GEBZE TECHNICAL UNIVERSITY

CSE 222
DATA STRUCTURES AND ALGORITHMS

HOMEWORK 7
REPORT

OGUZ MUTLU
1801042624

1. DETAILED SYSTEM REQUIREMENTS

- Take an input string and preprocess that string, if there is capital letter, convert it to lower case letter, If there is a character which is not a letter or space, delete it. Spaces should be kept in order
to keep track of the words.

- By using the preprocessed string we have to build custom map structure, in this structure we will have to keep count of occurences of each letter withing the string along with the words that it was seen

- After custom my map build operation is done we have to sort this array by using count values of myMap class we have to do selection, insertion, bubble, and quick sort on that array

2. PROBLEM SOLUTION APPROACH

-Selection Sort

This method performs selection sort on the originalMap based on the count of the elements, the sorting is done in ascending order. This method iterates through the elements of the original map using the aux list. For each iteration it selects the elements with the minimum count and swaps it with the current current element.The resulting sorted elemets are stored in the aux list by key values, originalMap does not change.

```
public void SelectionSort(){
    int first, second, tempIndex;

    for(int i = 0; i < originalMap.getMapSize()-1; i++){
        first = originalMap.getMap().get(aux.get(i)).getCount();
        tempIndex = i;

        for(int j = i+1; j < originalMap.getMapSize(); j++){
            second = originalMap.getMap().get(aux.get(j)).getCount();
            if(first > second){
                tempIndex = j;
                first = second;
            }
        }
        String temp = aux.get(i);
        aux.set(i, aux.get(tempIndex));
        aux.set(tempIndex, temp);
    }

    buildSortedMap();
```

-Insertion Sort

This method performs selection sort on the originalMap based on the count of the elements, the sorting is done in ascending order. This method iterates through the elements of the original map using the aux list. For each iteration If the count of the selected element is smaller, it shifts the previous elements to make space for the selected element.The resulting sorted elemets are stored in the aux list by key values, originalMap does not change.

```
@throws IndexOutOfBoundsException if the index is out of range (i.e., negative or
*/
public void InsertionSort(){
    int first, tempIndex;
    int j = 0;
    for(int i = 1; i < originalMap.getMapSize(); i++){
        first = originalMap.getMap().get(aux.get(i)).getCount();
        tempIndex = i;
        j = i-1;
        while(j >= 0 && originalMap.getMap().get(aux.get(j)).getCount() > first)
        {
            String temp = aux.get(tempIndex);
            aux.set(tempIndex, aux.get(j));
            aux.set(j, temp);
            tempIndex = j;
            j--;
        }
    }
    buildSortedMap();
}
```

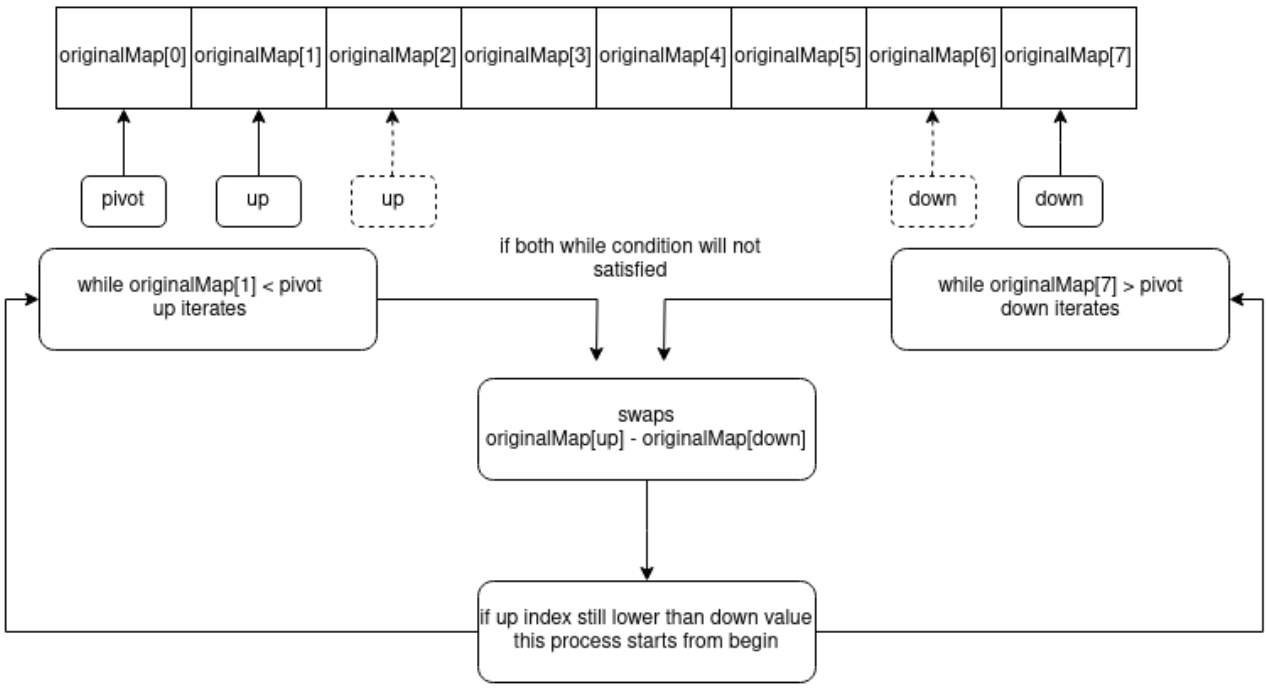
-Quick Sort

This helper method used in the quick sort algorithm this process done by recursively, sort the elements within the specific range, it divides aux list or the originalMap into sub-lists, doing that process by using pivot, up and down.

QuickSort is a highly efficient sorting algorithm that follows the divide-and-conquer approach. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

In the figure below I will show how my code works.

Figure 1.




```
private void quickSortHelper(int left, int rigth)
{
    if(left < rigth)
    {
        int pivotIndex = partition(left, rigth);
        quickSortHelper(left, pivotIndex-1);
        quickSortHelper(pivotIndex+1, rigth);
    }
}
```

```
private int partition(int left, int rigth)
{

    String pivot = aux.get(left);
    int up = left;
    int down = rigth;

    do{
        while(originalMap.getMap().get(pivot).getCount() >= originalMap.getMap().get(aux.get(up)).getCount()
        {
            up++;
        }
        while(originalMap.getMap().get(pivot).getCount() < originalMap.getMap().get(aux.get(down)).getCount(
        {
            down--;
        }
        if(up < down)
        {
            swap(up, down);
        }
    }while(up<down);
    swap(left, down);
    return down;
}
```

-Bubble Sort

this method performs bubble sort on the originalMap based on the count of elements sorting is done by ascending order It compares adjacent elements and swaps them if they are in the wrong order.

```
public void BubbleSort() {
    for (int i = 0; i < originalMap.getMapSize()-1; i++)
    {
        for(int j = 0; j < originalMap.getMapSize()-i- 1; j++ )
        {
            int first = originalMap.getMap().get(aux.get(j)).getCount();
            int second = originalMap.getMap().get(aux.get(j+1)).getCount();

            if(first > second)
            {
                String temp = aux.get(j);
                aux.set(j, aux.get(j+1));
                aux.set(j+1, temp);
            }
        }
    }
    buildSortedMap();
}
```

Part2.a
TIME COMPLEXITY ANALYSIS (Theoretical)

1. Merge Sort
Merge sort shows the two-element arrays formed by merging two-element pairs and so on each merge operation will iterate n size of array so merge operation will be done in O(n).The number of operations that require merging is log n because each recursive step splits the array in half. So the total effort to reconstruct the sorted array through merging is O(n log n).
Best: O(n*logn)
Worst: O(n*logn)
Average:O(n*logn)

2.Selection Sort
Since selection sort iterates all over the comparable object whether it is sorted or not it iterates all over the objects, so that means from first index to the (n-1) for first loop, i+1 to n for second loop that means involves a comparison of items and is performed (n – 1 – i) times for each value of i. Since i takes on all values between 0 and n − 2, the following series computes the number of executions of the selection sort
(n – 1) +(n – 2)+ . . . 3+ 2+ 1 and that means n(n-1)/2 = (n^2)/2 - (n)/2
Best: O(n^2)
Worst: O(n^2)
Average:O(n^2)

3.Insertion Sort
Best: Since insertion sort checks previous indexes if our comparable object is already sorted our if condition executed and swap operations will not done so that means inner while loop never executes and best case will be O(n)
Worst: If our comparable object will not be sorted order then from index 1 to size of comparable object(n-1) will be iterated then we compare temp value from i-1 to 0 so that means (n-1) => (n-1)(n-1) = n^2 -2n +1 so that means worst case will be O(n^2)
Average:O(n^2)

4.Bubble Sort

If the comparable object is not given sorted order that means all bubble sort iterates all over the object for comparison and then iterates all remaining object for comparison also that means two inner for loop iterates mathematical explanation means in the code part I will not optimize bubble sort so that means for loop iterates over (i) to (n-1) and j to (n – (i)-1) => (n-1)+ (n-2) + (n-3) + … + 2 + 1
so that means (n-1)*(n-1+1) / 2 => (n-1)(n)/2 => O(n^2)

Best:O(n^2)
Worst: O(n^2)
Average:O(n^2)

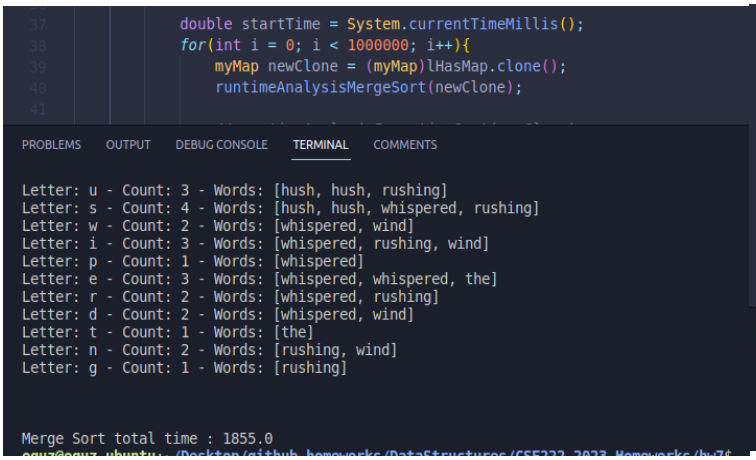
5.Quick Sort

If the pivot value is a random value selected from the current subarray, then statistically it is expected that half of the items in the subarray will be less than the pivot and half will be greater than the pivot. If both subarrays always have the same number of elements (the best case), there will be log n levels of recursion. At each level, the partitioning process involves moving every element into its correct partition, so quicksort is O(n log n) There is one situation, however, where quicksort gives very poor behavior. If, each time we partition the array, we end up with a subarray that is empty(already sorted comparable objects), the other subarray will have one less element than the one just split (only the pivot value will be removed). Therefore, we will have n levels of recursive calls (instead of log n), and the algorithm will be O(n2).

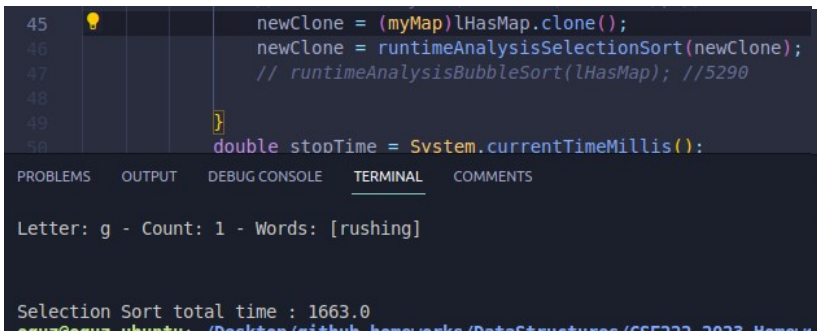
Best: O(n*logn)
Worst: O(n^2)
Average:O(n*logn)

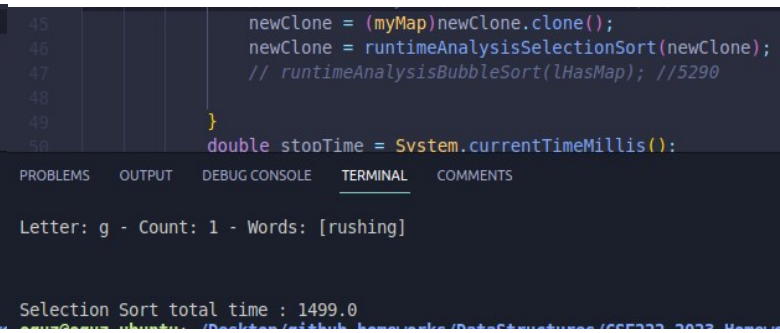
Part2.b

-Merge Sort (1855 ms unsorted – 1905ms for sorted)

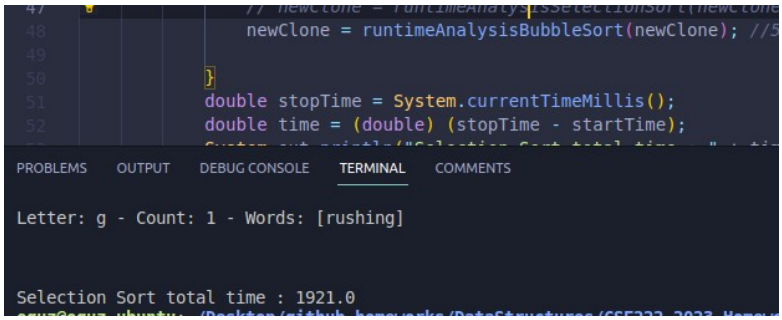


-Selection Sort (1633 ms unsorted – 1500ms for sorted)

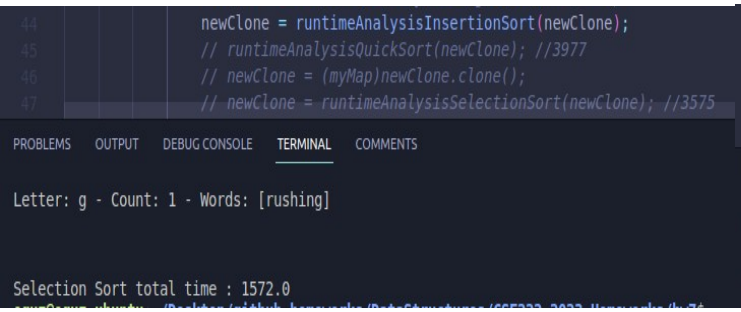


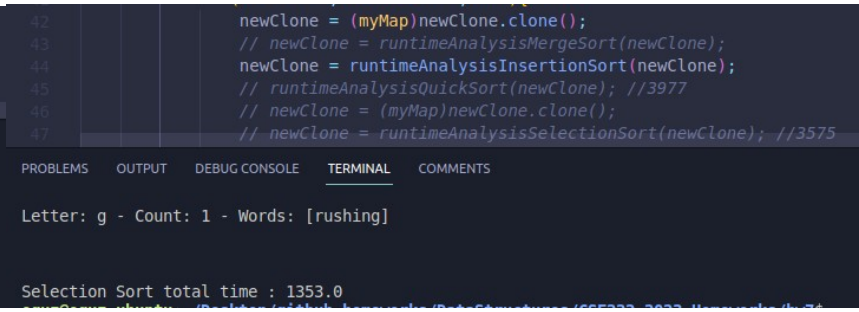


-Bubble Sort (1921 ms unsorted)

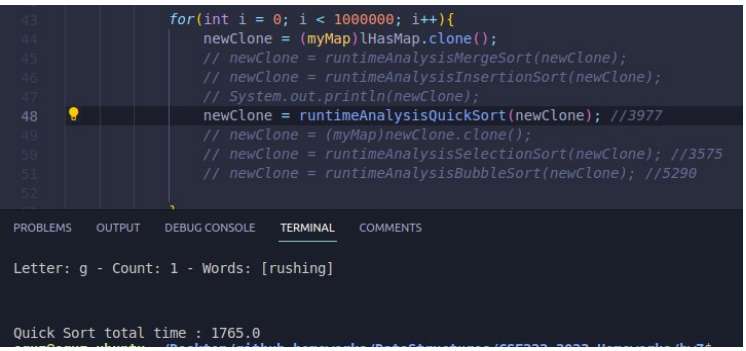


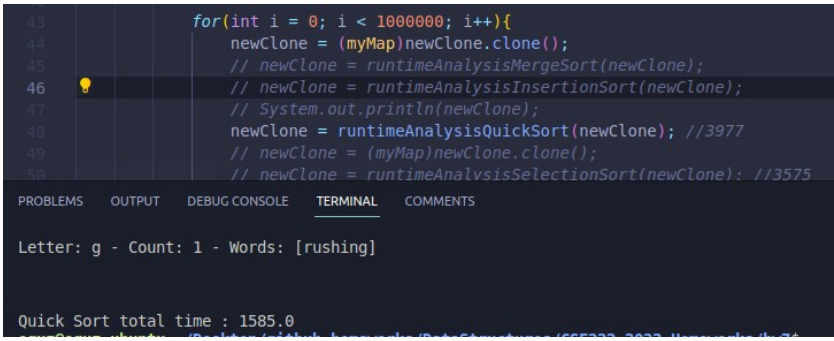
-Insertion Sort (1575 ms unsorted – 1353ms for sorted)





-Quick Sort (1765 ms unsorted – 1585ms for sorted)





Part2.c

TIME COMPLEXITY ANALYSIS (Experimental - Theoretical)

1. Merge Sort:
   - Theoretical time complexity: O(n log n)
   - Experimental time: 1855
2. Selection Sort:
   - Theoretical time complexity: O(n^2)
   - Experimental time: 1633
3. Bubble Sort:
   - Theoretical time complexity: O(n^2)
   - Experimental time: 1921
4. Insertion Sort:
   - Theoretical time complexity: O(n^2)
   - Experimental time: 1585
5. Quick Sort:
   - Theoretical time complexity: O(n log n) on average
   - Experimental time: 1765

These times might vary depending on the hardware, programming language, compiler optimizations, and other factors. When comparing the experimental times with the theoretical time complexities, we can observe that the experimental times do not fit the theoretical expectations. Also in insertion sort when we sort our map then we give that sorted array as an input we will see that execution time decrease, but in Quick sort, if we give sorted array as an input time will decrease also, this calculations depends on specific features e.gcompiler optimization, cache memory, our dataset size…

Part2.d

There is a difference in quick sort compared to other sort outputs. The reason for this will be examined through the diagram in the title where quick sort is explained.
As seen in the figure, first the first element of the array is selected as the pivot and comparisons are started. If the up<down condition is met, the do{}while structure is exited.and then a swap between pivot and down is performed. For this reason, the entry order is revealed differently from other sorts.For example, let our first employee be our smallest employee, but let our employee come first. When the down value is lower than the up value, let our first element be replaced with the 4th element. Since this process continues by fragmenting, the swap operation cannot be guaranteed to be the first entry element.

3.TEST CASES AND RESULTS
-Selection Sort Result

```
SELECTION SORT
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```

```
SELECTION SORT
Letter: i - Count: 1 - Words: [buzzing]
Letter: n - Count: 1 - Words: [buzzing]
Letter: g - Count: 1 - Words: [buzzing]
Letter: s - Count: 1 - Words: [bees]
Letter: u - Count: 2 - Words: [buzzing, buzz]
Letter: e - Count: 2 - Words: [bees, bees]
Letter: b - Count: 3 - Words: [buzzing, bees, buzz]
Letter: z - Count: 4 - Words: [buzzing, buzzing, buzz, buzz]
```

-Insertion Sort Result

```
INSERTION SORT
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```

```
INSERTION SORT
Letter: i - Count: 1 - Words: [buzzing]
Letter: n - Count: 1 - Words: [buzzing]
Letter: g - Count: 1 - Words: [buzzing]
Letter: s - Count: 1 - Words: [bees]
Letter: u - Count: 2 - Words: [buzzing, buzz]
Letter: e - Count: 2 - Words: [bees, bees]
Letter: b - Count: 3 - Words: [buzzing, bees, buzz]
Letter: z - Count: 4 - Words: [buzzing, buzzing, buzz, buzz]
```

-Bubble Sort Result

```
BUBBLE SORT
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```

```
BUBBLE SORT
Letter: i - Count: 1 - Words: [buzzing]
Letter: n - Count: 1 - Words: [buzzing]
Letter: g - Count: 1 - Words: [buzzing]
Letter: s - Count: 1 - Words: [bees]
Letter: u - Count: 2 - Words: [buzzing, buzz]
Letter: e - Count: 2 - Words: [bees, bees]
Letter: b - Count: 3 - Words: [buzzing, bees, buzz]
Letter: z - Count: 4 - Words: [buzzing, buzzing, buzz, buzz]
```

-Quick Sort Result

```
QUICK SORT
Letter: t - Count: 1 - Words: [the]
Letter: p - Count: 1 - Words: [whispered]
Letter: g - Count: 1 - Words: [rushing]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```

```
QUICK SORT
Letter: n - Count: 1 - Words: [buzzing]
Letter: s - Count: 1 - Words: [bees]
Letter: i - Count: 1 - Words: [buzzing]
Letter: g - Count: 1 - Words: [buzzing]
Letter: u - Count: 2 - Words: [buzzing, buzz]
Letter: e - Count: 2 - Words: [bees, bees]
Letter: b - Count: 3 - Words: [buzzing, bees, buzz]
Letter: z - Count: 4 - Words: [buzzing, buzzing, buzz, buzz]
```

4.NOTE

```
// System.out.println(iSort.getSortedMap());

returnedMap = runtimeAnalysisInsertionSort(reverseSorted);//insertion sort worst case
returnedMap = runtimeAnalysisInsertionSort(lHasMap); //insertion sort average case
returnedMap = runtimeAnalysisInsertionSort(normalsorted); //insertion sort best case
```

I take as an reverse order, normal order and default map as a average best and worst case scenario.