By repeating this process, we eventually achieve a sorted array.

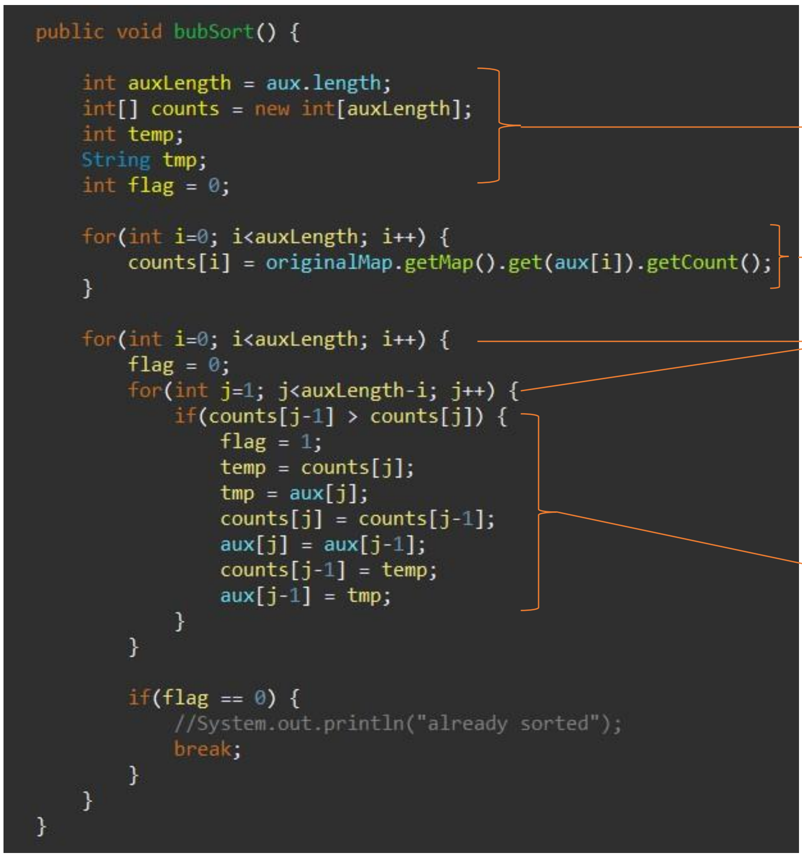
Let's we demonstrate the working of the bubble sort algorithm with an example.


original string
Buzzing bees buzz.
map
buzinges
b 3 | u 2 | z 4 | i 1 | n 1 | g 1 | e 2 | s 1
step 1 =>
u 2 | b 3 | i 1 | n 1 | g 1 | e 2 | s 1 | z 4
step 2 =>
u 2 | i 1 | n 1 | g 1 | e 2 | s 1 | b 3 | z 4
step 3 =>
i 1 | n 1 | g 1 | u 2 | s 1 | e 2 | b 3 | z 4
step 4 =>
i 1 | n 1 | g 1 | s 1 | u 2 | e 2 | b 3 | z 4

First, I assigned the key values from my created map to my aux array. Then, to implement the bubble sort algorithm, I created an int array and assigned the count values of the keys from the aux array to this array. I compared the values of consecutive elements in the count array and performed a swap operation if necessary. I also applied this swap operation in the aux array. By repeating these steps, I achieved a sorted array.

In bubble sort, since the largest element always moves to the end after each iteration, we do not need to recheck it.

Now let's examine the time complexity of bubble sort;



```java
public void bubSort() {

    int auxLength = aux.length;
    int[] counts = new int[auxLength];
    int temp;
    String tmp;
    int flag = 0;

    for(int i=0; i<auxLength; i++) {
        counts[i] = originalMap.getMap().get(aux[i]).getCount();
    }

    for(int i=0; i<auxLength; i++) {
        flag = 0;
        for(int j=1; j<auxLength-i; j++) {
            if(counts[j-1] > counts[j]) {
                flag = 1;
                temp = counts[j];
                tmp = aux[j];
                counts[j] = counts[j-1];
                aux[j] = aux[j-1];
                counts[j-1] = temp;
                aux[j-1] = tmp;
            }
        }

        if(flag == 0) {
            //System.out.println("already sorted");
            break;
        }
    }
}
```

O(1)

O(n) Because it loops as much as the number of elements in the array.

We have two nested loops. The outer loop iterates 'n' times for 'n' elements. The inner loop iterates 'n-i' times. Therefore, the time complexity becomes $O(n^2)$.

Time complexity of swap operations is O(1)

So,

- **Worst case time complexity =>** The worst-case scenario for bubble sort occurs when the elements in the array are given in reverse order (i.e., from largest to smallest). The time complexity becomes **O(n^2)**. As I mentioned in the analysis above, the outer loop iterates 'n' times for 'n' elements, and the inner loop iterates 'n-i' times.

- **Best case time complexity =>** It is possible to further optimize the bubble sort algorithm. For this, I have introduced a flag. If no swap operation is performed, it means that the array is already sorted, and we can exit the loop.
  Therefore, the best-case scenario for bubble sort is when the elements in the array are already sorted in ascending order. In this case, since the array is already sorted, the flag value will be unchanged, and we can exit the loop. As a result, the time complexity becomes **O(n)**. What I mean is that if we provide a sorted array, and the inner loop does not perform any swap operation, we will exit the loop after iterating 'n' times. Hence, the best-case time complexity is O(n).

- **Average case time complexity =>** In bubble sort, the average case time complexity is still **O(n^2)**. ((n+n^2)/2)). For example, the array can be partially sorted or partially unsorted.

	Best case	Average Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)

5.Quick Sort;

In this algorithm too, we use the divide and conquer approach . We are dividing the array according to a certain value (is called pivot). The left subarray is less than or equal to the pivot.Right subarray are greater than pivot. By repeating this process, we arrive at our sorted array.

Time complexities of this algortihm;

	Best case	Average Case	Worst Case
Quick sort	O(nlogn)	O(nlogn)	O(n^2)

Summary;

	Worst Case - Notation	**Average Case - Notation**	**Best Case- Notation**
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Insertion Sort	O(n^2)	O(n^2)	O(n)
Bubble Sort	O(n^2)	O(n^2)	O(n)
Quick sort	O(n^2)	O(nlogn)	O(nlogn)

b. Running time of each sorting algorithm for each input.

- Worst case input = "hhhhhhgg ffgggff eeedd ccBA"
- Average case input = "ABeee ccffdd gggffgg hhhhhh"
- Best Case input = "ABcc ddeee ffgggff gghhhhhh"

	Worst Case	**Average Case**	**Best Case**
Merge Sort	28600	27300	22500
Selection Sort	18900	17200	18500
Insertion Sort	15900	13800	10700
Bubble Sort	15200	11800	9700
Quick sort	16100	15800	15700

c. Comparison of the sorting algorithms (by using the information from Part2-a and Part2-b) Which algorithm is faster in which case?

	Worst Case - Notation	**Average Case - Notation**	**Best Case- Notation**
Merge Sort	28600 - O(nlogn)	27300 - O(nlogn)	21500 - O(nlogn)
Selection Sort	18900 - $O(n^2)$	17200 - $O(n^2)$	18500 - $O(n^2)$
Insertion Sort	15900 - $O(n^2)$	13800 - $O(n^2)$	10700 - O(n)
Bubble Sort	15200 - $O(n^2)$	11800 - $O(n^2)$	9700 - O(n)
Quick sort	16100 - $O(n^2)$	15800 - O(nlogn)	15700 - O(nlogn)

- According to time complexity analysis (2.a)
  - ❖ **For worst case;**
    (Selection Sort = Insertion Sort = Bubble Sort = Quick Sort) > Merge Sort
  - ❖ **For average case;**
    (Selection Sort = Insertion Sort = Bubble Sort) > (Merge Sort = Quick Sort)
  - ❖ **For best case;**
    Selection Sort > (Merge Sort = Quick Sort) > (Insertion Sort = Bubble Sort)

- According to running time (2.b)
  - **For worst case;**
    MergeSort > Selection Sort > Quick Sort> Insertion Sort > Bubble Sort
  - **For average case;**
    MergeSort> Selection Sort> Quick Sort> Insertion Sort> BubbleSort
  - **For best case;**
    Merge Sort> Selection Sort> Quick Sort> Insertion Sort> Bubble Sort

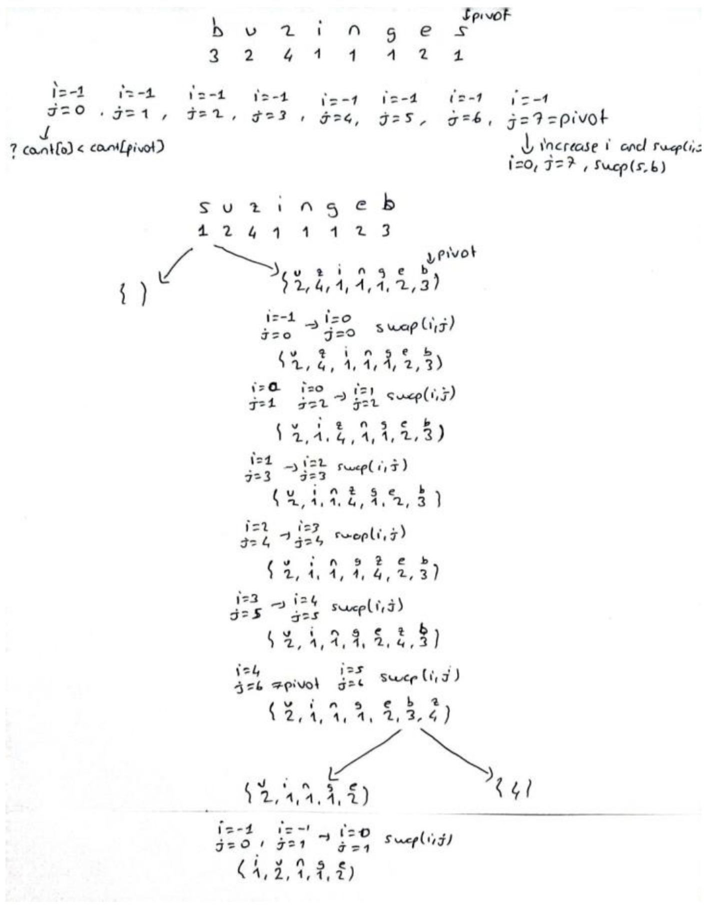
- According to both time complexity analysis and running time results, bubble sort was the fastest algorithm in the best case.

** I have had very different results for running times. Usually, I ran the algorithm several times consecutively and write the result that I get the most.

** There are minor discrepancies in time complexity and running times. Especially in cases where merge sort should be small, it came out more. This seems to be due to the calculation of the running time. I tried it on both windows and linux,and the results generally fall within a similar range.

d.You are expected to analyze which algorithms keep the input ordering and which don't?

Above, I showed how all algorithms work by drawing a test. As we can understand from there, we get the same output in mergeSort, selection sort, insertion sort and bubble sort. That is, they preserve the input order of the letter in the input. But we get different results in quick sort. In the picture below, I tried to explain the reason and the working logic of the algorithm.



As you can see, the order of the letters in the word is not preserved here. In the algorithm, we choose the last element as pivot. By using two variables, i and j, we perform comparison and swap operations with elements in the loop. Here, if the element is greater than or equal to the pivot, we ignore it. If it is less than the pivot, we increase i and we swap(i,j). So equal values with counts do not preserve the input order. For example, in this example, the letter 's' is prefixed according to the algorithm. But in other algorithms, the input order was preserved and the letter 'i' was prefixed. For example, in selection sort, it finds the smallest element in the array and places it at the beginning. When examining the elements from the beginning, the first element with the smallest count, which is 'i,' will be found, and it is moved to the front.