GTU Department of Computer Engineering

CSE 222/505 - Spring 2023

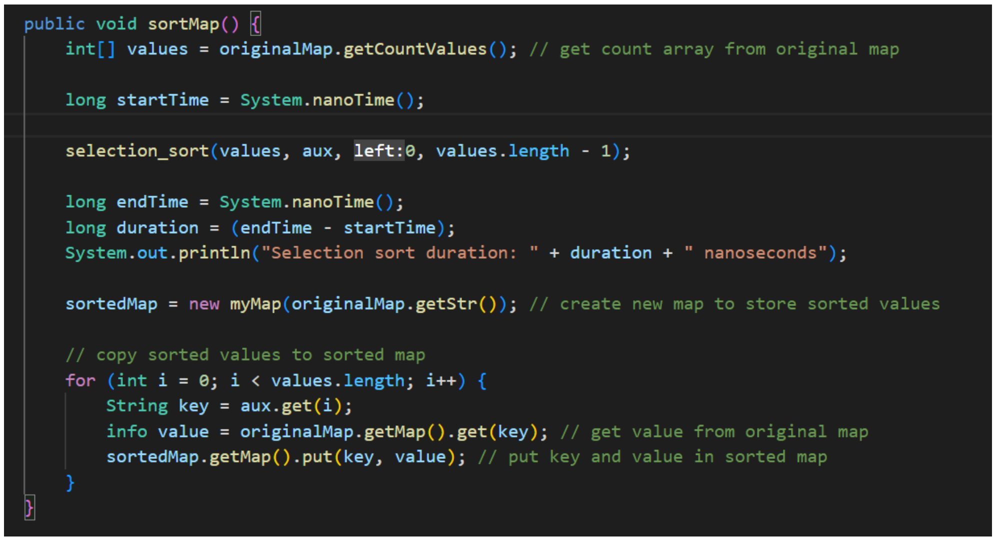
Homework 7 Report

BERRU LAFCI

1901042681

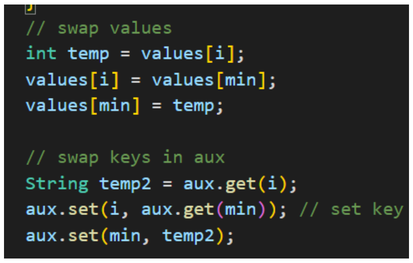
Problem Solution Approach: *(I forgot to add this part in hw6, sorry for that)*

In this homework, as I did in HW6 I used a sortMap() method for all sorts to use in main. I send the values array(counts), aux array(keys), left and right values to sort methods. Also, it calculates the duration and makes the sorted map.



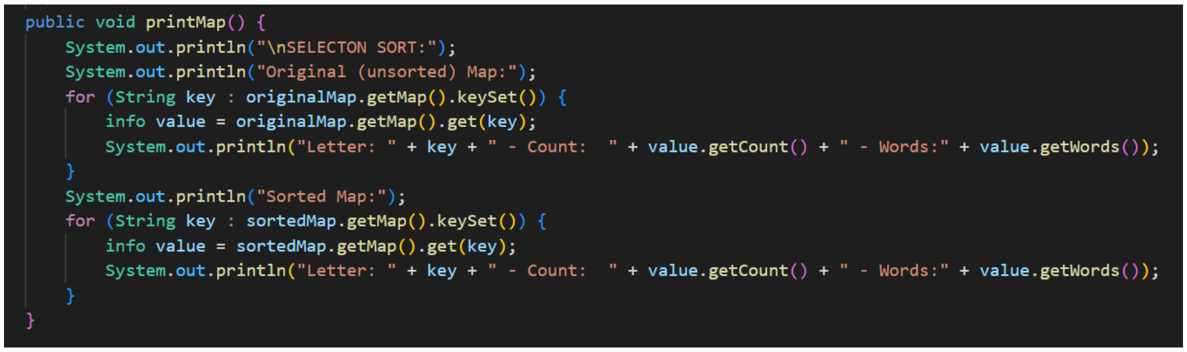
```java
public void sortMap() {
    int[] values = originalMap.getCountValues(); // get count array from original map

    long startTime = System.nanoTime();

    selection_sort(values, aux, left:0, values.length - 1);

    long endTime = System.nanoTime();
    long duration = (endTime - startTime);
    System.out.println("Selection sort duration: " + duration + " nanoseconds");

    sortedMap = new myMap(originalMap.getStr()); // create new map to store sorted values

    // copy sorted values to sorted map
    for (int i = 0; i < values.length; i++) {
        String key = aux.get(i);
        info value = originalMap.getMap().get(key); // get value from original map
        sortedMap.getMap().put(key, value); // put key and value in sorted map
    }
}
```

I used values array to sort easily. So, after swapping values, I also swap keys in sort methods.



```java
// swap values
int temp = values[i];
values[i] = values[min];
values[min] = temp;

// swap keys in aux
String temp2 = aux.get(i);
aux.set(i, aux.get(min)); // set key
aux.set(min, temp2);
```

Also, every method has printMap method which prints the original and the sorted map.



```java
public void printMap() {
    System.out.println("\nSELECTON SORT:");
    System.out.println("Original (unsorted) Map:");
    for (String key : originalMap.getMap().keySet()) {
        info value = originalMap.getMap().get(key);
        System.out.println("Letter: " + key + " - Count:  " + value.getCount() + " - Words:" + value.getWords());
    }
    System.out.println("Sorted Map:");
    for (String key : sortedMap.getMap().keySet()) {
        info value = sortedMap.getMap().get(key);
        System.out.println("Letter: " + key + " - Count:  " + value.getCount() + " - Words:" + value.getWords());
    }
}
```

In merge sort I split the values array and aux array as left and right recursively. Then I compared and sorted them. Then I merged them back in a sorted way.

In selection sort, I tracked the minimum value during each iteration and at the end of each iteration I swapped the minimum value with the value at the current index. It went through the end of arrays.

In insertion sort, I began at index 1 in for loop. I examined elements to the left with a while loop. If any elements are larger than the current element, I shift those elements to the right until the previous element is less than the current element. And this goes until the end of array.

In bubble sort, I compared every pair of adjacent elements until the end of array. Then it goes back to the next index where we start and again, I compare every pair of adjacent elements. I go until the end of array for outer loop.

In quick sort, I dedicated the pivot place with partition method. The partition method rearranges the elements so that values less than the pivot are on the left and values greater than or equal to the pivot are on the right and returns the pivot index. Then I recursively called itself on the left and right partitions, sorting them separately.

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

Merge Sort:

Best-case time complexity:

It occurs when the array is **already sorted**. So, there will still be a split stage to subarrays, but no sorting while merging since it can determine that it is sorted at the merge stage.

Splitting array recursively into halves until individual elements are reached is O(log n). After sorting, merging will be O(n). So, the complexity will be O(n log n).

Shortly, the logarithmic term arises from the fact that the array is halved at each recursive step, and the linear term arises from the merging process.

Average-case time complexity:

It occurs when the array is **randomly shuffled**. So, there will still be a split stage to subarrays, and also sorting in merge stage.

Splitting array recursively into halves until individual elements are reached is O(log n). After sorting, merging will be O(n). So, the complexity will be O(n log n).

Although the actual number of comparisons and movements may vary, on average, the algorithm still performs a similar number of operations as in the best-case scenario.

Worst-case time complexity:

It occurs when the array is **in reverse order or in descending order**. In this case, merge sort needs to perform the maximum number of comparisons and movements at each level of recursion. So, there will still be a split stage to subarrays, and also sorting in merge stage.

Splitting array recursively into halves until individual elements are reached is O(log n). After sorting, merging will be O(n). So, the complexity will be O(n log n).

So, the complexity is still O(n log n) because even though more operations are needed compared to the best-case scenario, each variant of this algorithm has subarray splitting and then merging.

Selection Sort:

Best-case time complexity: already sorted.

It will not make any swap operation, but it will still iterate through the array as nested loop (quadratic number). So the complexity is O(n^2).

Average-case time complexity: randomly shuffled.

It's again O(n^2) because it will iterate through the whole array as nested loop. And also, it will probably make swap but this doesn't affect the complexity.

Worst-case time complexity: reverse order or in descending order.

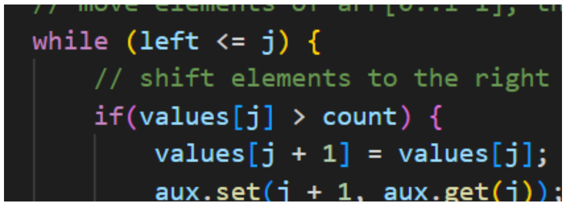
In this case, the algorithm needs to make the maximum number of comparisons and swaps at each pass. The worst-case time complexity is again O(n^2) because of same reasons with best and average.

Insertion Sort:

Best-case time complexity: already sorted.

In this case, each element in the array is compared to its previous elements and found to be already in its correct position, resulting in no swaps. The best-case time complexity is O(n) because the array just needs to iterate start to end to verify the sorted order.

It will not enter this loop:



```
// move elements of arr[0..i-1], th
while (left <= j) {
    // shift elements to the right
    if(values[j] > count) {
        values[j + 1] = values[j];
        aux.set(j + 1, aux.get(j));
```

Average-case time complexity: randomly shuffled.

On average, a couple of elements will shift those on its left to the right until the smallest one (quadratic number). So, the complexity is O(n^2). Although some elements may require fewer comparisons and some may require more, on average, the algorithm performs a quadratic number of comparisons and movements.

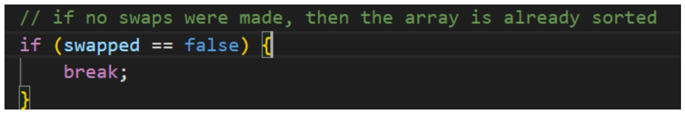
Worst-case time complexity: reverse order or in descending order.

In this case, each element needs to be compared with all the previously sorted elements, resulting in the maximum number of comparisons and shifts. The complexity is O(n^2). This occurs because of same reason as the average case but for all elements.

Bubble Sort:

Best-case time complexity: already sorted.

In this case, Bubble Sort only needs to make a single pass through the array to confirm that it is sorted, without any swaps. With the **swapped** variable, it will detect that swap is done or not. If not, we understand that the array is sorted, and it will break and exit from the outer loop.



```
// if no swaps were made, then the array is already sorted
if (swapped == false) {
    break;
}
```

So, the best-case time complexity is O(n).

Average-case time complexity: randomly shuffled.

As opposed to above, if the swapped variable occurs true, it will iterate through the array as nested loop (quadratic number). So, the complexity will be O(n^2).

Worst-case time complexity: reverse order or in descending order.

The complexity will be O(n^2) with the same reason as the average case but for all elements.

Quick Sort

Best-case time complexity: sorted

The best-case scenario occurs when the selected pivot divides the array into two equal-sized subarrays in each recursive call. This leads to balanced partitioning, resulting in efficient sorting. In the best-case scenario time complexity is O(n log n). This is because the array is divided into log n levels, and at each level, all 'n' elements are processed.

Average-case time complexity: randomly shuffled

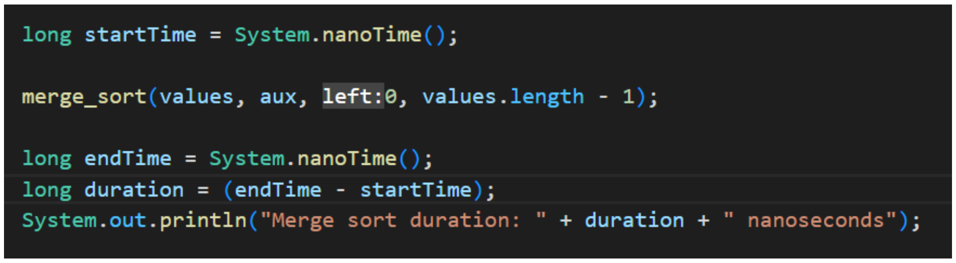
The average-case scenario occurs when the array is randomly shuffled. On average, Quick Sort achieves a time complexity of O(n log n). The pivot selection and partitioning process distribute the elements relatively evenly across the subarrays, leading to balanced recursive calls.

Worst-case time complexity: descending

The worst-case scenario for Quick Sort occurs when the selected pivot is either the smallest or largest element in the array in every recursive call. In the worst-case scenario, Quick Sort degrades to a time complexity of O(n^2).

b) Running time of each sorting algorithm for each input. You can add a method to each sorting algorithm's class to measure the time.

I calculated running times in their classes like this:



```java
long startTime = System.nanoTime();

merge_sort(values, aux, left:0, values.length - 1);

long endTime = System.nanoTime();
long duration = (endTime - startTime);
System.out.println("Merge sort duration: " + duration + " nanoseconds");
```

And I used these inputs for testing best, average and worst:

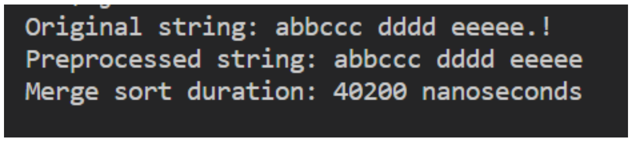
abbccc dddd eeeee.! ***(1,2,3,4,5) (sorted)***

aaa bbbbbb cc dddddddddd eee.! ***(3,6,2,10,3) (randomly)***

aaaaaa bbbbb cccc dd e.! ***(6,5,4,2,1) (descending)***

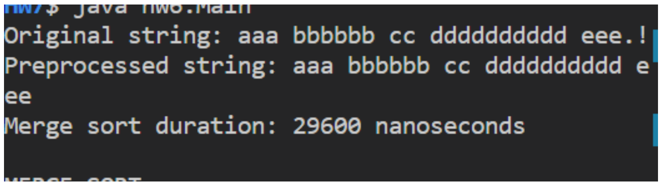
Merge sort:

Best: O(n log n)



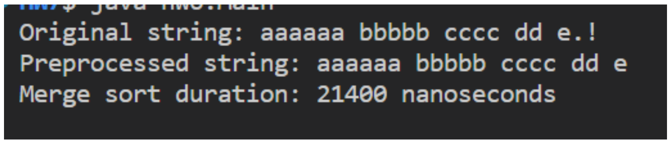
```
Original string: abbccc dddd eeeee.!
Preprocessed string: abbccc dddd eeeee
Merge sort duration: 40200 nanoseconds
```

Average: O(n log n)



```
Original string: aaa bbbbbb cc dddddddddd eee.!
Preprocessed string: aaa bbbbbb cc dddddddddd e
ee
Merge sort duration: 29600 nanoseconds
```

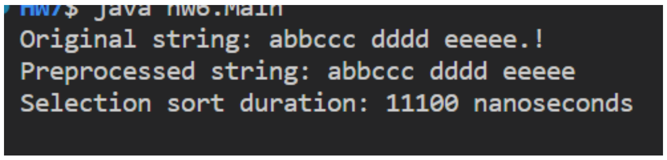
Worst: O(n log n)



```
Original string: aaaaaa bbbbb cccc dd e.!
Preprocessed string: aaaaaa bbbbb cccc dd e
Merge sort duration: 21400 nanoseconds
```

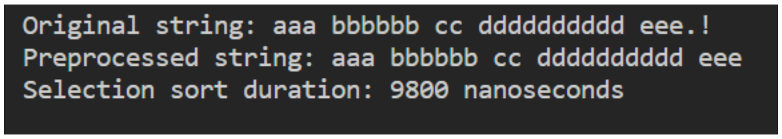
Selection Sort:

Best: O(n^2)



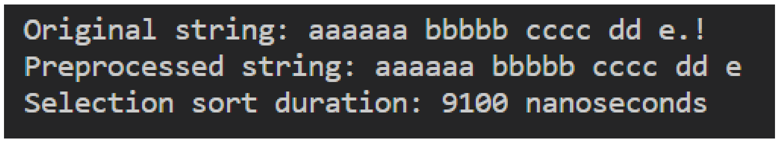
```
HW7$ java hw6.Main
Original string: abbccc dddd eeeee.!
Preprocessed string: abbccc dddd eeeee
Selection sort duration: 11100 nanoseconds
```

Average: O(n^2).



```
Original string: aaa bbbbbb cc dddddddddd eee.!
Preprocessed string: aaa bbbbbb cc dddddddddd eee
Selection sort duration: 9800 nanoseconds
```

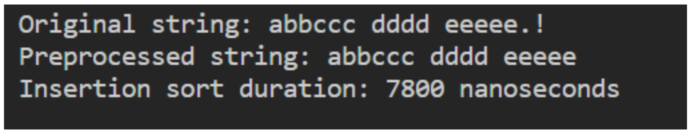
Worst: O(n^2).



```
Original string: aaaaaa bbbbb cccc dd e.!
Preprocessed string: aaaaaa bbbbb cccc dd e
Selection sort duration: 9100 nanoseconds
```

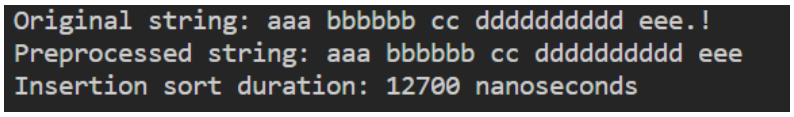
Insertion Sort:

Best: O(n)



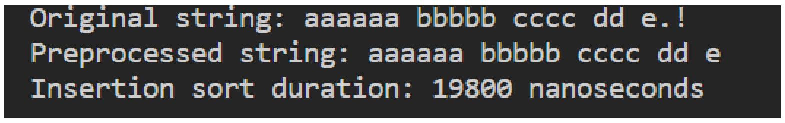
```
Original string: abbccc dddd eeeee.!
Preprocessed string: abbccc dddd eeeee
Insertion sort duration: 7800 nanoseconds
```

Average: O(n^2)



```
Original string: aaa bbbbbb cc dddddddddd eee.!
Preprocessed string: aaa bbbbbb cc dddddddddd eee
Insertion sort duration: 12700 nanoseconds
```

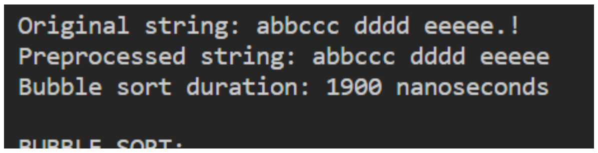
Worst: O(n^2)



```
Original string: aaaaaa bbbbb cccc dd e.!
Preprocessed string: aaaaaa bbbbb cccc dd e
Insertion sort duration: 19800 nanoseconds
```

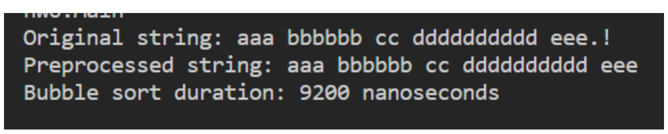
Bubble Sort:

Best: O(n)



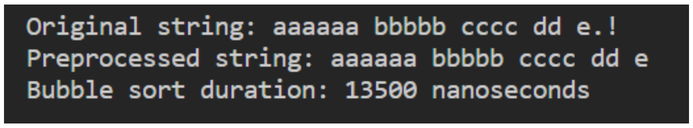
```
Original string: abbccc dddd eeeee.!
Preprocessed string: abbccc dddd eeeee
Bubble sort duration: 1900 nanoseconds

BUBBLE SORT:
```

Average: O(n^2)



```
Original string: aaa bbbbbb cc dddddddddd eee.!
Preprocessed string: aaa bbbbbb cc dddddddddd eee
Bubble sort duration: 9200 nanoseconds
```

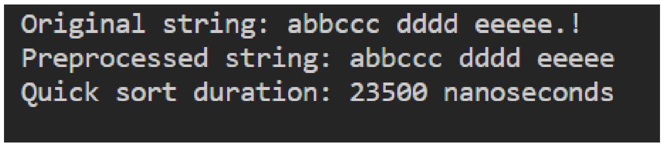
Worst: O(n^2)



```
Original string: aaaaaa bbbbb cccc dd e.!
Preprocessed string: aaaaaa bbbbb cccc dd e
Bubble sort duration: 13500 nanoseconds
```

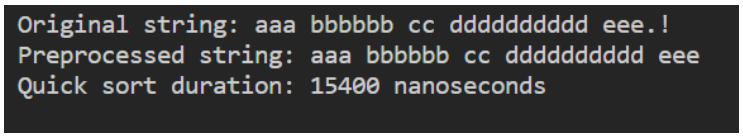
Quick Sort:

Best: O(n log n)



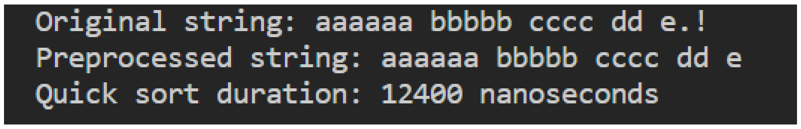
```
Original string: abbccc dddd eeeee.!
Preprocessed string: abbccc dddd eeeee
Quick sort duration: 23500 nanoseconds
```

Average: O(n log n)



```
Original string: aaa bbbbbb cc dddddddddd eee.!
Preprocessed string: aaa bbbbbb cc dddddddddd eee
Quick sort duration: 15400 nanoseconds
```

Worst: O(n^2)



```
Original string: aaaaaa bbbbb cccc dd e.!
Preprocessed string: aaaaaa bbbbb cccc dd e
Quick sort duration: 12400 nanoseconds
```

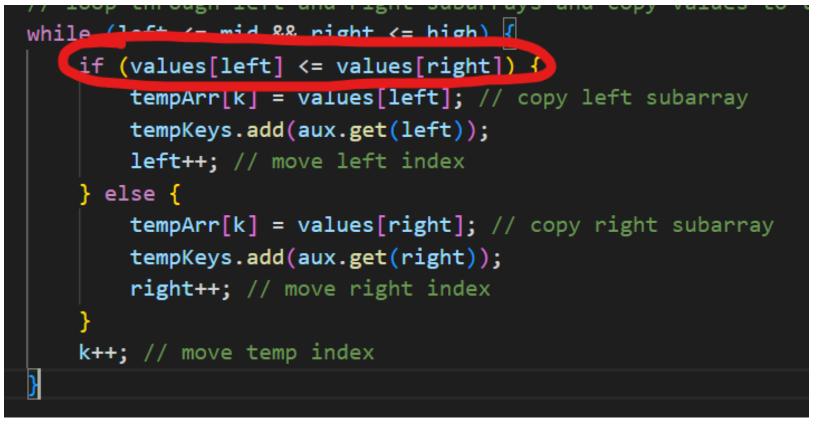
c) Comparison of the sorting algorithms (by using the information from Part2-a and Part2- b). Which algorithm is faster in which case?

If we look at runtimes, Insertion and bubble are faster in best case. In general, selection sort is the fastest for all cases among all sorts.

If we look at complexities, we can observe that both Merge Sort and Quick Sort perform well in the average and best cases, with a time complexity of O(n log n). However, in the worst case, Quick Sort can degrade to O(n^2) time complexity. Also, insertion and bubble are O(n) in best case which is fastest among them.

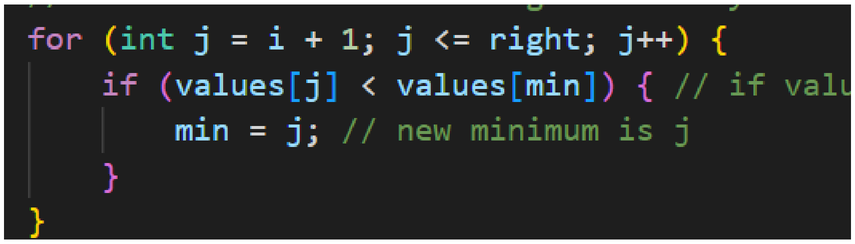
d) In HW6, it was highlighted that the ordering of the letters with same count value should be the same as their addition order to myMap object. However, for these 4 sorting algorithms, the case might not be the same. You are expected to analyze which algorithms keep the input ordering and which don't, along with the code snippet that causes/ensures this.

Merge: Because of <= in marked line, when they are equal it gives priority to previous one. So, it occurs in straight order.



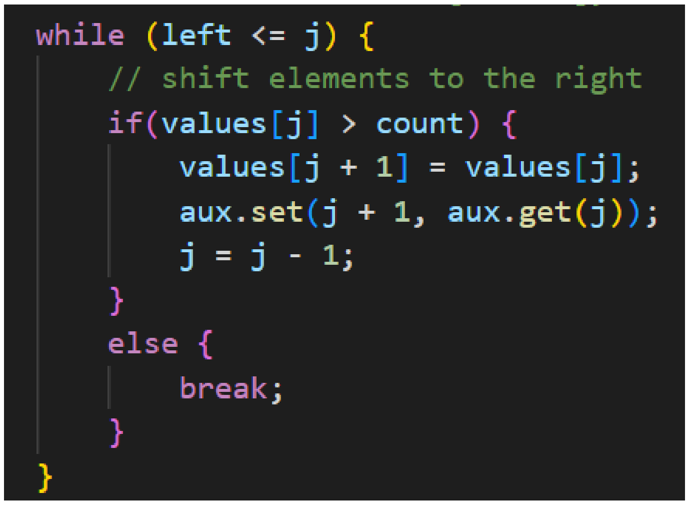
```
// loop through left and right subarrays and copy values to
while (left <= mid && right <= high) {
    if (values[left] <= values[right]) {
        tempArr[k] = values[left]; // copy left subarray
        tempKeys.add(aux.get(left));
        left++; // move left index
    } else {
        tempArr[k] = values[right]; // copy right subarray
        tempKeys.add(aux.get(right));
        right++; // move right index
    }
    k++; // move temp index
}
```

Selection: Since we track just the minimum values, there is no need to change into <=. So, it occurs in straight order.



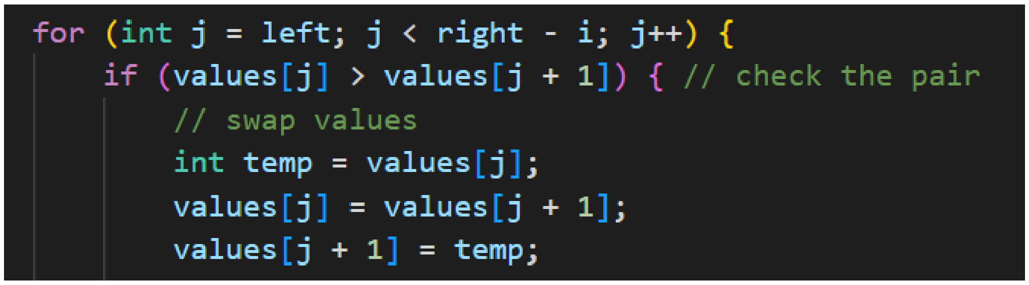
```
for (int j = i + 1; j <= right; j++) {
    if (values[j] < values[min]) { // if valu
        min = j; // new minimum is j
    }
}
```

Insertion: Since they are shifting when smaller values are found, left ones stay on left. So, it occurs in straight order.



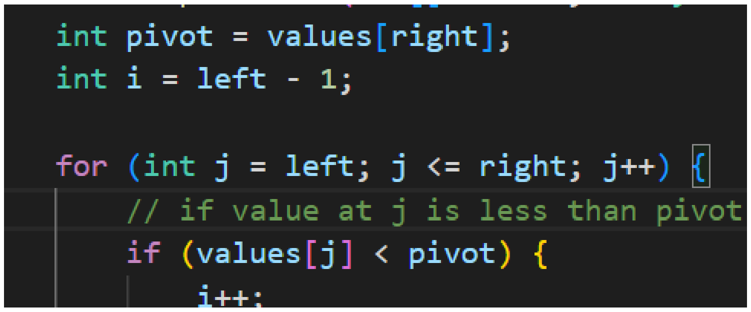
```
while (left <= j) {
    // shift elements to the right
    if(values[j] > count) {
        values[j + 1] = values[j];
        aux.set(j + 1, aux.get(j));
        j = j - 1;
    }
    else {
        break;
    }
}
```

Bubble: Because it only swaps when it is bigger, the order does not change. So, it occurs in straight order.



```
for (int j = left; j < right - i; j++) {
    if (values[j] > values[j + 1]) { // check the pair
        // swap values
        int temp = values[j];
        values[j] = values[j + 1];
        values[j + 1] = temp;
```

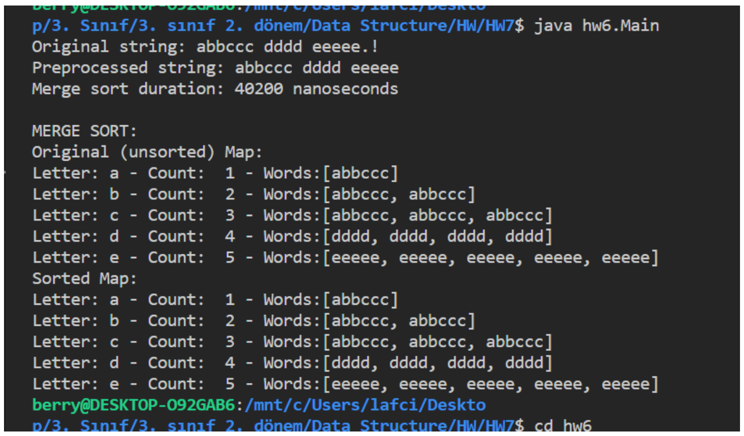
Quick: Since I declared the pivot with last element, it cannot protect the order. But if I declare the pivot in another place, it can be in straight order. But now for my algorithm, the order is not straight. (Example in test case for average case)



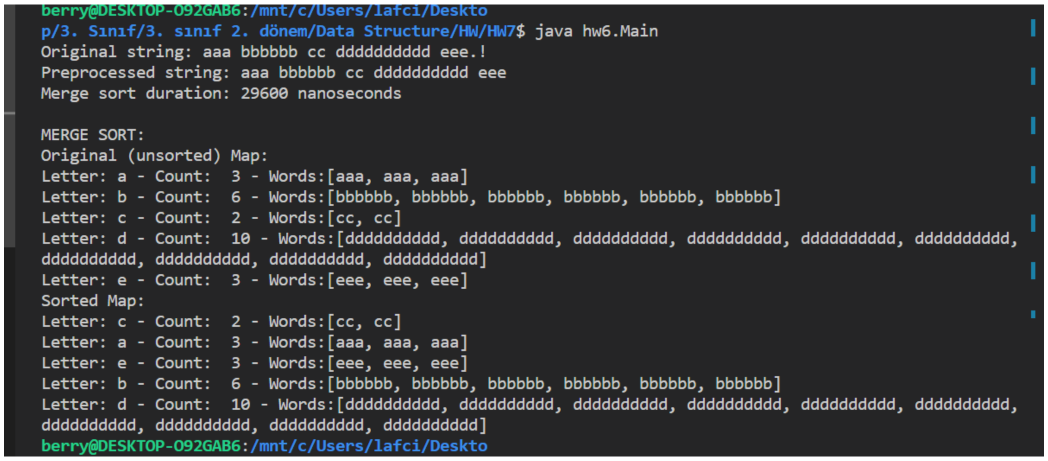
```
  int pivot = values[right];
  int i = left - 1;

  for (int j = left; j <= right; j++) {
      // if value at j is less than pivot
      if (values[j] < pivot) {
          i++;
```

TEST CASES:

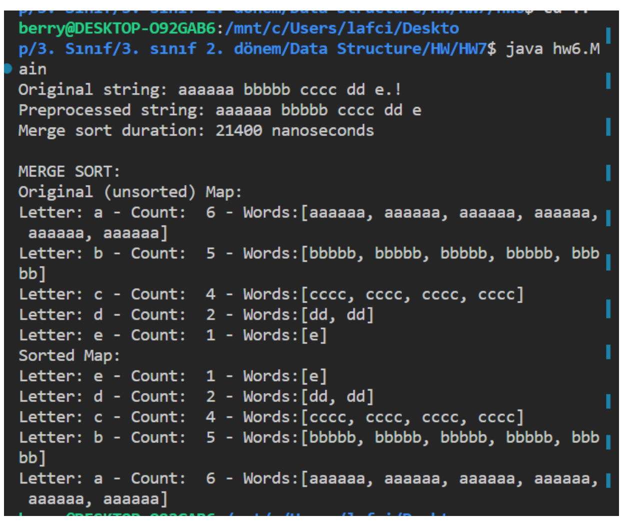
Merge sort:



```
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Deskto
p/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ java hw6.Main
Original string: abbccc dddd eeeee.!
Preprocessed string: abbccc dddd eeeee
Merge sort duration: 40200 nanoseconds

MERGE SORT:
Original (unsorted) Map:
Letter: a - Count:  1 - Words:[abbccc]
Letter: b - Count:  2 - Words:[abbccc, abbccc]
Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
Sorted Map:
Letter: a - Count:  1 - Words:[abbccc]
Letter: b - Count:  2 - Words:[abbccc, abbccc]
Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Deskto
p/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ cd hw6
```

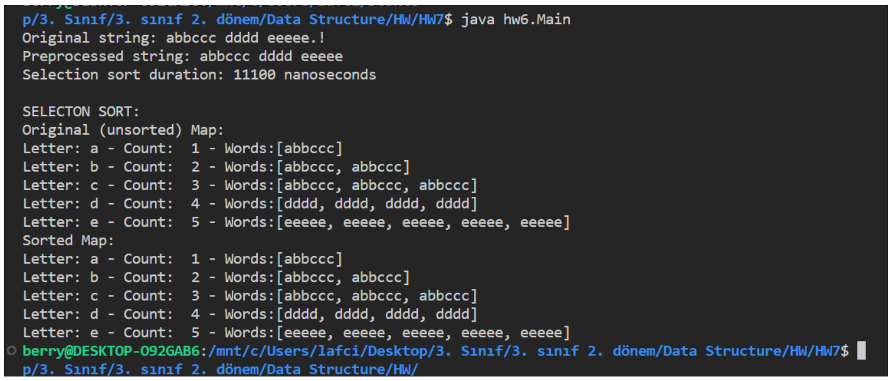


```
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Deskto
p/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ java hw6.Main
Original string: aaa bbbbbb cc dddddddddd eee.!
Preprocessed string: aaa bbbbbb cc dddddddddd eee
Merge sort duration: 29600 nanoseconds

MERGE SORT:
Original (unsorted) Map:
Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
Letter: c - Count:  2 - Words:[cc, cc]
Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
dddddddddd, dddddddddd, dddddddddd, dddddddddd]
Letter: e - Count:  3 - Words:[eee, eee, eee]
Sorted Map:
Letter: c - Count:  2 - Words:[cc, cc]
Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
Letter: e - Count:  3 - Words:[eee, eee, eee]
Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
dddddddddd, dddddddddd, dddddddddd, dddddddddd]
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Deskto
```

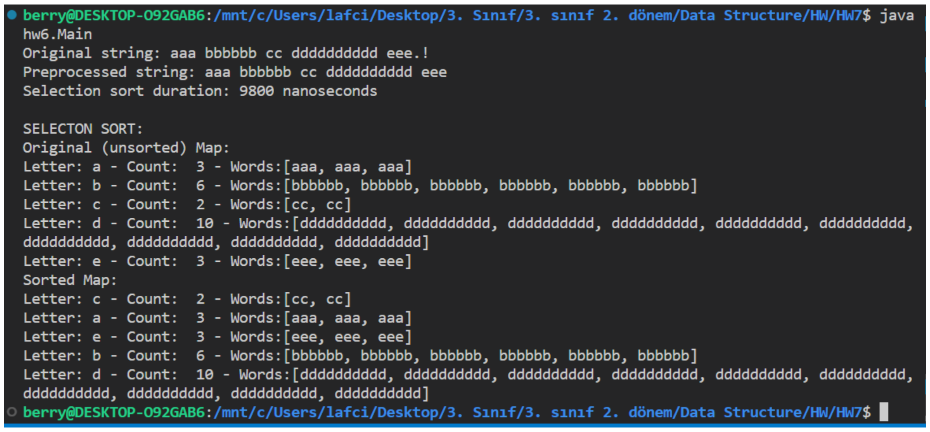


```
p/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ cd ..
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Deskto
p/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ java hw6.M
ain
Original string: aaaaaa bbbbb cccc dd e.!
Preprocessed string: aaaaaa bbbbb cccc dd e
Merge sort duration: 21400 nanoseconds

MERGE SORT:
Original (unsorted) Map:
Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa,
 aaaaaa, aaaaaa]
Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbb
bb]
Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
Letter: d - Count:  2 - Words:[dd, dd]
Letter: e - Count:  1 - Words:[e]
Sorted Map:
Letter: e - Count:  1 - Words:[e]
Letter: d - Count:  2 - Words:[dd, dd]
Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbb
bb]
Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa,
 aaaaaa, aaaaaa]
```

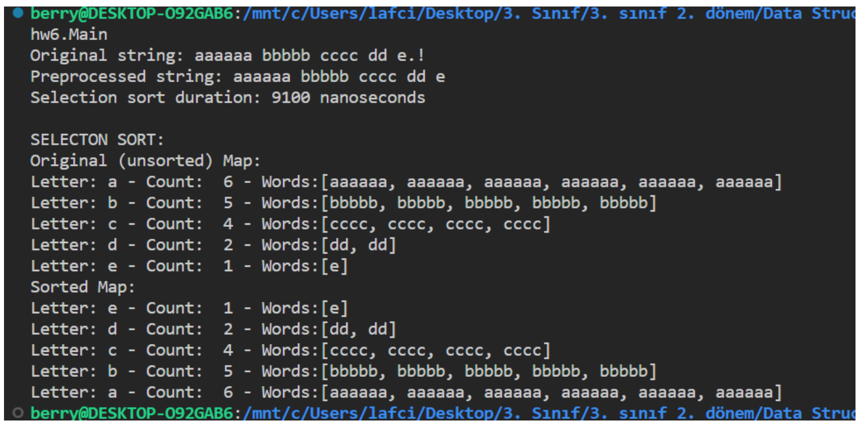
Selection Sort:



```
p/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ java hw6.Main
Original string: abbccc dddd eeeee.!
Preprocessed string: abbccc dddd eeeee
Selection sort duration: 11100 nanoseconds

SELECTON SORT:
Original (unsorted) Map:
Letter: a - Count:  1 - Words:[abbccc]
Letter: b - Count:  2 - Words:[abbccc, abbccc]
Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
Sorted Map:
Letter: a - Count:  1 - Words:[abbccc]
Letter: b - Count:  2 - Words:[abbccc, abbccc]
Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$
p/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/
```

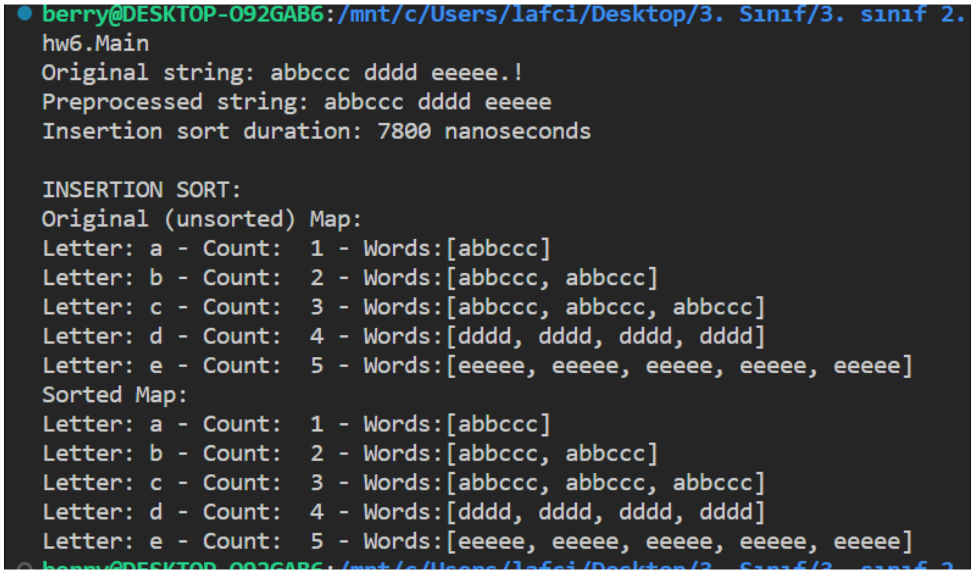


```
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ java
hw6.Main
Original string: aaa bbbbbb cc dddddddddd eee.!
Preprocessed string: aaa bbbbbb cc dddddddddd eee
Selection sort duration: 9800 nanoseconds

SELECTON SORT:
Original (unsorted) Map:
Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
Letter: c - Count:  2 - Words:[cc, cc]
Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
dddddddddd, dddddddddd, dddddddddd, dddddddddd]
Letter: e - Count:  3 - Words:[eee, eee, eee]
Sorted Map:
Letter: c - Count:  2 - Words:[cc, cc]
Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
Letter: e - Count:  3 - Words:[eee, eee, eee]
Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
dddddddddd, dddddddddd, dddddddddd, dddddddddd]
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$
```

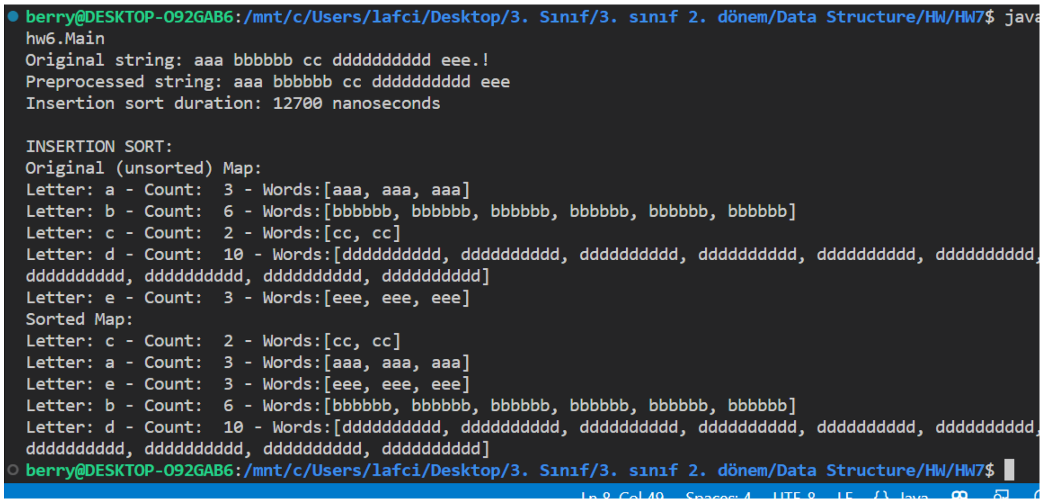


```
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Struc
hw6.Main
Original string: aaaaaa bbbbb cccc dd e.!
Preprocessed string: aaaaaa bbbbb cccc dd e
Selection sort duration: 9100 nanoseconds

SELECTON SORT:
Original (unsorted) Map:
Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa]
Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbbbb]
Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
Letter: d - Count:  2 - Words:[dd, dd]
Letter: e - Count:  1 - Words:[e]
Sorted Map:
Letter: e - Count:  1 - Words:[e]
Letter: d - Count:  2 - Words:[dd, dd]
Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbbbb]
Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa]
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Struc
```

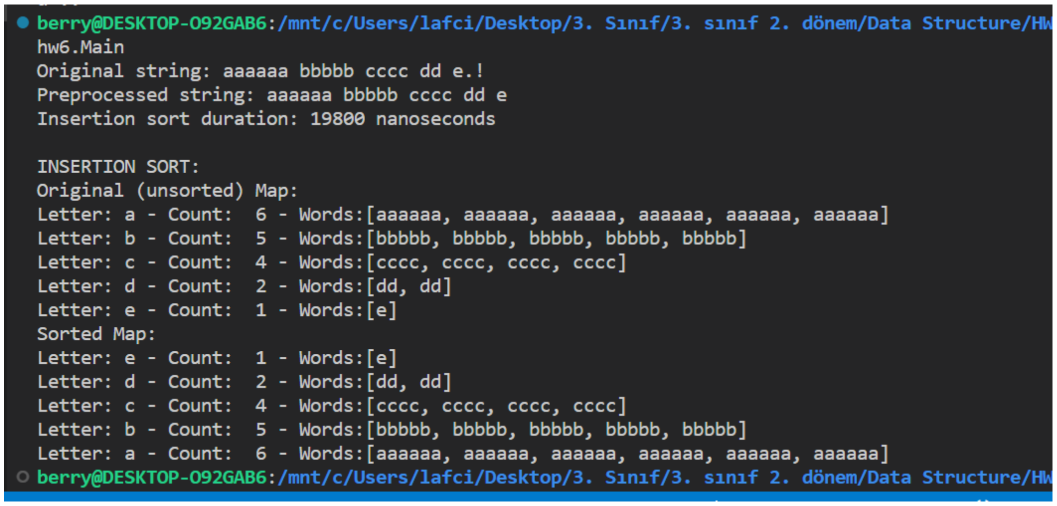
Insertion Sort:



```
● berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2.
  hw6.Main
  Original string: abbccc dddd eeeee.!
  Preprocessed string: abbccc dddd eeeee
  Insertion sort duration: 7800 nanoseconds

  INSERTION SORT:
  Original (unsorted) Map:
  Letter: a - Count:  1 - Words:[abbccc]
  Letter: b - Count:  2 - Words:[abbccc, abbccc]
  Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
  Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
  Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
  Sorted Map:
  Letter: a - Count:  1 - Words:[abbccc]
  Letter: b - Count:  2 - Words:[abbccc, abbccc]
  Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
  Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
  Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
```

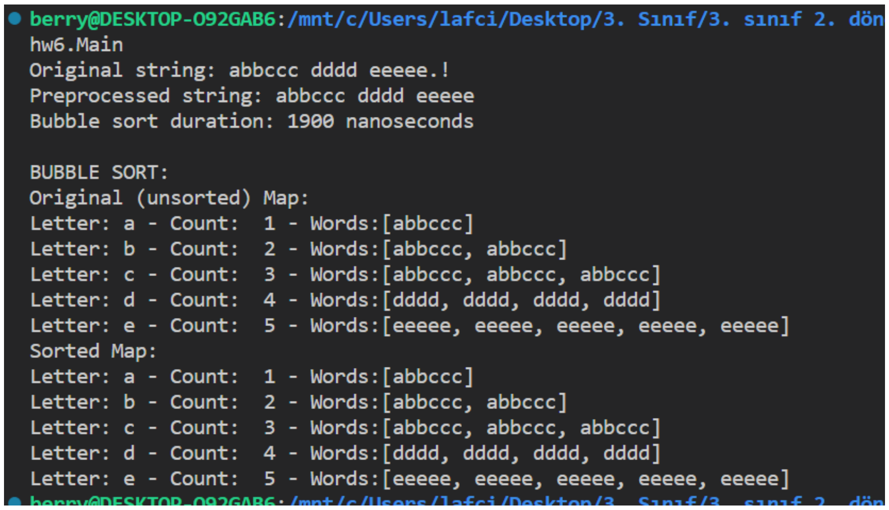


```
● berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ java
  hw6.Main
  Original string: aaa bbbbbb cc dddddddddd eee.!
  Preprocessed string: aaa bbbbbb cc dddddddddd eee
  Insertion sort duration: 12700 nanoseconds

  INSERTION SORT:
  Original (unsorted) Map:
  Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
  Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
  Letter: c - Count:  2 - Words:[cc, cc]
  Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
  dddddddddd, dddddddddd, dddddddddd, dddddddddd]
  Letter: e - Count:  3 - Words:[eee, eee, eee]
  Sorted Map:
  Letter: c - Count:  2 - Words:[cc, cc]
  Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
  Letter: e - Count:  3 - Words:[eee, eee, eee]
  Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
  Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
  dddddddddd, dddddddddd, dddddddddd, dddddddddd]
○ berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ █
```

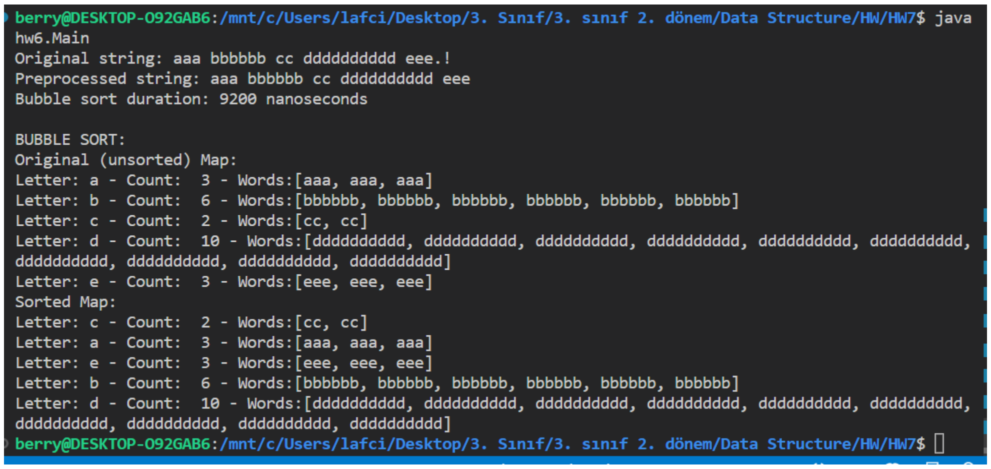


```
● berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW
  hw6.Main
  Original string: aaaaaa bbbbb cccc dd e.!
  Preprocessed string: aaaaaa bbbbb cccc dd e
  Insertion sort duration: 19800 nanoseconds

  INSERTION SORT:
  Original (unsorted) Map:
  Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa]
  Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbbbb]
  Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
  Letter: d - Count:  2 - Words:[dd, dd]
  Letter: e - Count:  1 - Words:[e]
  Sorted Map:
  Letter: e - Count:  1 - Words:[e]
  Letter: d - Count:  2 - Words:[dd, dd]
  Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
  Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbbbb]
  Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa]
○ berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW
```

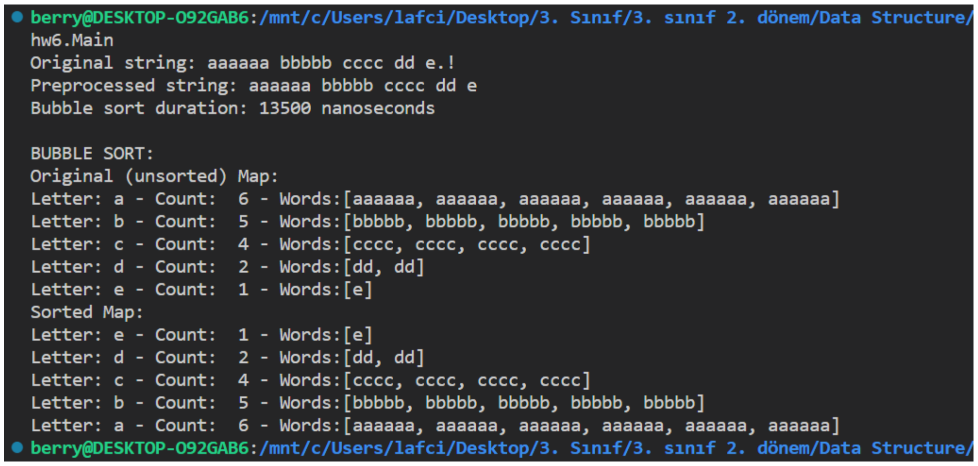
Bubble Sort:



```
● berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dön
  hw6.Main
  Original string: abbccc dddd eeeee.!
  Preprocessed string: abbccc dddd eeeee
  Bubble sort duration: 1900 nanoseconds

  BUBBLE SORT:
  Original (unsorted) Map:
  Letter: a - Count:  1 - Words:[abbccc]
  Letter: b - Count:  2 - Words:[abbccc, abbccc]
  Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
  Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
  Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
  Sorted Map:
  Letter: a - Count:  1 - Words:[abbccc]
  Letter: b - Count:  2 - Words:[abbccc, abbccc]
  Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
  Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
  Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
● berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dön
```

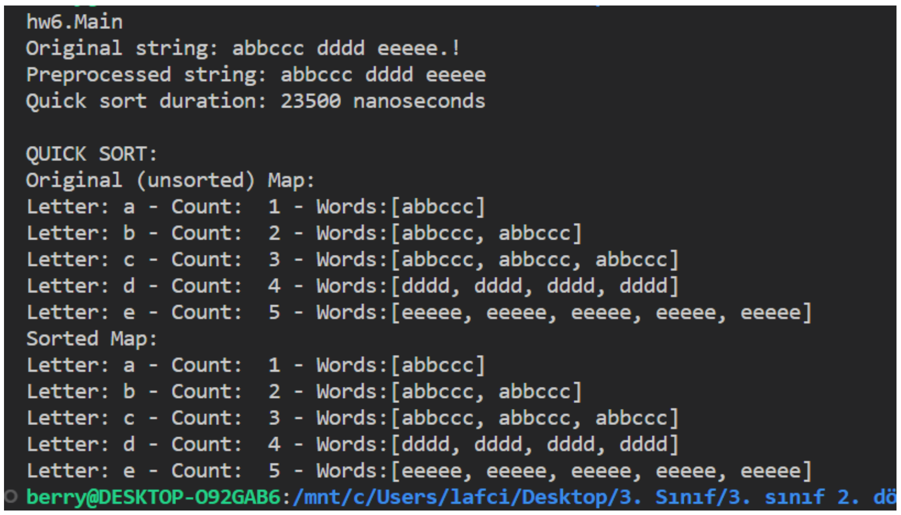


```
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ java
hw6.Main
Original string: aaa bbbbbb cc dddddddddd eee.!
Preprocessed string: aaa bbbbbb cc dddddddddd eee
Bubble sort duration: 9200 nanoseconds

BUBBLE SORT:
Original (unsorted) Map:
Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
Letter: c - Count:  2 - Words:[cc, cc]
Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
dddddddddd, dddddddddd, dddddddddd, dddddddddd]
Letter: e - Count:  3 - Words:[eee, eee, eee]
Sorted Map:
Letter: c - Count:  2 - Words:[cc, cc]
Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
Letter: e - Count:  3 - Words:[eee, eee, eee]
Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
dddddddddd, dddddddddd, dddddddddd, dddddddddd]
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$
```

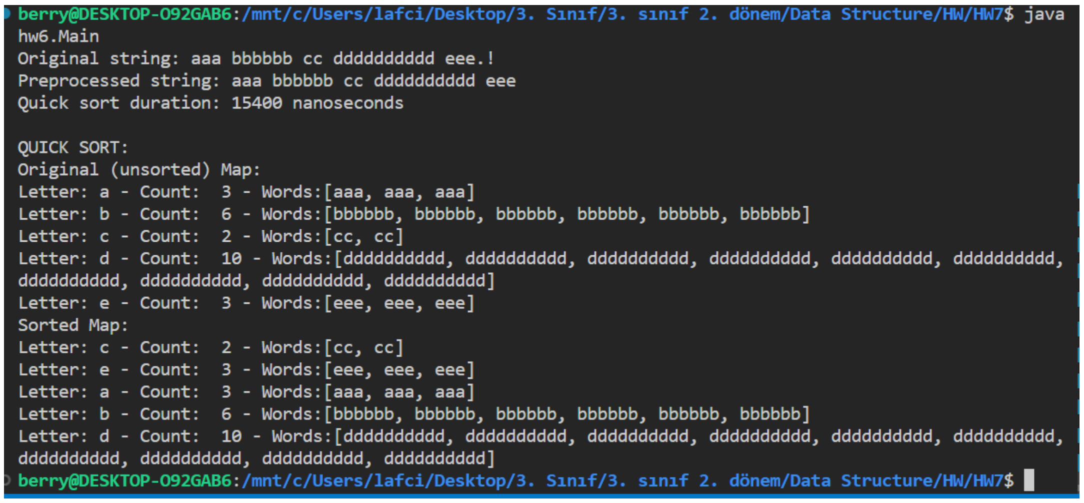


```
● berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/
  hw6.Main
  Original string: aaaaaa bbbbb cccc dd e.!
  Preprocessed string: aaaaaa bbbbb cccc dd e
  Bubble sort duration: 13500 nanoseconds

  BUBBLE SORT:
  Original (unsorted) Map:
  Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa]
  Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbbbb]
  Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
  Letter: d - Count:  2 - Words:[dd, dd]
  Letter: e - Count:  1 - Words:[e]
  Sorted Map:
  Letter: e - Count:  1 - Words:[e]
  Letter: d - Count:  2 - Words:[dd, dd]
  Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
  Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbbbb]
  Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa]
● berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/
```

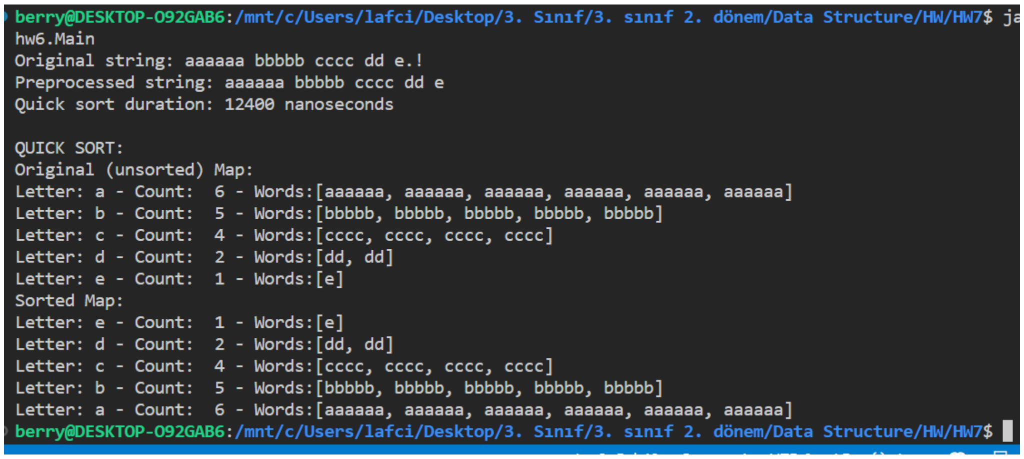
Quick Sort:



```
hw6.Main
Original string: abbccc dddd eeeee.!
Preprocessed string: abbccc dddd eeeee
Quick sort duration: 23500 nanoseconds

QUICK SORT:
Original (unsorted) Map:
Letter: a - Count:  1 - Words:[abbccc]
Letter: b - Count:  2 - Words:[abbccc, abbccc]
Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
Sorted Map:
Letter: a - Count:  1 - Words:[abbccc]
Letter: b - Count:  2 - Words:[abbccc, abbccc]
Letter: c - Count:  3 - Words:[abbccc, abbccc, abbccc]
Letter: d - Count:  4 - Words:[dddd, dddd, dddd, dddd]
Letter: e - Count:  5 - Words:[eeeee, eeeee, eeeee, eeeee, eeeee]
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dö
```

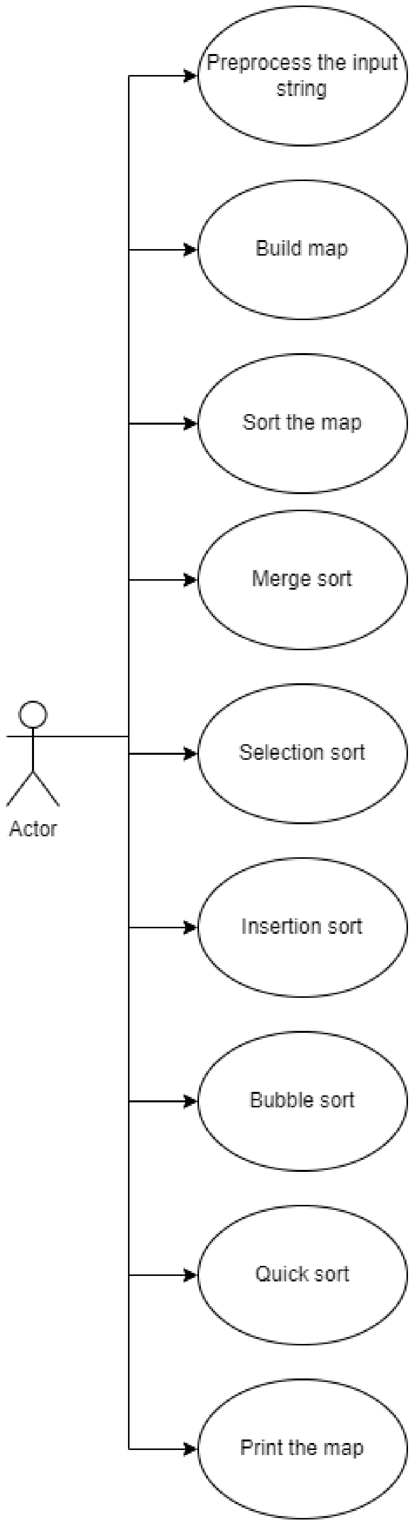


```
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ java
hw6.Main
Original string: aaa bbbbbb cc dddddddddd eee.!
Preprocessed string: aaa bbbbbb cc dddddddddd eee
Quick sort duration: 15400 nanoseconds

QUICK SORT:
Original (unsorted) Map:
Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
Letter: c - Count:  2 - Words:[cc, cc]
Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
dddddddddd, dddddddddd, dddddddddd, dddddddddd]
Letter: e - Count:  3 - Words:[eee, eee, eee]
Sorted Map:
Letter: c - Count:  2 - Words:[cc, cc]
Letter: e - Count:  3 - Words:[eee, eee, eee]
Letter: a - Count:  3 - Words:[aaa, aaa, aaa]
Letter: b - Count:  6 - Words:[bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb, bbbbbb]
Letter: d - Count:  10 - Words:[dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd, dddddddddd,
dddddddddd, dddddddddd, dddddddddd, dddddddddd]
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$
```



```
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$ ja
hw6.Main
Original string: aaaaaa bbbbb cccc dd e.!
Preprocessed string: aaaaaa bbbbb cccc dd e
Quick sort duration: 12400 nanoseconds

QUICK SORT:
Original (unsorted) Map:
Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa]
Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbbbb]
Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
Letter: d - Count:  2 - Words:[dd, dd]
Letter: e - Count:  1 - Words:[e]
Sorted Map:
Letter: e - Count:  1 - Words:[e]
Letter: d - Count:  2 - Words:[dd, dd]
Letter: c - Count:  4 - Words:[cccc, cccc, cccc, cccc]
Letter: b - Count:  5 - Words:[bbbbb, bbbbb, bbbbb, bbbbb, bbbbb]
Letter: a - Count:  6 - Words:[aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa, aaaaaa]
berry@DESKTOP-092GAB6:/mnt/c/Users/lafci/Desktop/3. Sınıf/3. sınıf 2. dönem/Data Structure/HW/HW7$
```

USE CASE


Preprocess the input string
Build map
Sort the map
Merge sort
Selection sort
Insertion sort
Bubble sort
Quick sort
Print the map
Actor