GTU Department of Computer Engineering
CSE 222/505 - Spring 2023
Homework 7 Report

Mehmet Mete Şamlıoğlu
200104004093

1-) SYSTEM REQUIREMENTS

The system requires an algorithm that first builds a LinkedHashMap structure after processing the input string. The map structure will be built according to the preprocess input string. After building the map properly, our main task is to sort the map entries by using the count(number of words) value, in cases where the frequency of count values are the same, the second condition that is going to be evaluated is the input order of the key. To perform this operation, it is required to use three classes, one for building the tree, the second one to perform the particular sorting operation. And the very last one is to store the words and the count value for each entry on the map. In the sorting class that contains the code for the wanted sorting, I used ArrayList to store the entries of the map. As was mentioned in the PDF I used my own sort algorithm to perform the sorting operation, I wrote my own sort algorithm to perform the sort on the map structures entries. In the end, I add a method to return a sorted map in the related sorting class. In the end, I obtained both sorted and unsorted map objects where the sorting object has been instantiated.

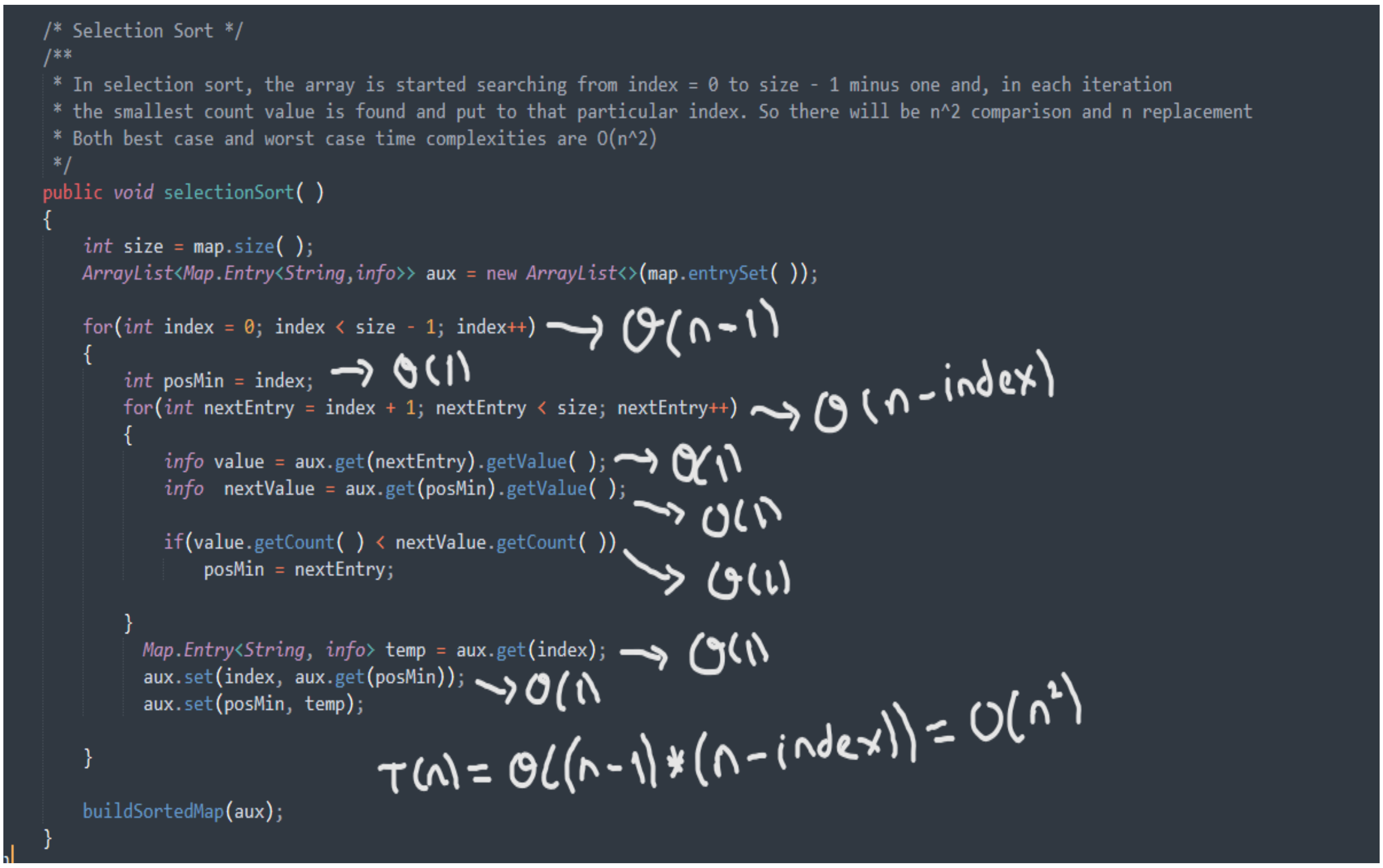
2-) PROBLEM-SOLUTION APPROACH

In this part, I will explain each sorting algorithm, also calculate the time complexity and running times for each one of them. In the end, I will compare their best–case, worst-case and average-case scenarios.

Selection Sort

It sorts the map entries by making several passes through the array, selecting the next smallest item in the array each time, and placing it where it belongs in the array. In each iteration, it finds the smallest item from the unsorted part of the array and replaces it with that particular index. It continues iterating until it reaches the index (size – 2).

Time Complexity Analysis for Selection Sort



```
/* Selection Sort */
/**
 * In selection sort, the array is started searching from index = 0 to size - 1 minus one and, in each iteration
 * the smallest count value is found and put to that particular index. So there will be n^2 comparison and n replacement
 * Both best case and worst case time complexities are O(n^2)
 */
public void selectionSort( )
{
    int size = map.size( );
    ArrayList<Map.Entry<String,info>> aux = new ArrayList<>(map.entrySet( ));

    for(int index = 0; index < size - 1; index++) → O(n-1)
    {
        int posMin = index; → O(1)
        for(int nextEntry = index + 1; nextEntry < size; nextEntry++) → O(n-index)
        {
            info value = aux.get(nextEntry).getValue( ); → O(1)
            info  nextValue = aux.get(posMin).getValue( ); → O(1)

            if(value.getCount( ) < nextValue.getCount( )) → O(1)
                posMin = nextEntry;

        }
         Map.Entry<String, info> temp = aux.get(index); → O(1)
         aux.set(index, aux.get(posMin)); → O(1)
         aux.set(posMin, temp);

    }                    T(n) = O((n-1)*(n-index)) = O(n^2)

    buildSortedMap(aux);
}
```

As you can see above when the index is 0 the inner for loop makes n – 1 execution, and when the index is 1 it makes n – 2 execution and it goes like this. In the end, when the index is equal to (size -2), it makes only one execution and stops iterating. If we add all the executions that have been made in each iteration we will get the following equation.

ber of executions of step 2.3.

$$(n-1)+(n-2)+\cdots+3+2+1$$

This is a well-known series that can be written in closed form as

$$\frac{n\times(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}$$

So that we can ignore all other terms except the most significant term in this expression, **the number of comparisons is O(n^2), and the number of the exchange is O(n). Since the number of comparisons increases with the square of n, the time complexity of the selection sort is O(n^2).**

Best-case, Worst-case, and Average-case Time Complexities for SelectionSort

Best-case scenario: Tb(n) = O(n^2)

- When the given array is already sorted the best-case scenario occurs for selection sort. Even in this scenario, the selection sort needs to make comparisons and exchanges for each element in the array. Therefore, the best Case time complexity is **O(n^2).**

Worst-case scenario: Tw(n) = O(n^2)

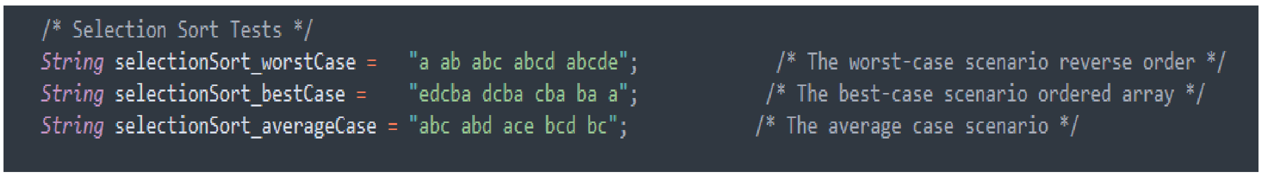
- The worst case occurs when the input array is in reverse order or contains entries that are unsorted completely. In this case, the selection sort needs to make comparisons and exchanges for each array entry and it will result in **O(n^2) time complexity.**

Average-case scenario: T(n) = O(n^2)

- It is also O(n^2) because on average selection sort needs to compare and swap for more than half of the total entries in the input array and this will result in **O(n^2).**

Running Results for SelectionSort for Best-case, Worst-case, and Average-case inputs

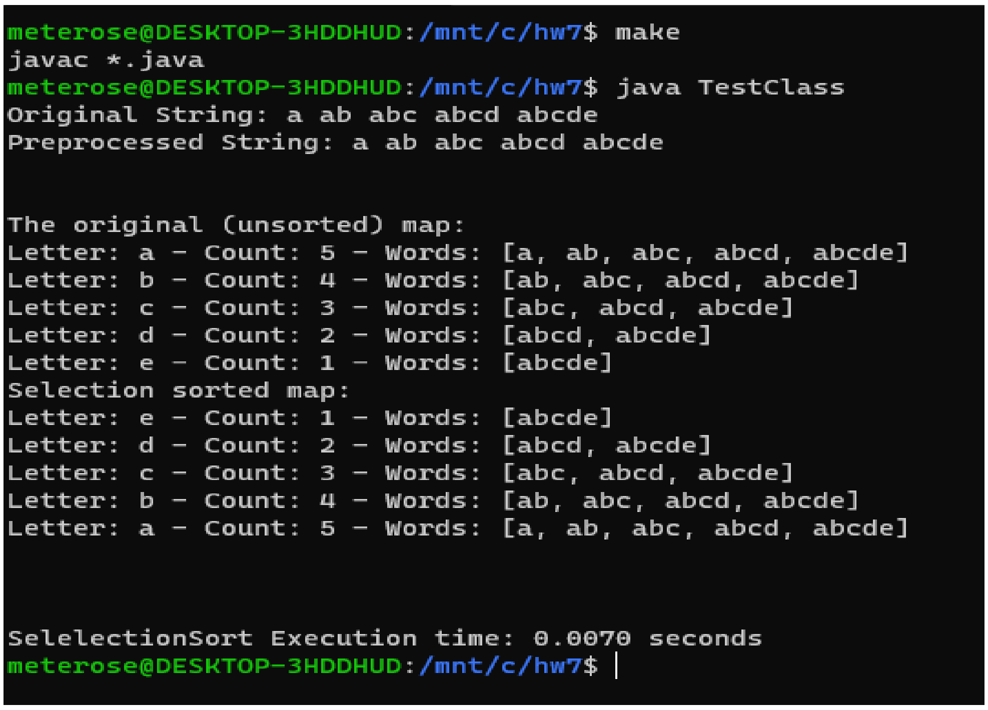
In this part, I will determine three inputs to trigger best-case, worst-case, and average-cascenariosse scenarios for the selection algorithm that I showed above. So I determined the following string to trigger worst-case, best-case and average-case scenarios.



```
/* Selection Sort Tests */
String selectionSort_worstCase =   "a ab abc abcd abcde";          /* The worst-case scenario reverse order */
String selectionSort_bestCase =    "edcba dcba cba ba a";         /* The best-case scenario ordered array */
String selectionSort_averageCase = "abc abd ace bcd bc";         /* The average case scenario */
```

I will pass them to the myMap object one by one and build a myMap by parsing them and after that, by instantiating a SelectionSort object with that myMap **I will measure the running times for each input case.**

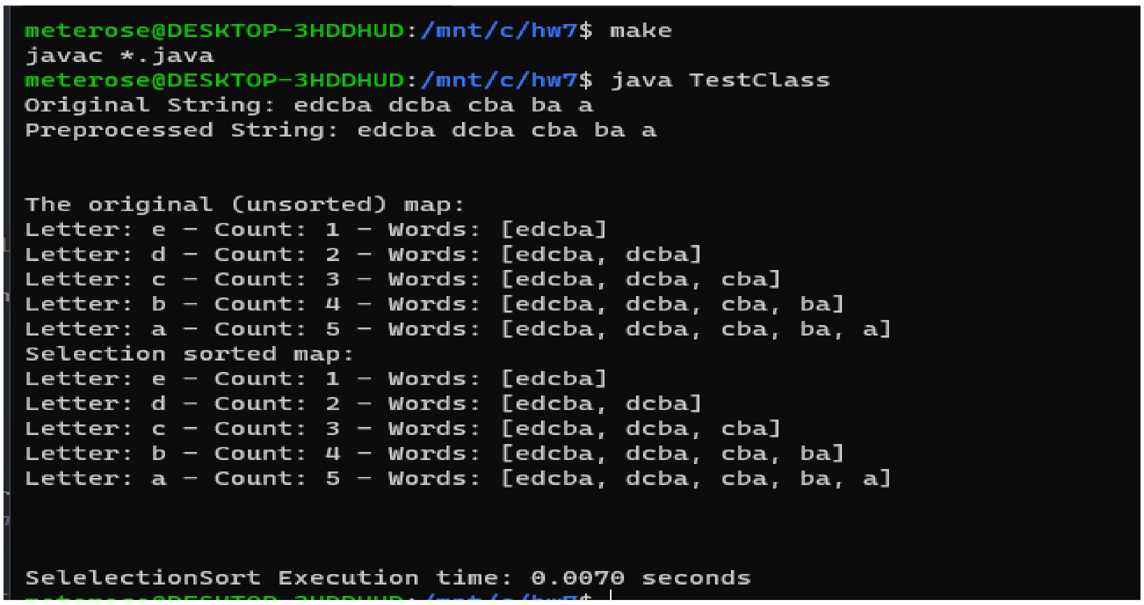
1-) Running time complexity of worst-case



```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ make
javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestClass
Original String: a ab abc abcd abcde
Preprocessed String: a ab abc abcd abcde


The original (unsorted) map:
Letter: a - Count: 5 - Words: [a, ab, abc, abcd, abcde]
Letter: b - Count: 4 - Words: [ab, abc, abcd, abcde]
Letter: c - Count: 3 - Words: [abc, abcd, abcde]
Letter: d - Count: 2 - Words: [abcd, abcde]
Letter: e - Count: 1 - Words: [abcde]
Selection sorted map:
Letter: e - Count: 1 - Words: [abcde]
Letter: d - Count: 2 - Words: [abcd, abcde]
Letter: c - Count: 3 - Words: [abc, abcd, abcde]
Letter: b - Count: 4 - Words: [ab, abc, abcd, abcde]
Letter: a - Count: 5 - Words: [a, ab, abc, abcd, abcde]



SelelectionSort Execution time: 0.0070 seconds
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ |
```

As you can see above the the string builds myMap in reverse order, and as it was mentioned above this is the worst-case scenario for the SelectionSort. **The running time of the worst-case scenario is 0.0070 seconds.**

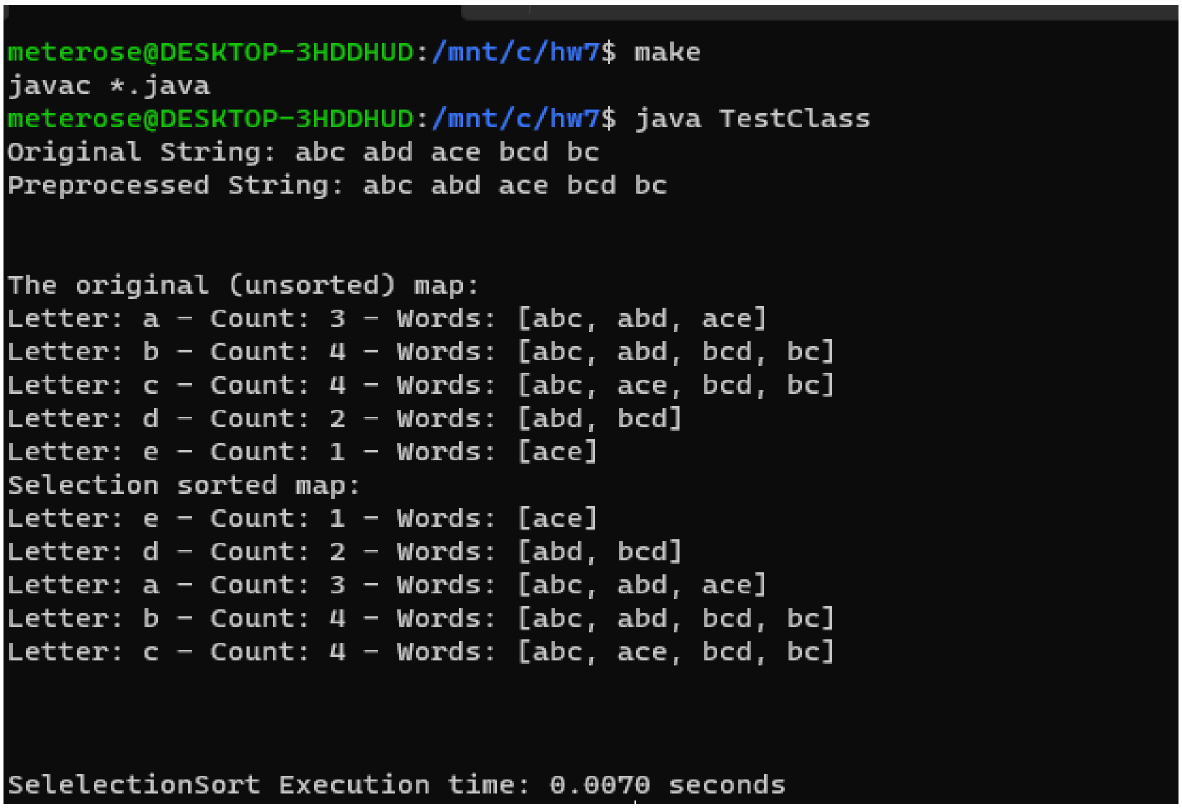
2-) Running time complexity of best-case



```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ make
javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestClass
Original String: edcba dcba cba ba a
Preprocessed String: edcba dcba cba ba a


The original (unsorted) map:
Letter: e - Count: 1 - Words: [edcba]
Letter: d - Count: 2 - Words: [edcba, dcba]
Letter: c - Count: 3 - Words: [edcba, dcba, cba]
Letter: b - Count: 4 - Words: [edcba, dcba, cba, ba]
Letter: a - Count: 5 - Words: [edcba, dcba, cba, ba, a]
Selection sorted map:
Letter: e - Count: 1 - Words: [edcba]
Letter: d - Count: 2 - Words: [edcba, dcba]
Letter: c - Count: 3 - Words: [edcba, dcba, cba]
Letter: b - Count: 4 - Words: [edcba, dcba, cba, ba]
Letter: a - Count: 5 - Words: [edcba, dcba, cba, ba, a]



SelelectionSort Execution time: 0.0070 seconds
```

In the best-case scenario the array is sorted therefore it is relatively easier to sort it. **The running time of the best-case scenario is 0.0070 seconds.**

3) Running time complexity of average-case

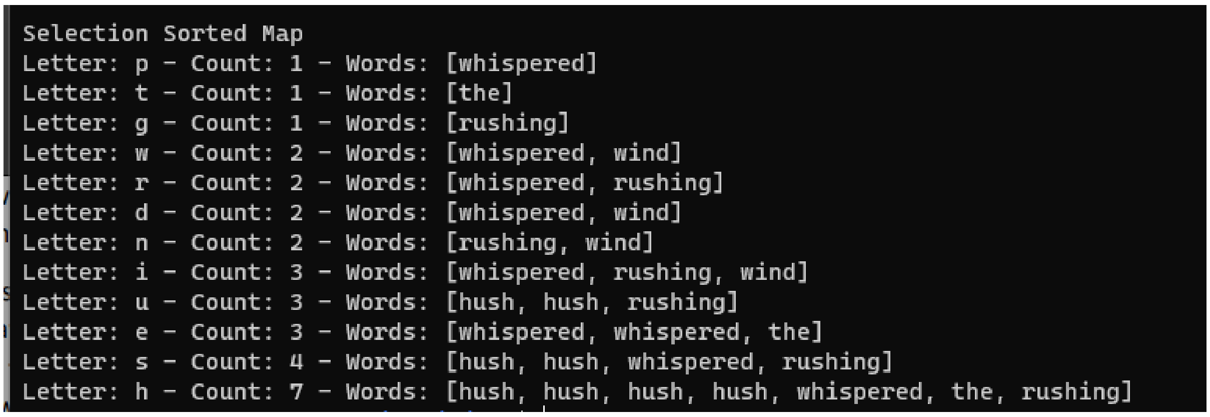


```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ make
javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestClass
Original String: abc abd ace bcd bc
Preprocessed String: abc abd ace bcd bc


The original (unsorted) map:
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: e - Count: 1 - Words: [ace]
Selection sorted map:
Letter: e - Count: 1 - Words: [ace]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]



SelelectionSort Execution time: 0.0070 seconds
```

In the average case, half of the array is sorted. **The running time of the average case scenario is 0.0070.**

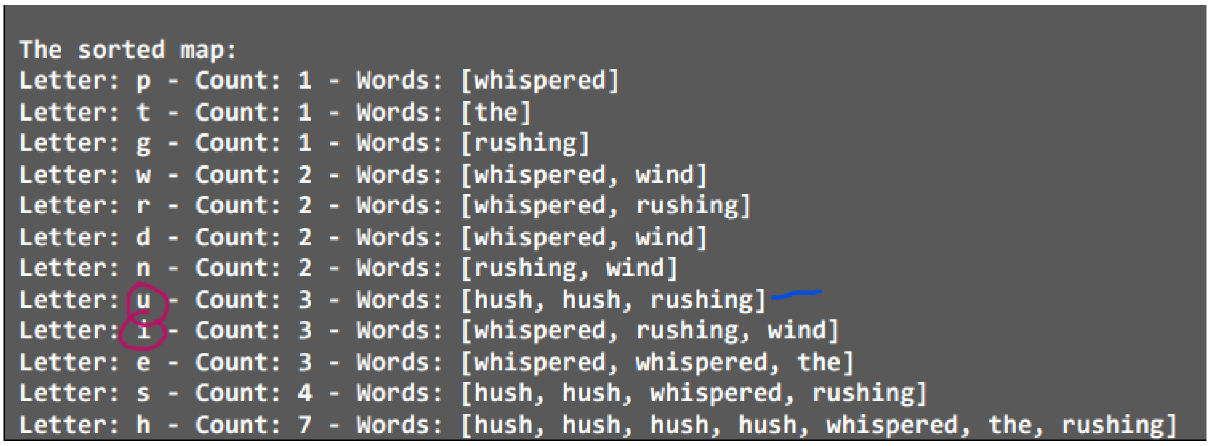
Since it is a small set of data input, running time of best, worst and average case are equal.

Analyze of whether SelectionSort keeps the Input Order (PartD)



```
Selection Sorted Map
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]
```

This is the result that has been sorted with Selection Sort and if you compare the result with the following expected input order below;



```
The sorted map:
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]
```

It is clear that, the key "u" and "i" are in the reverse order, **therefore we can say that selection order does not keep the input order.** The reason for is that, in each iteration it is just checking the smallest count value it the array, and when it finds it does not control it with the other entries that have the same count value. So it is arbitrarly gets the smallest count value, and therefore it does not eep the input order.

Insertion Sort

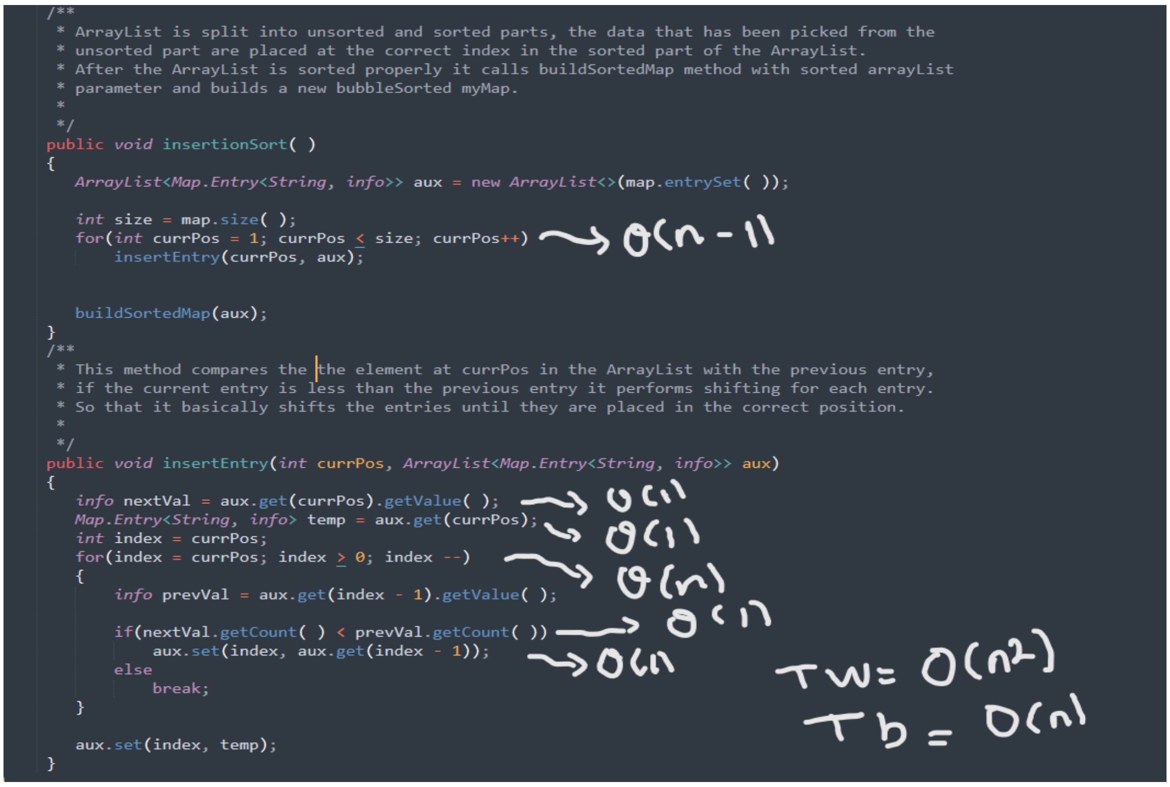
In Insertion sort the array that contains the entries of myMap object is split into unsorted and sorted parts, the data that has been picked from the unsorted part are placed at the correct index in the sorted part of the array.

Time Complexity Analysis for Insertion Sort

In the worst case scenario the insertion step is repeated n -1 times. In the worst case, all the elements in the sorted subarray are compared to nextVal for each insertion, so the maximum number of comparisons is represented by the series

$1 + 2 + 3 + ... + (n - 2) + (n - 1)$

Which is **O(n^2)** time complexity. In the best case(when array is sorted), there is only one comparisons is needed to be made for each iteration and it does not requires any swaps , so the number of comparisons in best-case scenario is **O(n)**



```java
/**
 * ArrayList is split into unsorted and sorted parts, the data that has been picked from the
 * unsorted part are placed at the correct index in the sorted part of the ArrayList.
 * After the ArrayList is sorted properly it calls buildSortedMap method with sorted arrayList
 * parameter and builds a new bubbleSorted myMap.
 *
 */
public void insertionSort( )
{
   ArrayList<Map.Entry<String, info>> aux = new ArrayList<>(map.entrySet( ));

   int size = map.size( );
   for(int currPos = 1; currPos <= size; currPos++)
       insertEntry(currPos, aux);


   buildSortedMap(aux);
}
/**
 * This method compares the the element at currPos in the ArrayList with the previous entry,
 * if the current entry is less than the previous entry it performs shifting for each entry.
 * So that it basically shifts the entries until they are placed in the correct position.
 *
 */
public void insertEntry(int currPos, ArrayList<Map.Entry<String, info>> aux)
{
   info nextVal = aux.get(currPos).getValue( );
   Map.Entry<String, info> temp = aux.get(currPos);
   int index = currPos;
   for(index = currPos; index >= 0; index --)
   {
       info prevVal = aux.get(index - 1).getValue( );

       if(nextVal.getCount( ) < prevVal.getCount( ))
           aux.set(index, aux.get(index - 1));
       else
           break;
   }

   aux.set(index, temp);
}
```

Best-case, Worst-case, and Average-case Time Complexities for InsertionSort

Best-case scenario: Tb(n) = O(n)

- If the array is already sorted, insertion sort makes only one comparison for each entry and it does not make any swaps. Therefore, the best-case time complexity of insertion sort is **O(n).**

Worst-case scenario: Tw(n) = O(n^2)

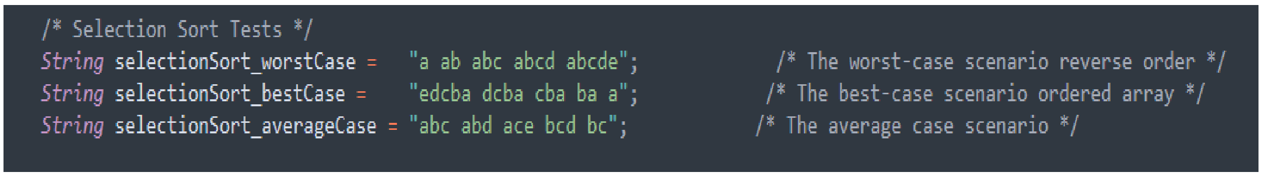
- The worst case occurs when the input array is in reverse order or contains elements in a compeletely unsorted. So the insertion sort rquires comparisons and swaps for each entry in array and this will result in O(n^2) time complexity.

Average-case scenario: T(n) = O(n^2)

- It is also O(n^2) like the the worst-case scenario because, insertion sort requires swaps and comparisons for at least half of the entries in the input array. Therefore the time complexity for average case is O(n^2).

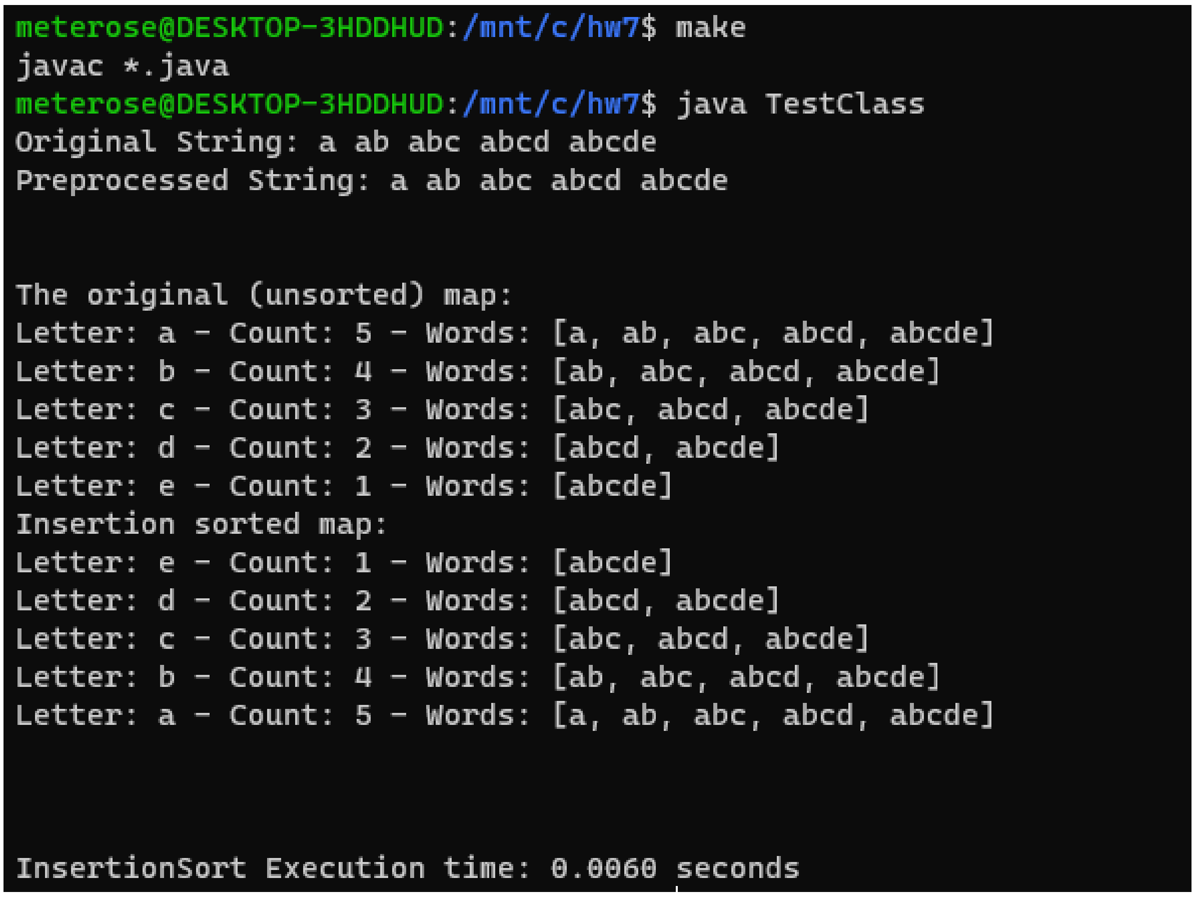
- **Running Results for Insertion Sort for Best-case, Worst-case, and Average-case inputs**

I will used the same input set that I used in the SelectionSort because it requires the same type of inputs to trigger best, worst and the average case.



```
/* Selection Sort Tests */
String selectionSort_worstCase =   "a ab abc abcd abcde";          /* The worst-case scenario reverse order */
String selectionSort_bestCase =    "edcba dcba cba ba a";          /* The best-case scenario ordered array */
String selectionSort_averageCase = "abc abd ace bcd bc";          /* The average case scenario */
```

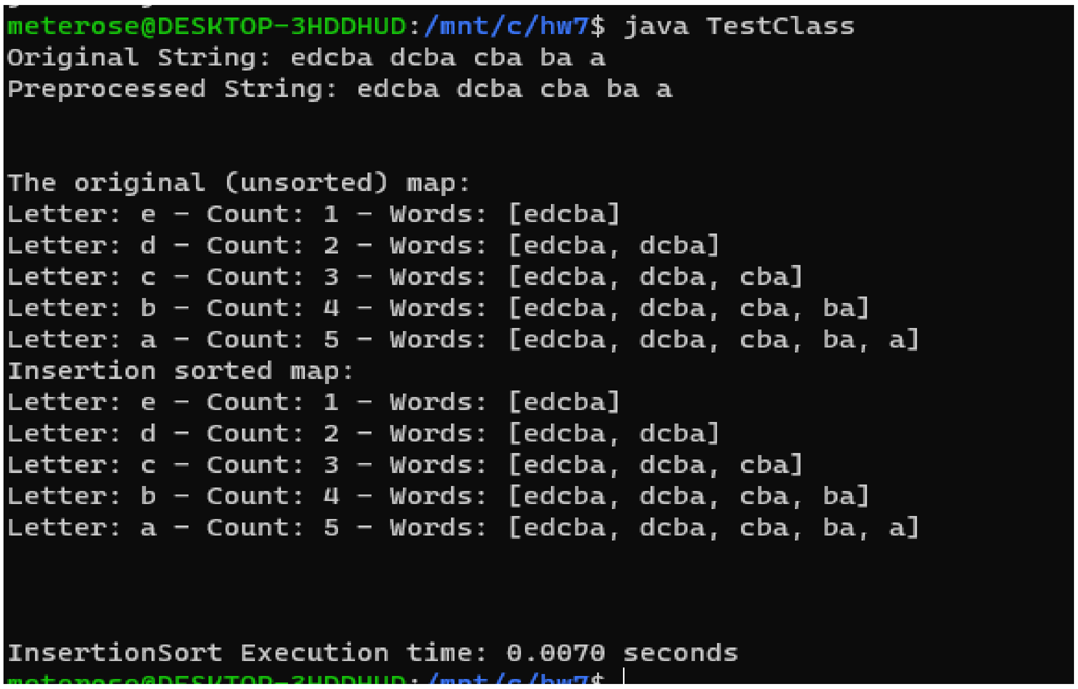
1-) Running time complexity of worst-case



```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ make
javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestClass
Original String: a ab abc abcd abcde
Preprocessed String: a ab abc abcd abcde


The original (unsorted) map:
Letter: a - Count: 5 - Words: [a, ab, abc, abcd, abcde]
Letter: b - Count: 4 - Words: [ab, abc, abcd, abcde]
Letter: c - Count: 3 - Words: [abc, abcd, abcde]
Letter: d - Count: 2 - Words: [abcd, abcde]
Letter: e - Count: 1 - Words: [abcde]
Insertion sorted map:
Letter: e - Count: 1 - Words: [abcde]
Letter: d - Count: 2 - Words: [abcd, abcde]
Letter: c - Count: 3 - Words: [abc, abcd, abcde]
Letter: b - Count: 4 - Words: [ab, abc, abcd, abcde]
Letter: a - Count: 5 - Words: [a, ab, abc, abcd, abcde]



InsertionSort Execution time: 0.0060 seconds
```

In worst-case when array is in reverse order sorted. **The running time is 0.0060.**

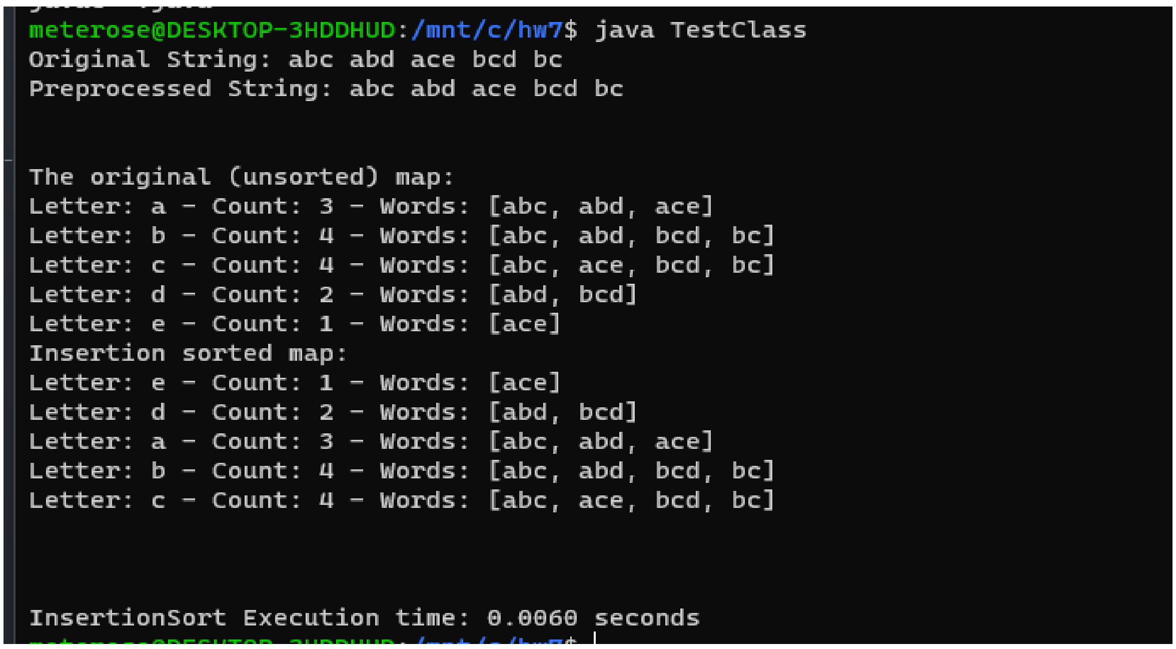
2-) Running time complexity of best-case



```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestClass
Original String: edcba dcba cba ba a
Preprocessed String: edcba dcba cba ba a


The original (unsorted) map:
Letter: e - Count: 1 - Words: [edcba]
Letter: d - Count: 2 - Words: [edcba, dcba]
Letter: c - Count: 3 - Words: [edcba, dcba, cba]
Letter: b - Count: 4 - Words: [edcba, dcba, cba, ba]
Letter: a - Count: 5 - Words: [edcba, dcba, cba, ba, a]
Insertion sorted map:
Letter: e - Count: 1 - Words: [edcba]
Letter: d - Count: 2 - Words: [edcba, dcba]
Letter: c - Count: 3 - Words: [edcba, dcba, cba]
Letter: b - Count: 4 - Words: [edcba, dcba, cba, ba]
Letter: a - Count: 5 - Words: [edcba, dcba, cba, ba, a]



InsertionSort Execution time: 0.0070 seconds
```

In the best-case when array is already sorted, **the running time is 0.0070.**

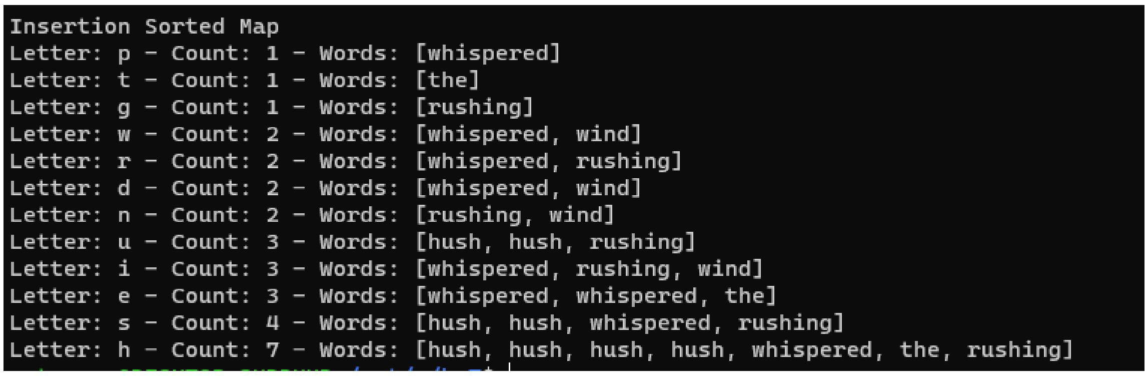
3-) Running time complexity of average-case



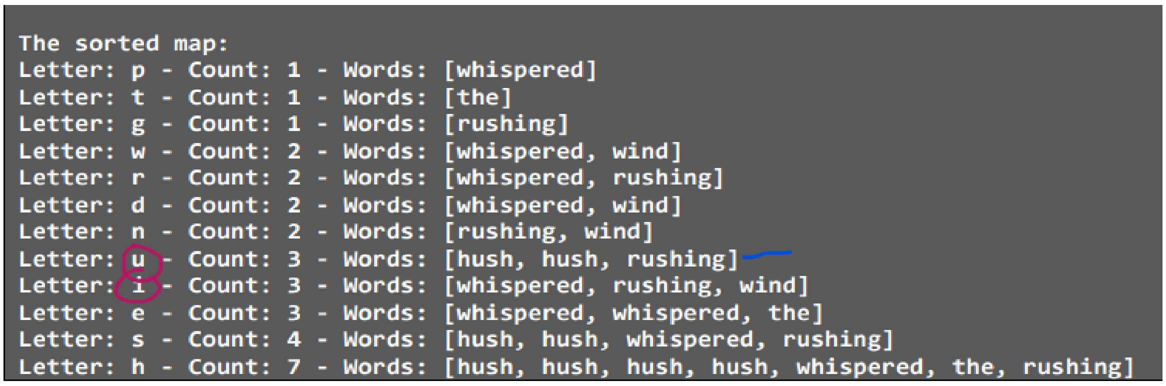
```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestClass
Original String: abc abd ace bcd bc
Preprocessed String: abc abd ace bcd bc


The original (unsorted) map:
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: e - Count: 1 - Words: [ace]
Insertion sorted map:
Letter: e - Count: 1 - Words: [ace]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]



InsertionSort Execution time: 0.0060 seconds
```

In the average case running time is 0.0060.

Analyze of whether Insertion sort keeps the Input Order



```
Insertion Sorted Map
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]
```



```
The sorted map:
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]
```

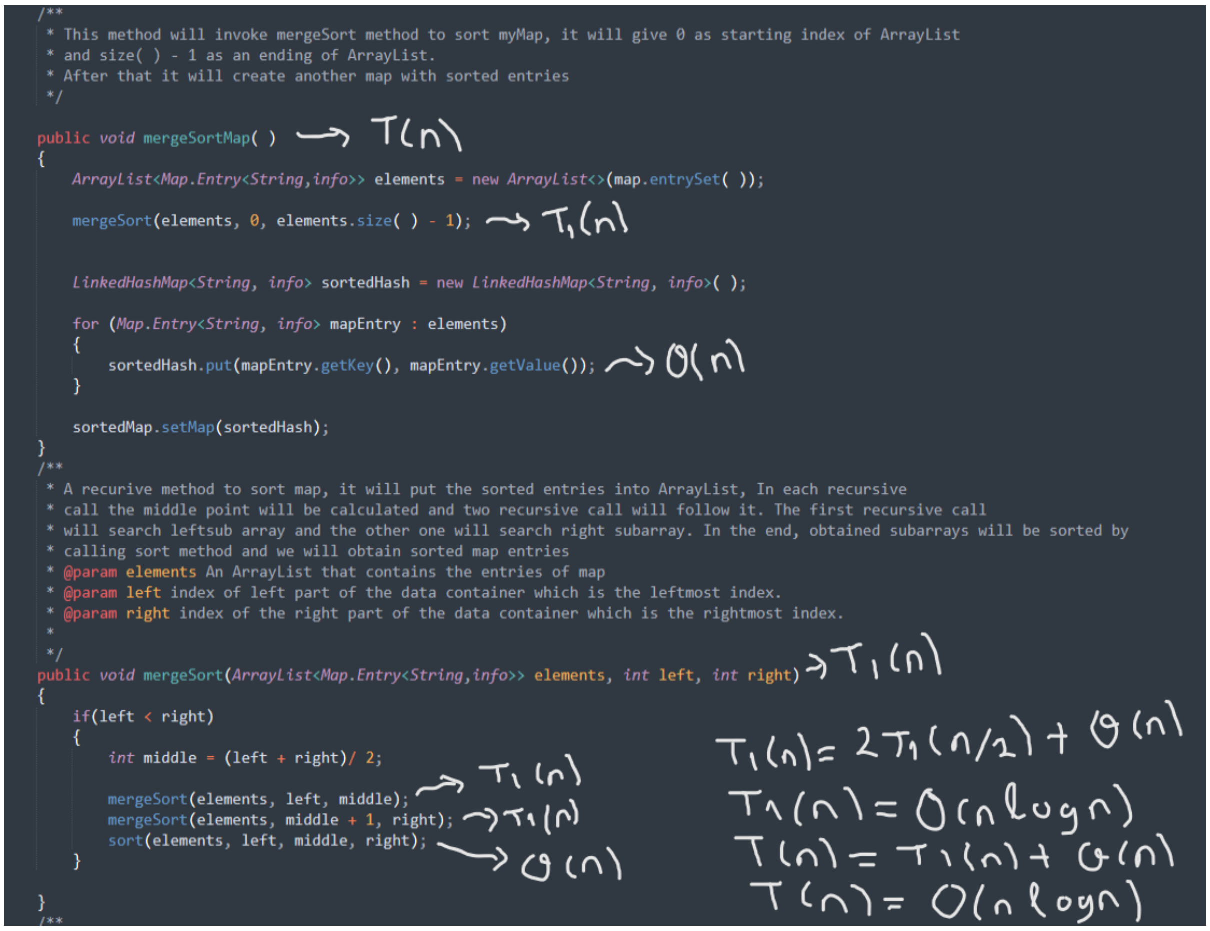
As you can see above, **the insertion sort keeps the input order**. The reason for that is that insertion sort always inserst each entries into to the correct position right after previously sorted entries. Therefore, it maintains the original order of equal entries. Insertion sort keeps the input order because it repedately inserts entries to their correct position in the sorted section.

Merge Sort

Merge sort is basically divides the array into two halves and sort each half and after that it merges them. So it divides the array into smaller subarrays, after that sorts each subarray and in the end merges the sorted subarrays back together to build the final sorted array.

Time Complexity Analysis for Merge Sort

So in the previous two sorting algorithms we get O(n^2) time complexities, the advantageous of merge sort algorithm over these two previous algorithm is the time complexity of MergeSort which is **O(nlogn).** This means that when it comes the large size input it sorts them quickly. Therefore it is a good choice for sorting large datasets.



```
/**
 * This method will invoke mergeSort method to sort myMap, it will give 0 as starting index of ArrayList
 * and size( ) - 1 as an ending of ArrayList.
 * After that it will create another map with sorted entries
 */

public void mergeSortMap( )  → T(n)
{
    ArrayList<Map.Entry<String,info>> elements = new ArrayList<>(map.entrySet( ));

    mergeSort(elements, 0, elements.size( ) - 1);  → T1(n)


    LinkedHashMap<String, info> sortedHash = new LinkedHashMap<String, info>( );

    for (Map.Entry<String, info> mapEntry : elements)
    {
        sortedHash.put(mapEntry.getKey(), mapEntry.getValue());  → O(n)
    }

    sortedMap.setMap(sortedHash);
}
/**
 * A recurive method to sort map, it will put the sorted entries into ArrayList, In each recursive
 * call the middle point will be calculated and two recursive call will follow it. The first recursive call
 * will search leftsub array and the other one will search right subarray. In the end, obtained subarrays will be sorted by
 * calling sort method and we will obtain sorted map entries
 * @param elements An ArrayList that contains the entries of map
 * @param left index of left part of the data container which is the leftmost index.
 * @param right index of the right part of the data container which is the rightmost index.
 *
 */
public void mergeSort(ArrayList<Map.Entry<String,info>> elements, int left, int right) → T1(n)
{
    if(left < right)
    {
        int middle = (left + right)/ 2;

        mergeSort(elements, left, middle);  → T1(n)
        mergeSort(elements, middle + 1, right);  → T1(n)
        sort(elements, left, middle, right);  → O(n)
    }

}
/**
```

$$T_1(n) = 2T_1(n/2) + O(n)$$
$$T_1(n) = O(n \log n)$$
$$T(n) = T_1(n) + O(n)$$
$$T(n) = O(n \log n)$$

So as you can see above, the time complexity of mergeSort algorithm becomes a reccurence relation which is **T(n) = 2T(n/2) + n,** when we solve it by using master theorem we will get **T(n) = O(n logn)** as the time complexity. The sort function in mergeSort just sorts the both left and right subarrays therefore it has a O(n) time complexity.

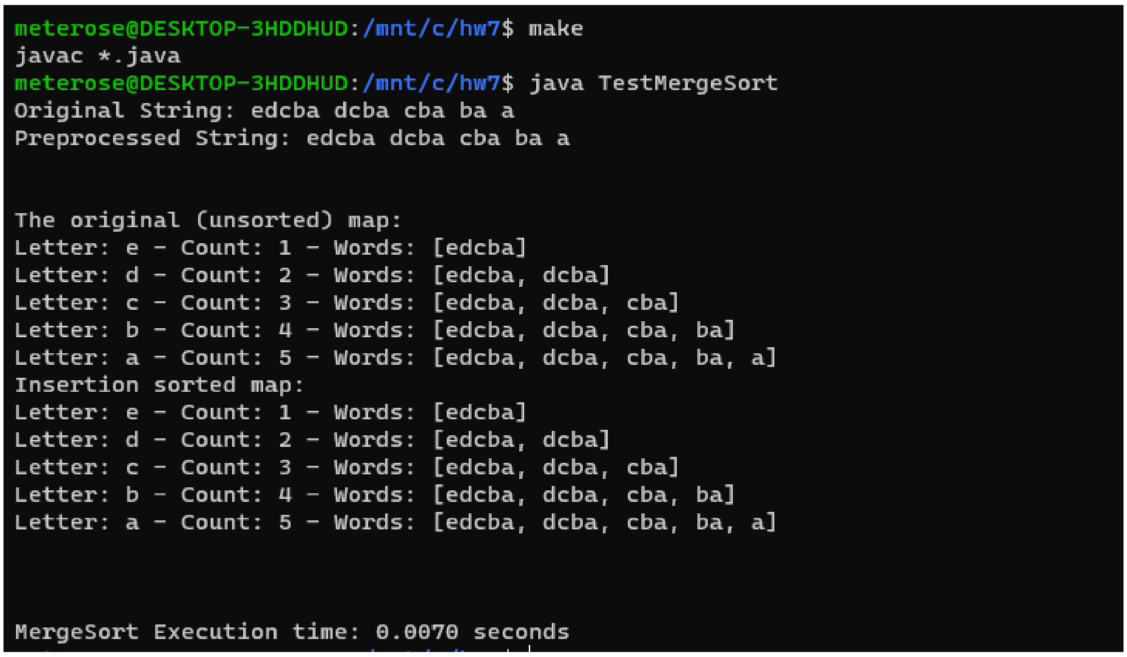
Best-case, Worst-case, and Average-case Time Complexities for MergeSort

The time complexity of Merge Sort is the same for average-case, worst-case scenario and best-case scenario because merge sort always divides the array into two halves and to sort and merge the each halves takes linear time. Therefore time complexity of merge sort for each scenarios is **O(n.logn)**

Best-case scenario: Tb(n) = O(n logn)
Worst-case scenario: Tw(n) = O(n logn)
Average-case scenario: T(n) = O(n logn)

Running Results for MergeSort for Best-case, Worst-case, and Average-case inputs

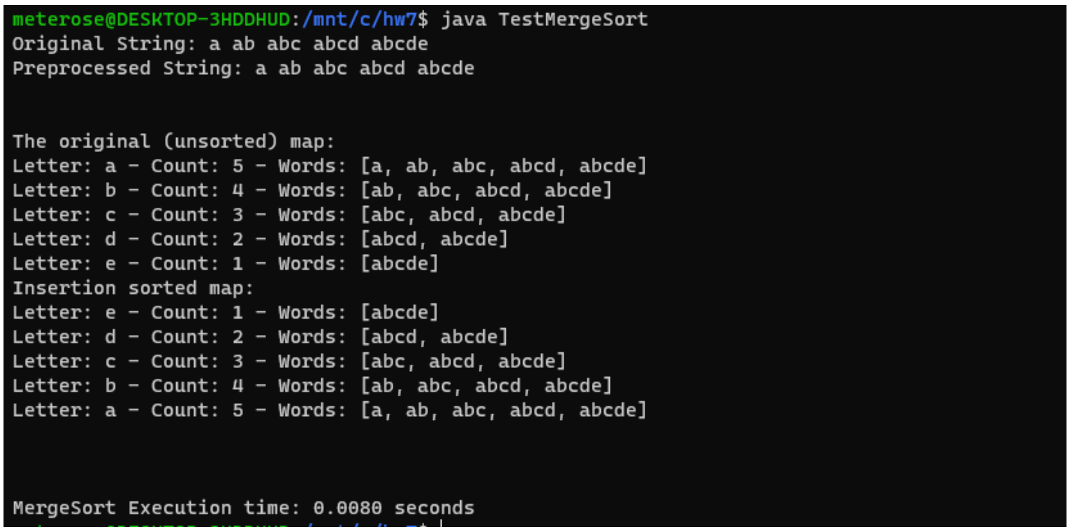
1-) Running time complexity of best-case



```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ make
javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestMergeSort
Original String: edcba dcba cba ba a
Preprocessed String: edcba dcba cba ba a


The original (unsorted) map:
Letter: e - Count: 1 - Words: [edcba]
Letter: d - Count: 2 - Words: [edcba, dcba]
Letter: c - Count: 3 - Words: [edcba, dcba, cba]
Letter: b - Count: 4 - Words: [edcba, dcba, cba, ba]
Letter: a - Count: 5 - Words: [edcba, dcba, cba, ba, a]
Insertion sorted map:
Letter: e - Count: 1 - Words: [edcba]
Letter: d - Count: 2 - Words: [edcba, dcba]
Letter: c - Count: 3 - Words: [edcba, dcba, cba]
Letter: b - Count: 4 - Words: [edcba, dcba, cba, ba]
Letter: a - Count: 5 - Words: [edcba, dcba, cba, ba, a]



MergeSort Execution time: 0.0070 seconds
```

The running time of the best case is **0.0070.**

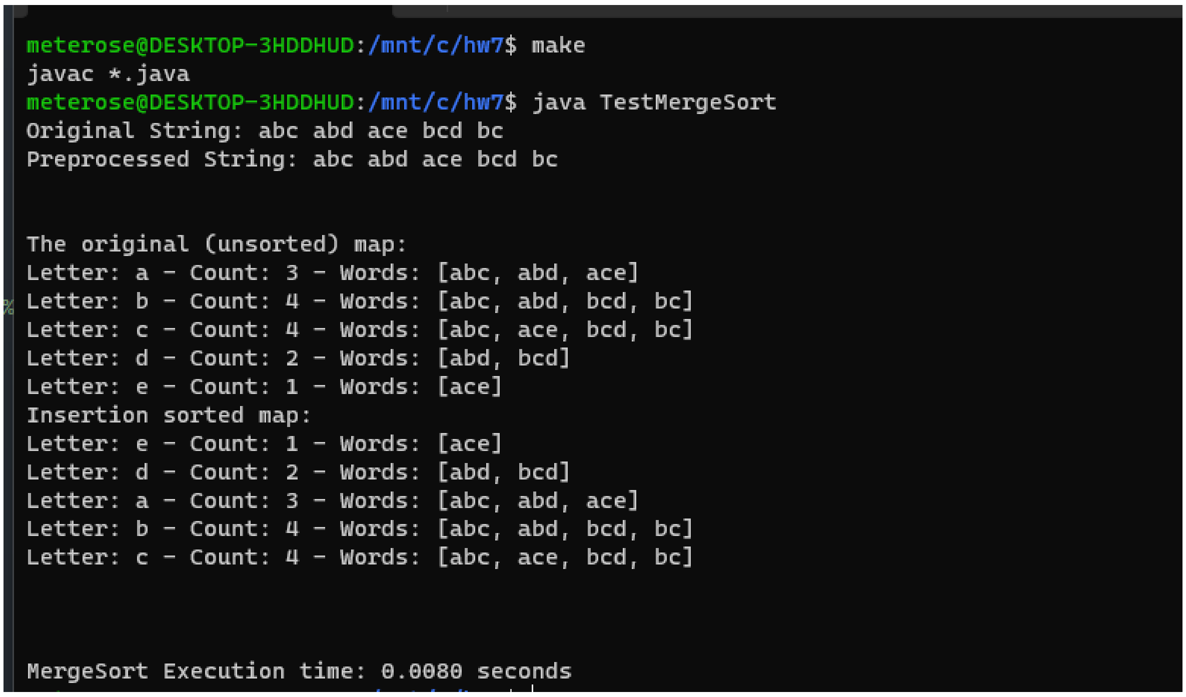
2-) Running time complexity of worst-case



```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestMergeSort
Original String: a ab abc abcd abcde
Preprocessed String: a ab abc abcd abcde


The original (unsorted) map:
Letter: a - Count: 5 - Words: [a, ab, abc, abcd, abcde]
Letter: b - Count: 4 - Words: [ab, abc, abcd, abcde]
Letter: c - Count: 3 - Words: [abc, abcd, abcde]
Letter: d - Count: 2 - Words: [abcd, abcde]
Letter: e - Count: 1 - Words: [abcde]
Insertion sorted map:
Letter: e - Count: 1 - Words: [abcde]
Letter: d - Count: 2 - Words: [abcd, abcde]
Letter: c - Count: 3 - Words: [abc, abcd, abcde]
Letter: b - Count: 4 - Words: [ab, abc, abcd, abcde]
Letter: a - Count: 5 - Words: [a, ab, abc, abcd, abcde]



MergeSort Execution time: 0.0080 seconds
```

The running time of the worst case is **0.0080.**

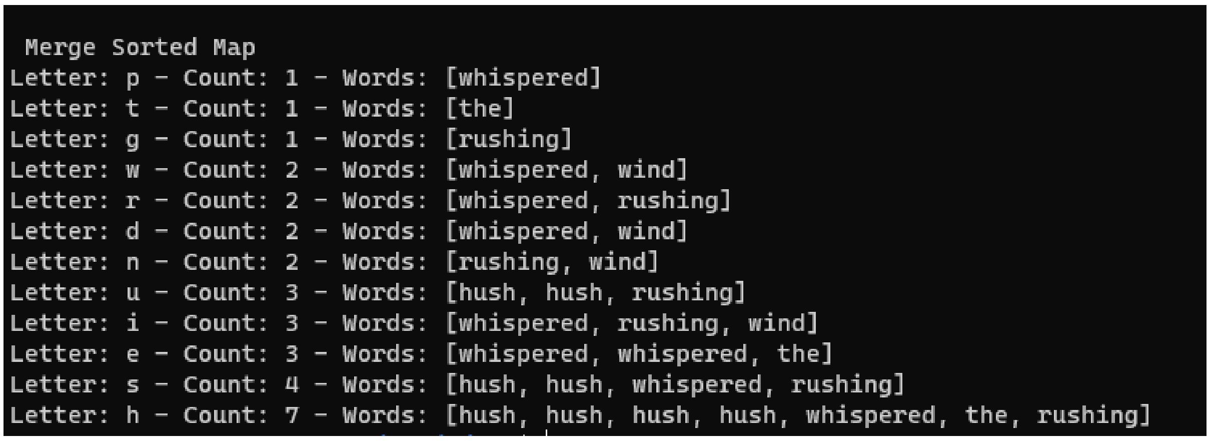
3-) Running time complexity of average-case



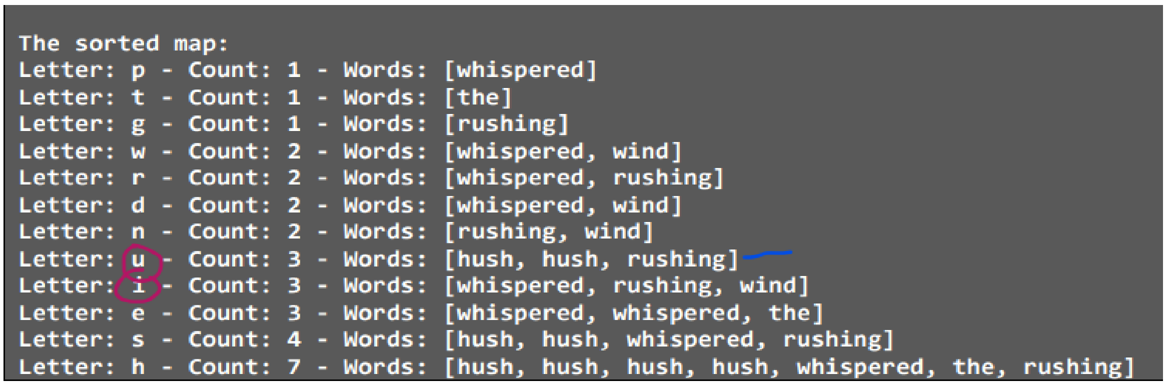
```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ make
javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestMergeSort
Original String: abc abd ace bcd bc
Preprocessed String: abc abd ace bcd bc


The original (unsorted) map:
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: e - Count: 1 - Words: [ace]
Insertion sorted map:
Letter: e - Count: 1 - Words: [ace]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]



MergeSort Execution time: 0.0080 seconds
```

The running time for the average-case is **0.0080.**

Analyze of whether Merge sort keeps the Input Order (PartD)



```
 Merge Sorted Map
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]
```



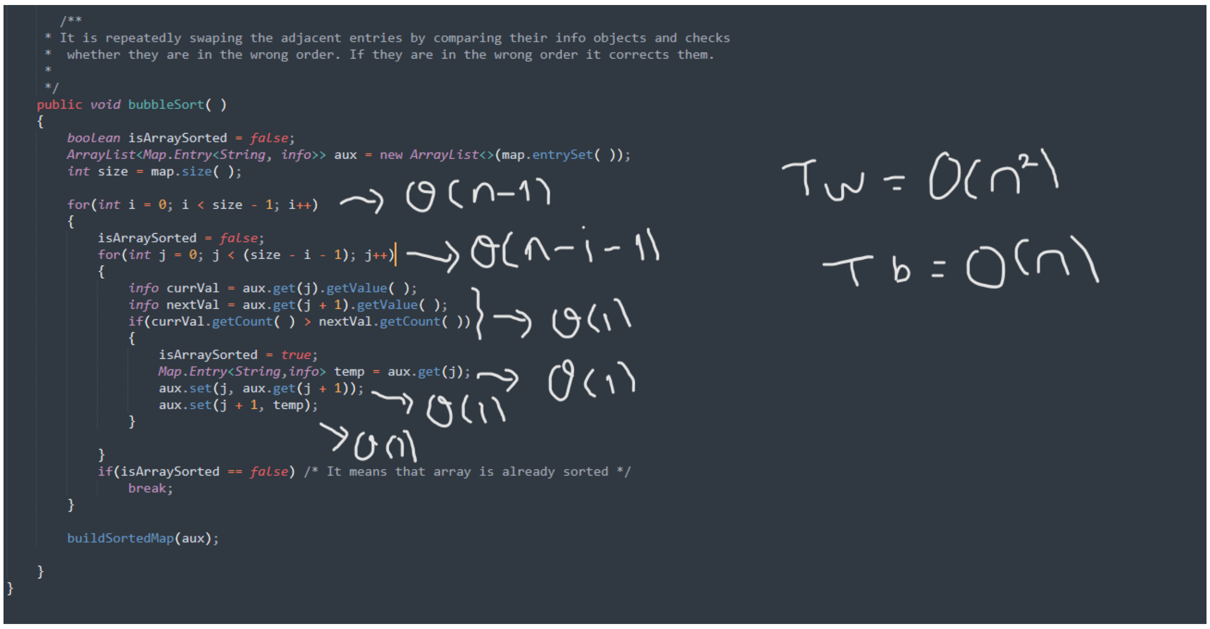
```
The sorted map:
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 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]
```

As you can see above, the merge sort algorithm keeps the input order, merge sort operation combines the smaller sorted subarrays into a larger sorted array is the way to maintaining the input order. While merging process is happening, the algorithm compares the entries from two subarrays and selects the smaller entry to be placed into the actual merged array. If two entry has the same value, the algorithm ensures that the element from left array is placed before the element from the right array because mergesort is a stable sorting algorithm. **Therefore, the merge sort keeps the input order.**

Bubble Sort

Bubble sorts the elements by repeatedely swapping the adjacent elements if they are in the incorrect order. For the larger data sets it is not an efficient algorithm, but when it comes to small input size it works properly.

Time Complexity Analysis for Bubble Sort



```
    /**
   * It is repeatedly swaping the adjacent entries by comparing their info objects and checks
   *  whether they are in the wrong order. If they are in the wrong order it corrects them.
   *
   */
  public void bubbleSort( )
  {
      boolean isArraySorted = false;
      ArrayList<Map.Entry<String, info>> aux = new ArrayList<>(map.entrySet( ));
      int size = map.size( );

      for(int i = 0; i < size - 1; i++)
      {
          isArraySorted = false;
          for(int j = 0; j < (size - i - 1); j++)
          {
              info currVal = aux.get(j).getValue( );
              info nextVal = aux.get(j + 1).getValue( );
              if(currVal.getCount( ) > nextVal.getCount( ))
              {
                  isArraySorted = true;
                  Map.Entry<String,info> temp = aux.get(j);
                  aux.set(j, aux.get(j + 1));
                  aux.set(j + 1, temp);
              }

          }
          if(isArraySorted == false) /* It means that array is already sorted */
              break;
      }

      buildSortedMap(aux);

  }
}
```

So when you examine the method above, you will see that when i is 0 the number of executions is (n -1), when i is 1 the number of executions becomes n – 2, in the end when i is equal to size – 2 the number of execution becomes 1, So we have the series of T(n) = (n-1) + (n-2) + (n-3) + …. + 1, when we calculate an remove the all terms other than most significant terms we will get **O(n^2) time complexity.** However, as you can see above when array is sorted, the outer loop does not iterates because of the boolean variable isArraySorted. So the in the best case when array is already sorted it does not iterate because the boolean isArraySorted value becomes true and it breaks the outer loop. Therefore the time complexity in **best-case scenario is O(n).**

Best-case, Worst-case, and Average-case Time Complexities for InsertionSort

Best-case scenario: Tb(n) = O(n)

- If the array is already sorted, the boolean isArraySorted value becomes true and breaks the outer loop, and it becomes O(n)

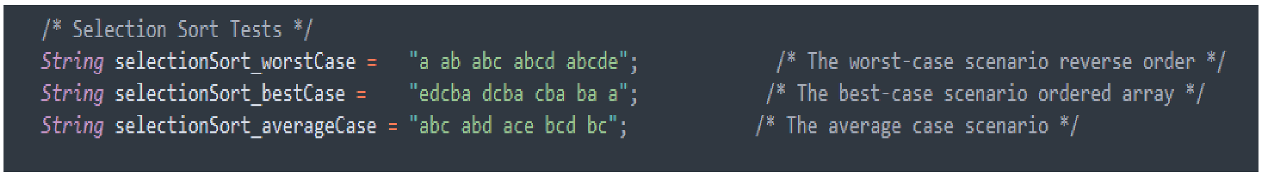
Worst-case scenario: Tw(n) = O(n^2)

- The worst case occurs when the input array is in reverse order.

Average-case scenario: T(n) = O(n^2)

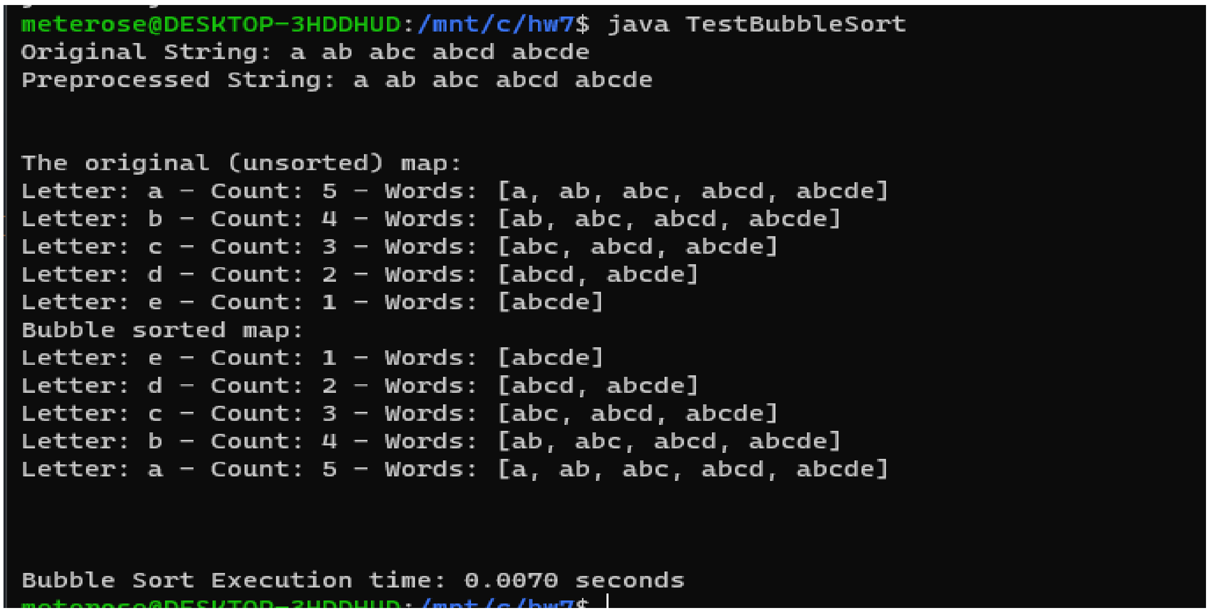
- It is also O(n^2) like the the worst-case scenario because, both loop iterates for each entry.

Running Results for BubblSort for Best-case, Worst-case, and Average-case inputs



```
/* Selection Sort Tests */
String selectionSort_worstCase =   "a ab abc abcd abcde";          /* The worst-case scenario reverse order */
String selectionSort_bestCase =    "edcba dcba cba ba a";         /* The best-case scenario ordered array */
String selectionSort_averageCase = "abc abd ace bcd bc";         /* The average case scenario */
```

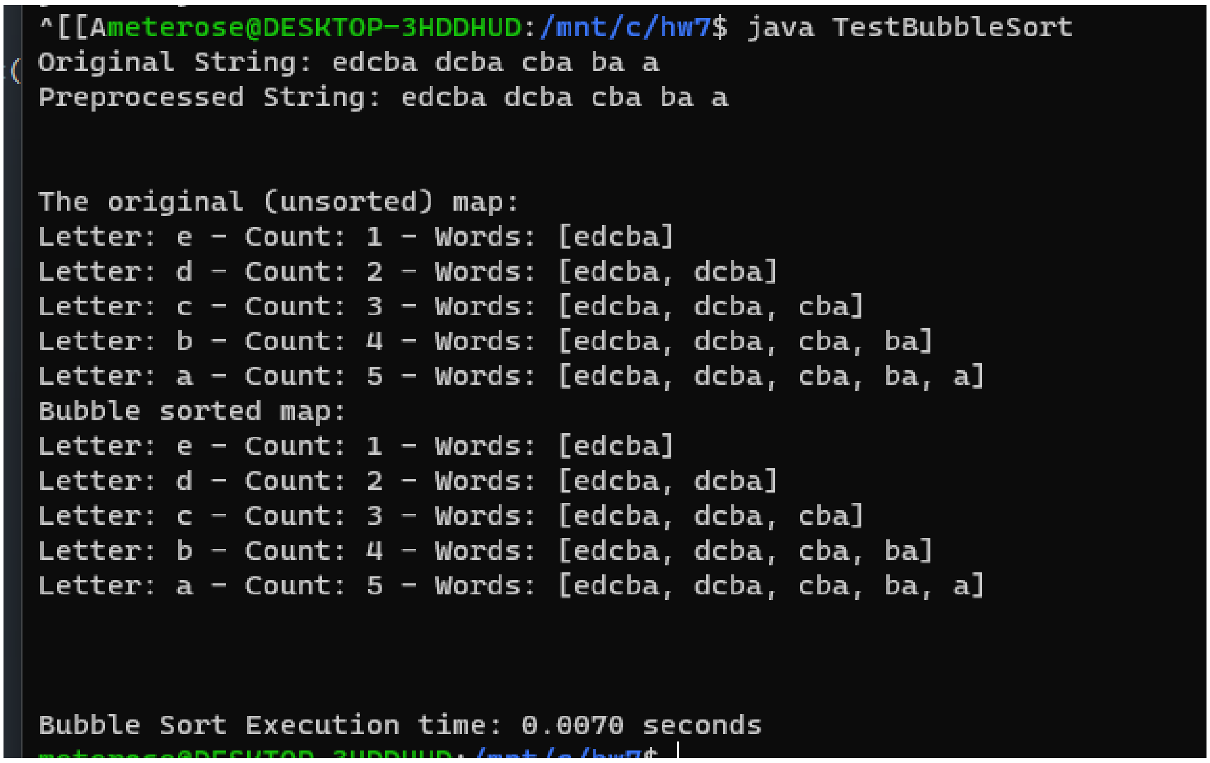
1-) Running time complexity of worst-case



```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestBubbleSort
Original String: a ab abc abcd abcde
Preprocessed String: a ab abc abcd abcde


The original (unsorted) map:
Letter: a - Count: 5 - Words: [a, ab, abc, abcd, abcde]
Letter: b - Count: 4 - Words: [ab, abc, abcd, abcde]
Letter: c - Count: 3 - Words: [abc, abcd, abcde]
Letter: d - Count: 2 - Words: [abcd, abcde]
Letter: e - Count: 1 - Words: [abcde]
Bubble sorted map:
Letter: e - Count: 1 - Words: [abcde]
Letter: d - Count: 2 - Words: [abcd, abcde]
Letter: c - Count: 3 - Words: [abc, abcd, abcde]
Letter: b - Count: 4 - Words: [ab, abc, abcd, abcde]
Letter: a - Count: 5 - Words: [a, ab, abc, abcd, abcde]



Bubble Sort Execution time: 0.0070 seconds
```

The running time for worst case is **0.0070.**

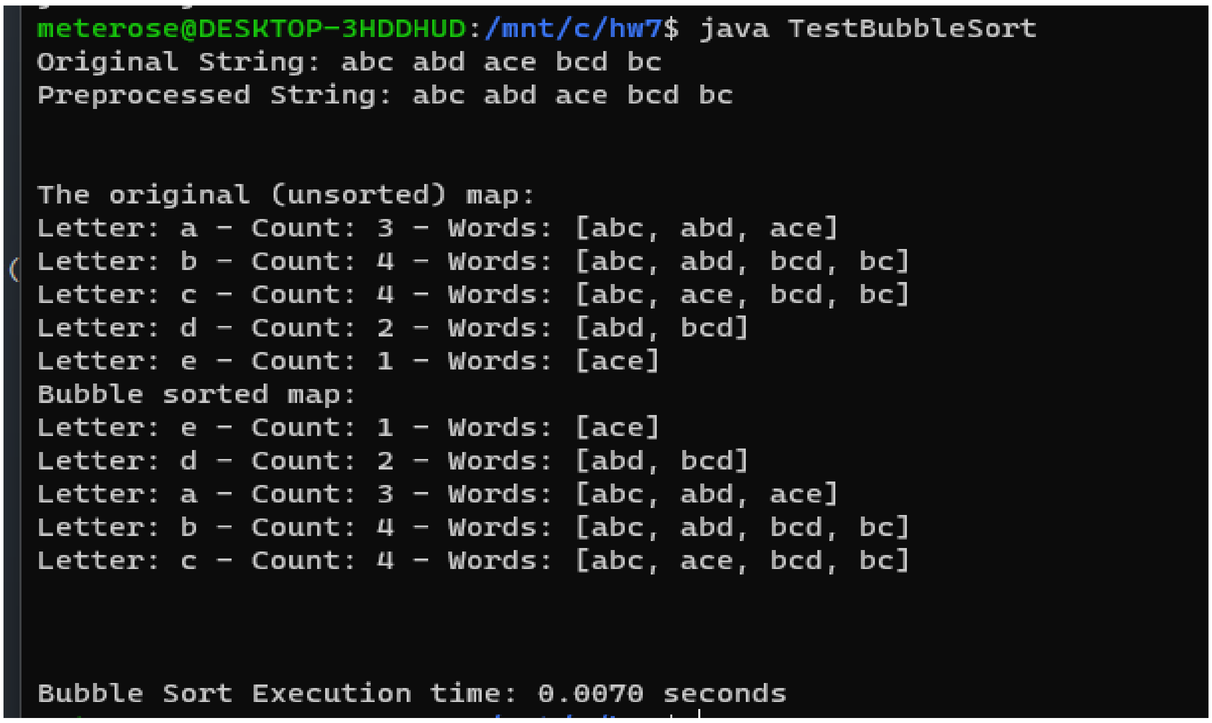
2-) Running time complexity of best-case



```
^[[Ameterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestBubbleSort
Original String: edcba dcba cba ba a
Preprocessed String: edcba dcba cba ba a


The original (unsorted) map:
Letter: e - Count: 1 - Words: [edcba]
Letter: d - Count: 2 - Words: [edcba, dcba]
Letter: c - Count: 3 - Words: [edcba, dcba, cba]
Letter: b - Count: 4 - Words: [edcba, dcba, cba, ba]
Letter: a - Count: 5 - Words: [edcba, dcba, cba, ba, a]
Bubble sorted map:
Letter: e - Count: 1 - Words: [edcba]
Letter: d - Count: 2 - Words: [edcba, dcba]
Letter: c - Count: 3 - Words: [edcba, dcba, cba]
Letter: b - Count: 4 - Words: [edcba, dcba, cba, ba]
Letter: a - Count: 5 - Words: [edcba, dcba, cba, ba, a]



Bubble Sort Execution time: 0.0070 seconds
```

The running time for worst case is **0.0070.**

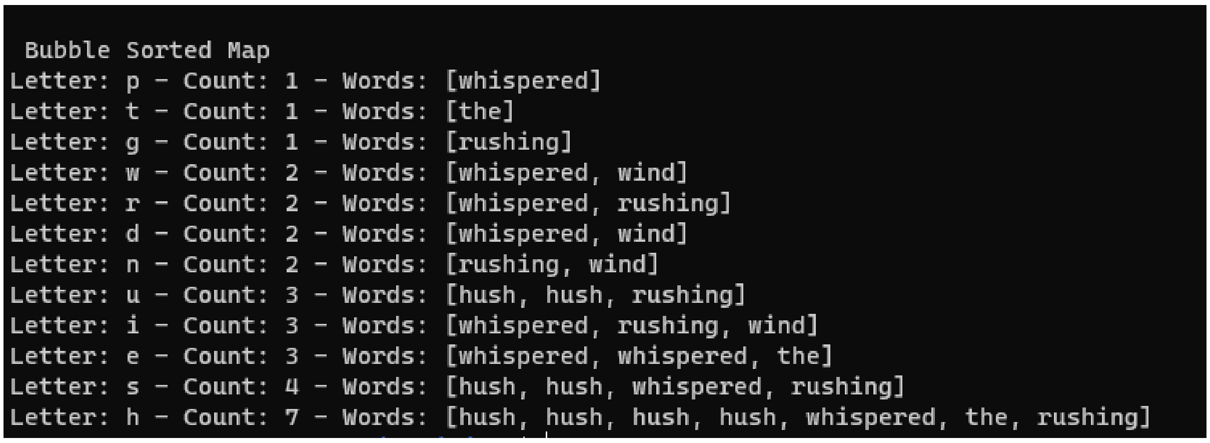
3-) Running time complexity of average-case



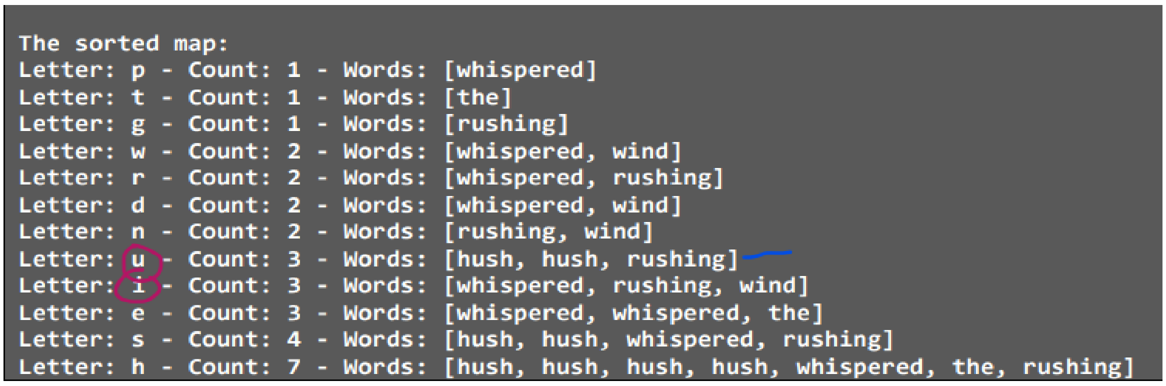
```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestBubbleSort
Original String: abc abd ace bcd bc
Preprocessed String: abc abd ace bcd bc


The original (unsorted) map:
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: e - Count: 1 - Words: [ace]
Bubble sorted map:
Letter: e - Count: 1 - Words: [ace]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]



Bubble Sort Execution time: 0.0070 seconds
```

The running time for average case is **0.0070.**

Analyze of whether Bubble sort keeps the Input Order (PartD)



```
 Bubble Sorted Map
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]
```



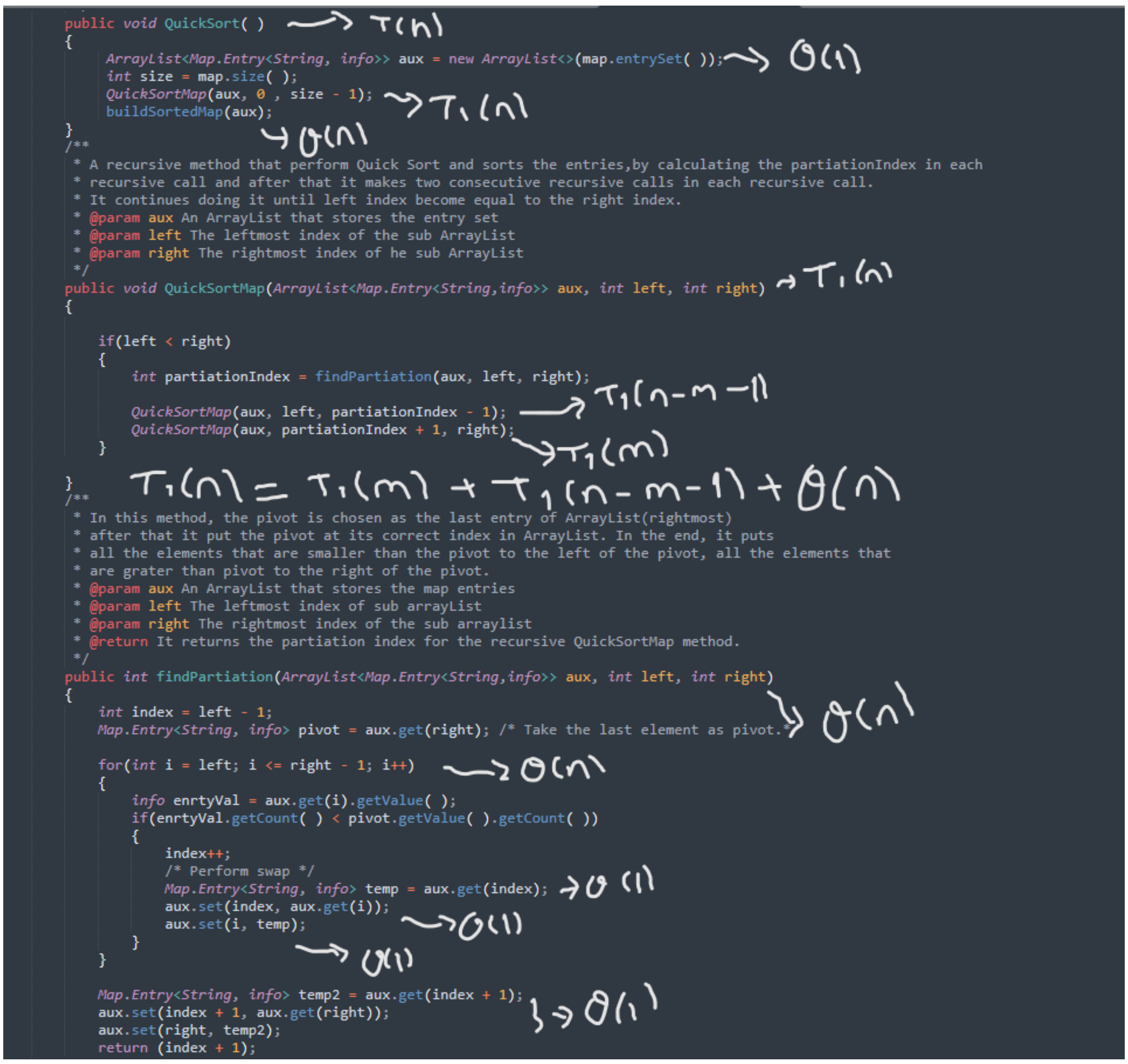
```
The sorted map:
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 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]
```

As you can see above, **the bubble sort algorithm keeps the input order.** Because it only swaps and compares the adjacent elements, starting from the beginning of the array and compares first pair, swaps them if necessary. After that it moves to the next pair of adjacent entry and repeadts that process. In other words, bubble sort maintains the input order because it only compares and swaps the adjacent entries, it doesn't change the order of equal elements.

Quick Sort

This is the last sorting algorithm that I implemented, It is also known as divide and conquerer sorting algorithm. In the quick sort algorithm that main thing is partionaning the array. We perform partitions to place the pivot in its correct index in the sorted array and put all the entries that are smaller than the pivot to the left of the pivot, and also put all the elemets that are greater than the pivot to the right of the pivot. This operation(partition) is performed recursively that finally sorts the array. Any item can be pivot but the way that we choose pivot affect time complexity of QuickSort algorithm.

Time Complexity Analysis for Quick Sort



```java
public void QuickSort( )
{
    ArrayList<Map.Entry<String, info>> aux = new ArrayList<>(map.entrySet( ));
    int size = map.size( );
    QuickSortMap(aux, 0 , size - 1);
    buildSortedMap(aux);
}
/**
 * A recursive method that perform Quick Sort and sorts the entries,by calculating the partiationIndex in each
 * recursive call and after that it makes two consecutive recursive calls in each recursive call.
 * It continues doing it until left index become equal to the right index.
 * @param aux An ArrayList that stores the entry set
 * @param left The leftmost index of the sub ArrayList
 * @param right The rightmost index of he sub ArrayList
 */
public void QuickSortMap(ArrayList<Map.Entry<String,info>> aux, int left, int right)
{

    if(left < right)
    {
        int partiationIndex = findPartiation(aux, left, right);

        QuickSortMap(aux, left, partiationIndex - 1);
        QuickSortMap(aux, partiationIndex + 1, right);
    }

}
/**
 * In this method, the pivot is chosen as the last entry of ArrayList(rightmost)
 * after that it put the pivot at its correct index in ArrayList. In the end, it puts
 * all the elements that are smaller than the pivot to the left of the pivot, all the elements that
 * are grater than pivot to the right of the pivot.
 * @param aux An ArrayList that stores the map entries
 * @param left The leftmost index of sub arrayList
 * @param right The rightmost index of the sub arraylist
 * @return It returns the partiation index for the recursive QuickSortMap method.
 */
public int findPartiation(ArrayList<Map.Entry<String,info>> aux, int left, int right)
{
    int index = left - 1;
    Map.Entry<String, info> pivot = aux.get(right); /* Take the last element as pivot.*/

    for(int i = left; i <= right - 1; i++)
    {
        info enrtyVal = aux.get(i).getValue( );
        if(enrtyVal.getCount( ) < pivot.getValue( ).getCount( ))
        {
            index++;
            /* Perform swap */
            Map.Entry<String, info> temp = aux.get(index);
            aux.set(index, aux.get(i));
            aux.set(i, temp);
        }
    }

    Map.Entry<String, info> temp2 = aux.get(index + 1);
    aux.set(index + 1, aux.get(right));
    aux.set(right, temp2);
    return (index + 1);
```

As you can see above, the time complexity of T(n) becomes;

$$\mathbf{T(n) = T(m) + T(n - m - 1) + O(n)}$$

The time complexity generally is the result of this reccurence relation but it changes according to the pivot. So the way that we chose pivot affects the time complexity of the QuickSort algorithm**. In the above equation the m represent the number of the entries that are smaller than the pivot, the O(n) is the operation for partiation.**

Best-case scenario: Tb(n) = O(n.logn)

- This scenario occurs when the partiation process always picks the middle element as pivot by doing this it makes the reccurence equation as follows T(n) = 2T(N/2) + O(n) and it will result in with O (N.logN)

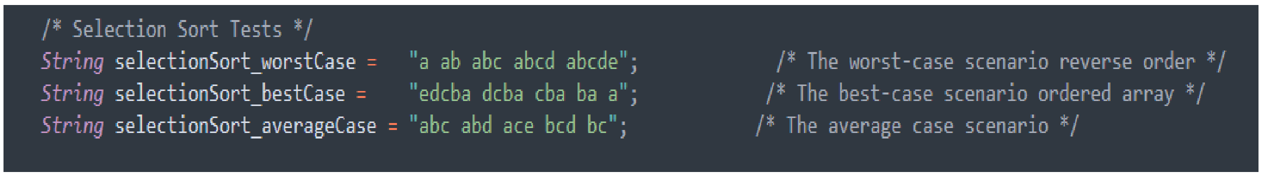
Worst-case scenario: Tw(n) = O(n^2)

- The worst case scenario happens when partiation **process choose always the first or the last element as pivot,** in my implementation I chose the last entry as the pivot each time, therefore the worst case would occur then te array is alread sorted. The time complexity of worst-case scenario is O(n^2).

Average-case scenario: T(n) = O(n.logn)

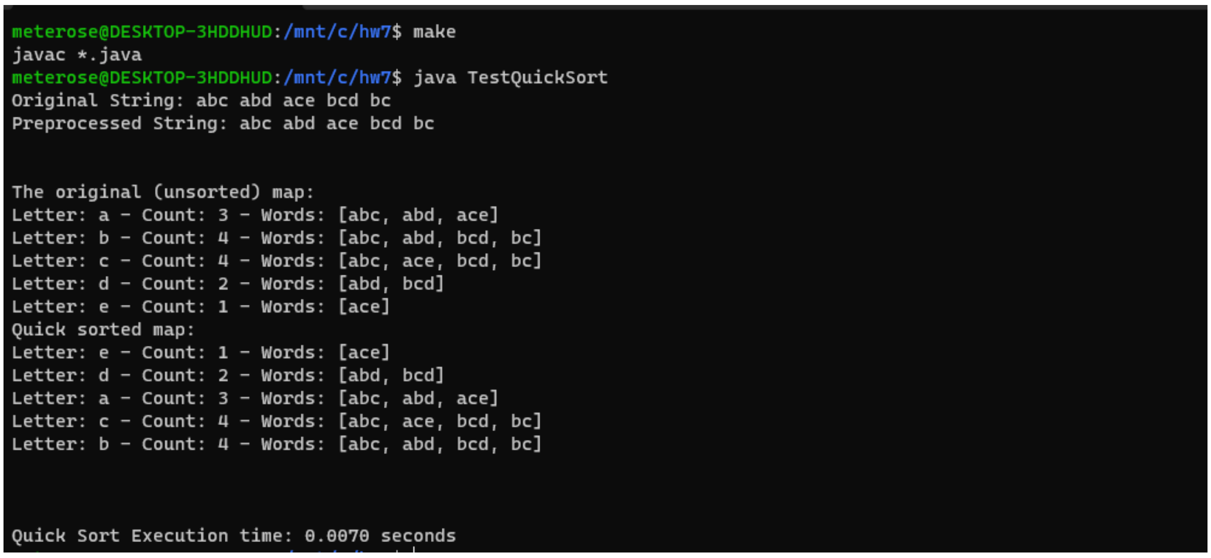
- In average case analysis, It is an obligation to consider all possible permutations of the array. In each iteartion the pivot divides the array into two almost equal halves during each partitioning step. This partitioning ensures that, on average half the elements are proccessed on each side of the pivot. Therefore, the recursion depth becomes logarithmic to the input size.

Running Results for QuickSort for Best-case, Worst-case, and Average-case inputs



```
/* Selection Sort Tests */
String selectionSort_worstCase =   "a ab abc abcd abcde";            /* The worst-case scenario reverse order */
String selectionSort_bestCase =    "edcba dcba cba ba a";           /* The best-case scenario ordered array */
String selectionSort_averageCase = "abc abd ace bcd bc";           /* The average case scenario */
```

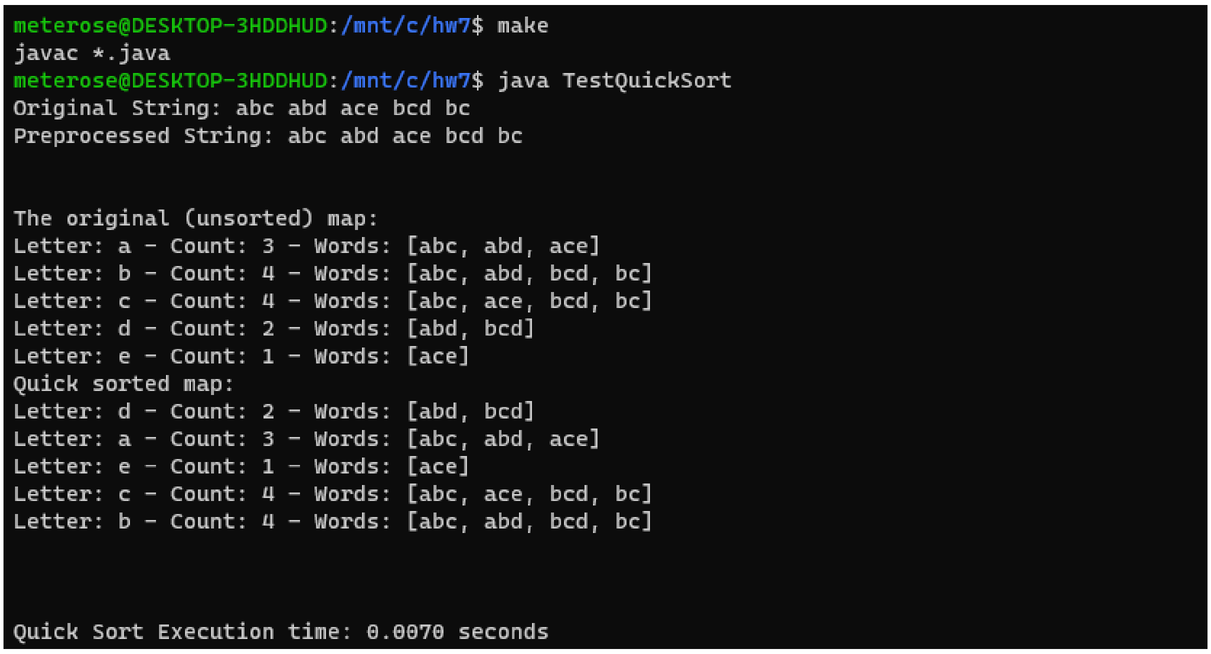
1-) Running time complexity of worst-case



```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ make
javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestQuickSort
Original String: abc abd ace bcd bc
Preprocessed String: abc abd ace bcd bc


The original (unsorted) map:
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: e - Count: 1 - Words: [ace]
Quick sorted map:
Letter: e - Count: 1 - Words: [ace]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]



Quick Sort Execution time: 0.0070 seconds
```

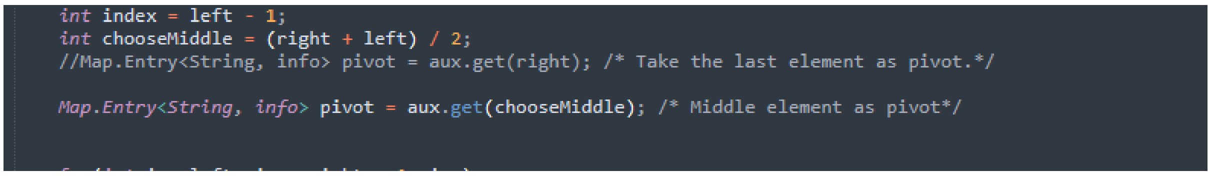
In the worst case scenario, I chose the last element as pivot in each iteration, so that the time complexity become O(n^2) and the running time is 0.0070.

2-) Running time complexity of best-case



```
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ make
javac *.java
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$ java TestQuickSort
Original String: abc abd ace bcd bc
Preprocessed String: abc abd ace bcd bc


The original (unsorted) map:
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: e - Count: 1 - Words: [ace]
Quick sorted map:
Letter: d - Count: 2 - Words: [abd, bcd]
Letter: a - Count: 3 - Words: [abc, abd, ace]
Letter: e - Count: 1 - Words: [ace]
Letter: c - Count: 4 - Words: [abc, ace, bcd, bc]
Letter: b - Count: 4 - Words: [abc, abd, bcd, bc]



Quick Sort Execution time: 0.0070 seconds
```

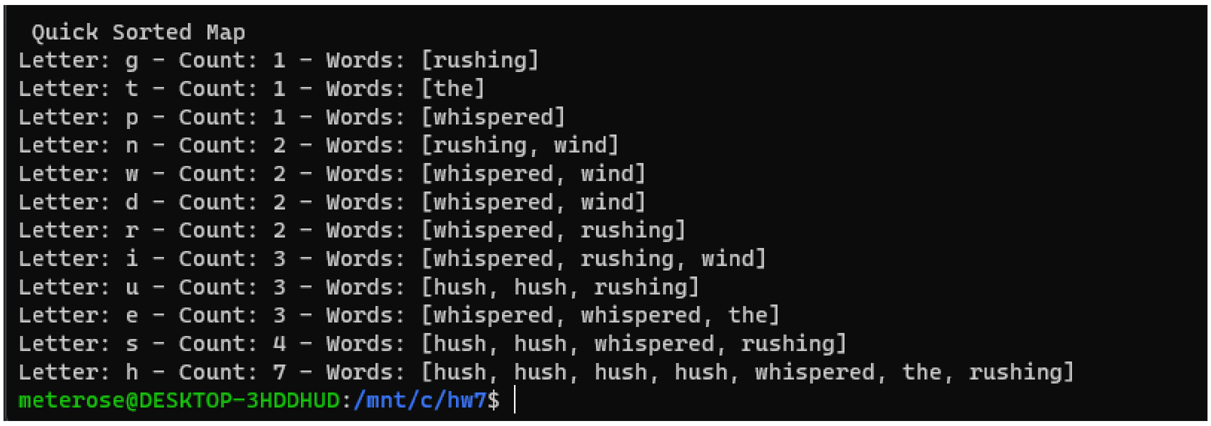
In the best case scenario, I modified the algorithm as follows;



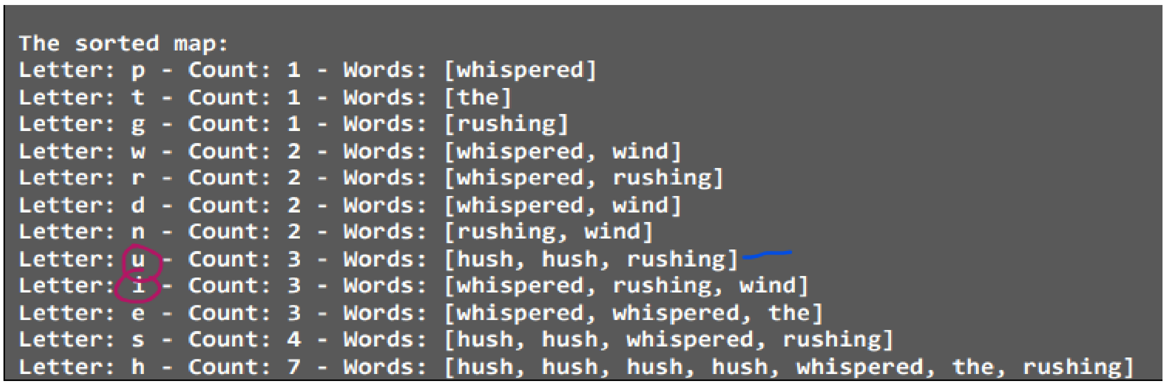
```
    int index = left - 1;
    int chooseMiddle = (right + left) / 2;
    //Map.Entry<String, info> pivot = aux.get(right); /* Take the last element as pivot.*/

    Map.Entry<String, info> pivot = aux.get(chooseMiddle); /* Middle element as pivot*/
```

By doing this modification. In each iteration I chose the middle element as pivot. So that I obtained O(nlogn) time complexity and the running time is 0.0070.

Analyze of whether Quick sort keeps the Input Order (PartD)



```
 Quick Sorted Map
Letter: g - Count: 1 - Words: [rushing]
Letter: t - Count: 1 - Words: [the]
Letter: p - Count: 1 - Words: [whispered]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
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]
meterose@DESKTOP-3HDDHUD:/mnt/c/hw7$
```



```
The sorted map:
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]
```

The QuickSort algorithm **does not keep the input order** as you can see above this is because, relies on on partitioning the array according to to the pivot that has been chosen by partition method. The partitioning method includes the rearranging entries that might change their original order. The entries that are smaller than the pivot will be placed left of the pivot and the entries that are greater than the pivot are going to be place in the right of the pivot. **Therefore, quick sort does not guarantee the input order.**

Summation;

Time complexities for each Sort Algorithm

Big O	**Wost-case**	**Average-case**	**Best-case**
SelectionSort	**O(n^2)**	**O(n^2)**	**O(n^2)**
InsertionSort	**O(n^2)**	**O(n^2)**	**O(n)**
MergeSort	**O(nlogn)**	**O(nlogn)**	**O(nlogn)**
BubbleSort	**O(n^2)**	**O(n^2)**	**O(n)**
QuickSort	**O(n^2)**	**O(nlogn)**	**O(nlogn)**

Time running times for each Sort Algorithm

Running Time	**Wost-case**	**Average-case**	**Best-case**
SelectionSort	**0.0070**	**0.0070**	**0.0070**
InsertionSort	**0.0060**	**0.0060**	**0.0070**
MergeSort	**0.0070**	**0.0080**	**0.0080**
BubbleSort	**0.0070**	**0.0070**	**0.0070**
QuickSort	**0.0070**		**0.0070**