GTU DEPARTMENT of COMPUTER ENGINEERING

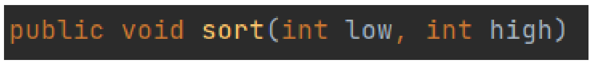
CSE222/505 – SPRING 2023

HOMEWORK 7 REPORT

ÖMER FARUK ÇOLAKEL

200104004043

1. SYSTEM REQUIREMENTS

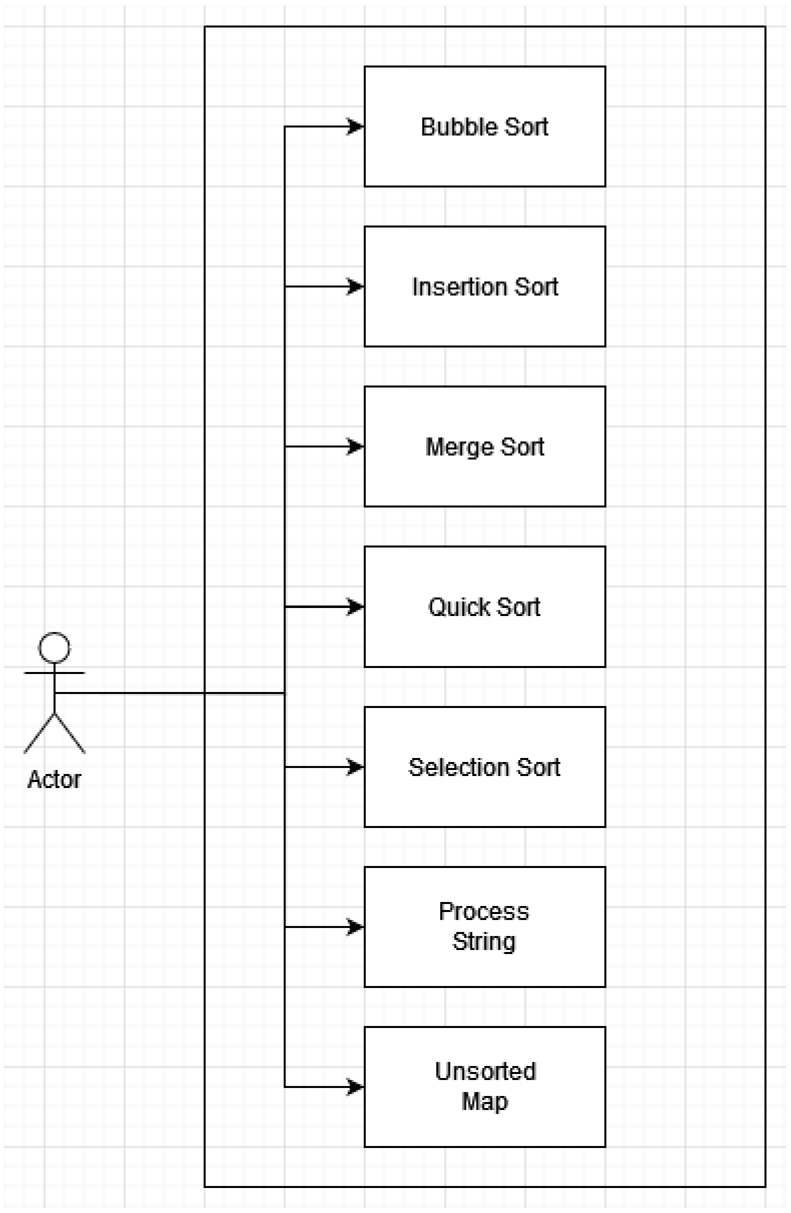


```
public void sort(int low, int high)
```

This method is in quickSort class. Low is the lowest index and high is the highest index of the subarray.

I didn’t include the methods that didn’t require anything or the ones that I already explained in the previous report.

2. USE CASE DIAGRAM



3. PROBLEM SOLUTION APPROACH

I already explained pre-processing input string, mapping and mergeSort in the previous report. So, I didn't include them in this one.

In selection sort, we find the minimum value in the array and swap it with the first element. Then we find the second smallest value and swap it with the second element. So and so forth.

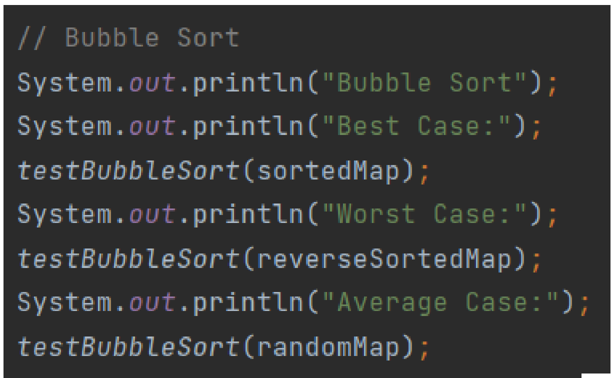
In insertion sort, we compare the value with current index with the value with index-1. If the previous value is greater than current value, we swap the current value with previous values until it reaches to index 0 or the previous value is smaller.

In bubble sort, we compare first two values and swapped them according to their values. Then we increase indexes and compare the values with index 1 and 2. By doing this, we are actually putting the greatest value to end and putting other values closer to their indexes that they should be in. We count the number of iterations because we don’t need to go to end of the array every time.

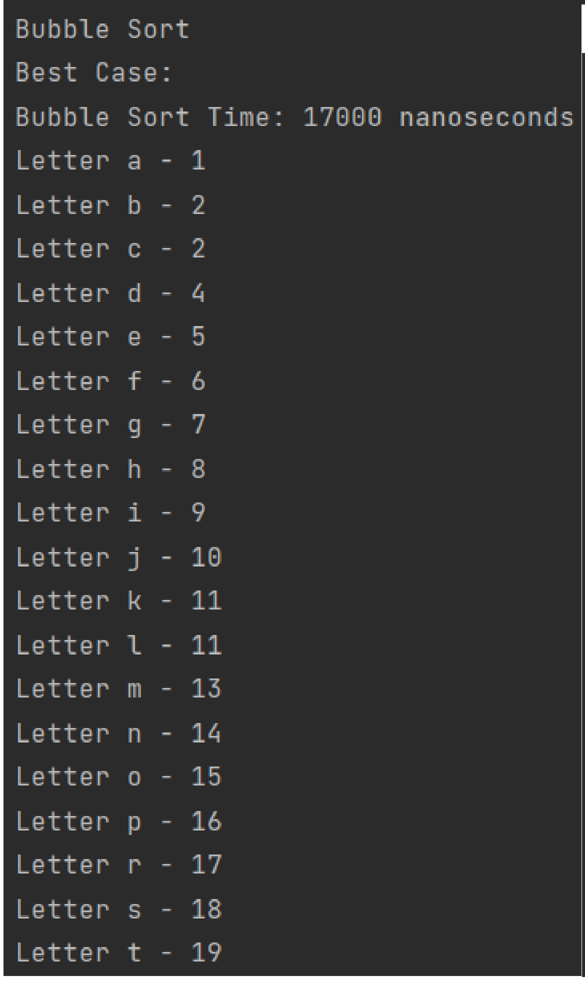
In quick sort we choose a pivot (first element in my code). We iterate through the array and put smaller values compared to pivot to left of it. So, the numbers smaller than pivot is in the left and greater is in the right. Then I called the same method to sort left and right separately.

4. TEST CASES AND RESULTS

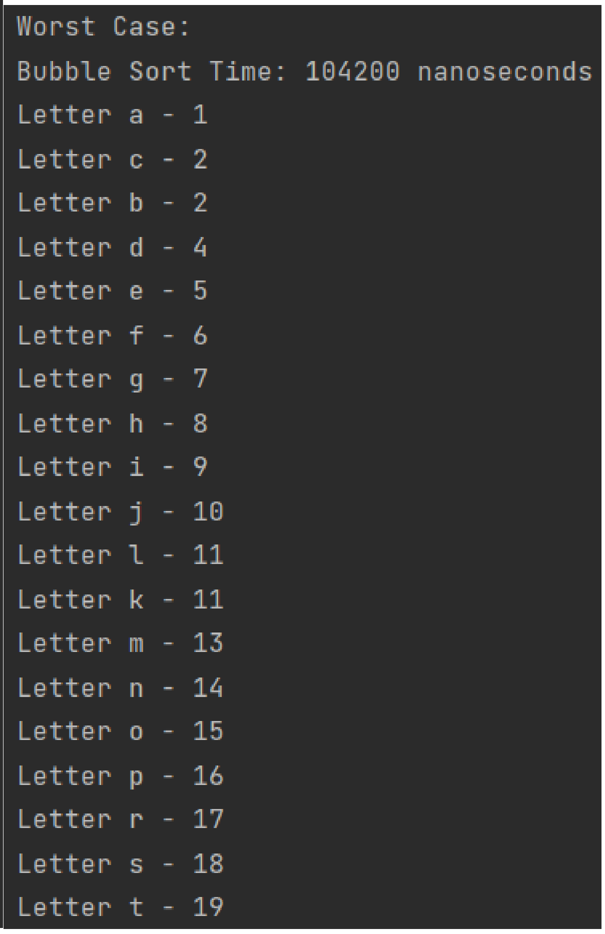
Bubble Sort Input and Output:



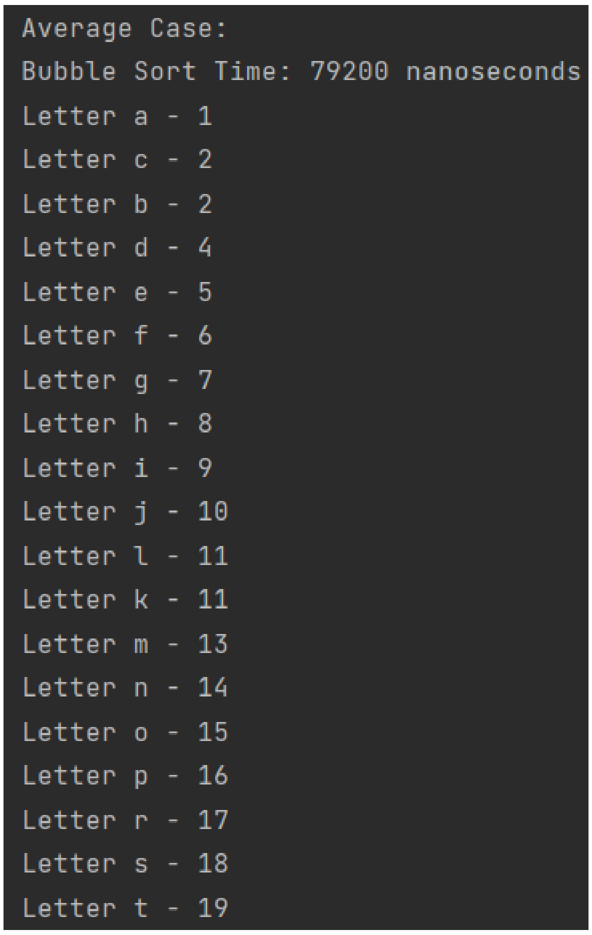
```java
// Bubble Sort
System.out.println("Bubble Sort");
System.out.println("Best Case:");
testBubbleSort(sortedMap);
System.out.println("Worst Case:");
testBubbleSort(reverseSortedMap);
System.out.println("Average Case:");
testBubbleSort(randomMap);
```



```
Bubble Sort
Best Case:
Bubble Sort Time: 17000 nanoseconds
Letter a - 1
Letter b - 2
Letter c - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter k - 11
Letter l - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

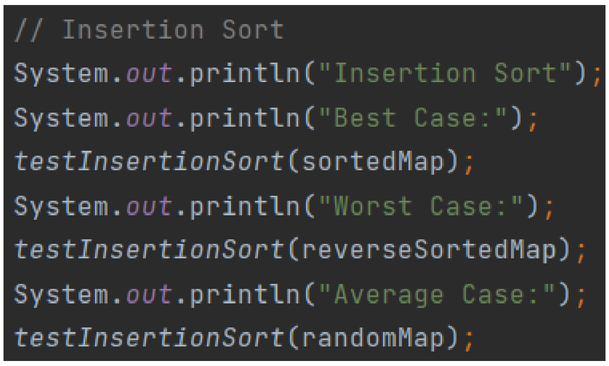


```
Worst Case:
Bubble Sort Time: 104200 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter l - 11
Letter k - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

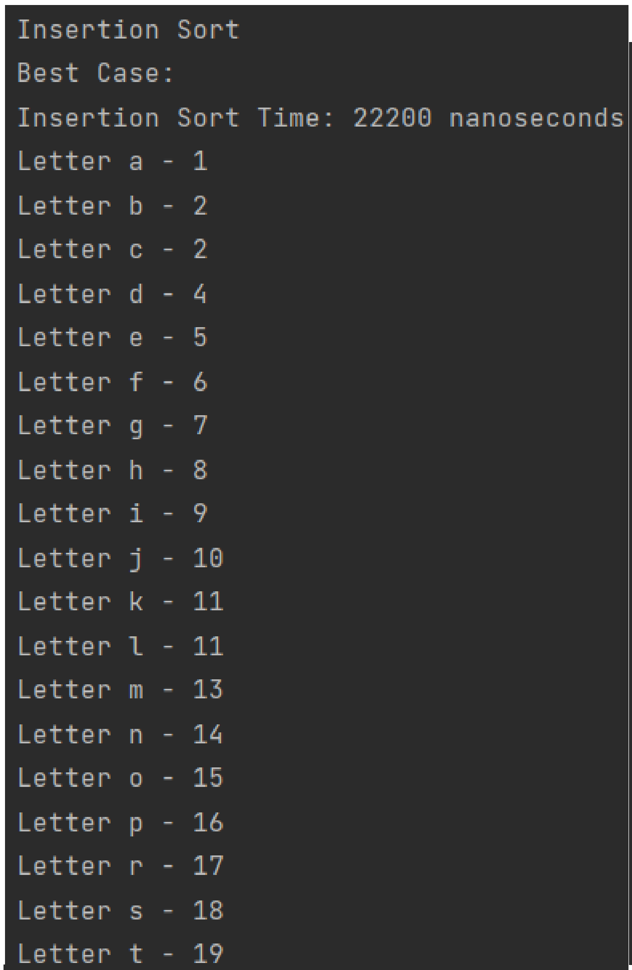


```
Average Case:
Bubble Sort Time: 79200 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter l - 11
Letter k - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

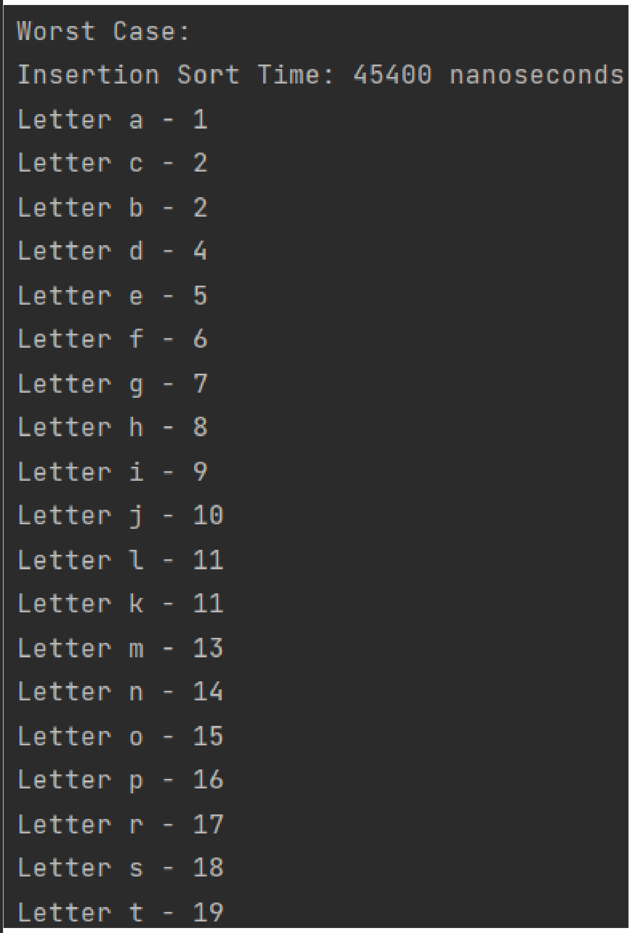
Insertion Sort Input and Output:



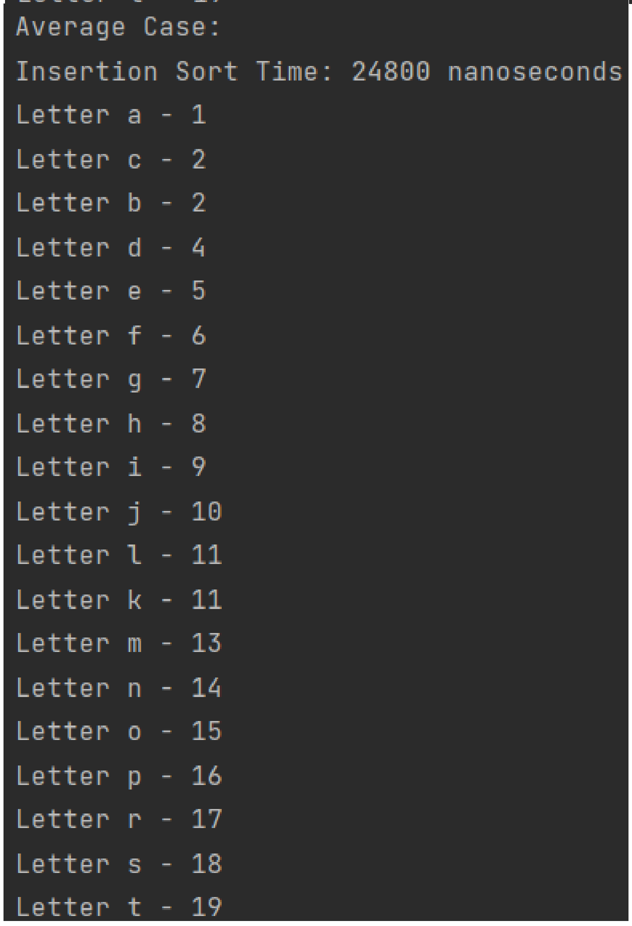
```java
// Insertion Sort
System.out.println("Insertion Sort");
System.out.println("Best Case:");
testInsertionSort(sortedMap);
System.out.println("Worst Case:");
testInsertionSort(reverseSortedMap);
System.out.println("Average Case:");
testInsertionSort(randomMap);
```



```
Insertion Sort
Best Case:
Insertion Sort Time: 22200 nanoseconds
Letter a - 1
Letter b - 2
Letter c - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter k - 11
Letter l - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

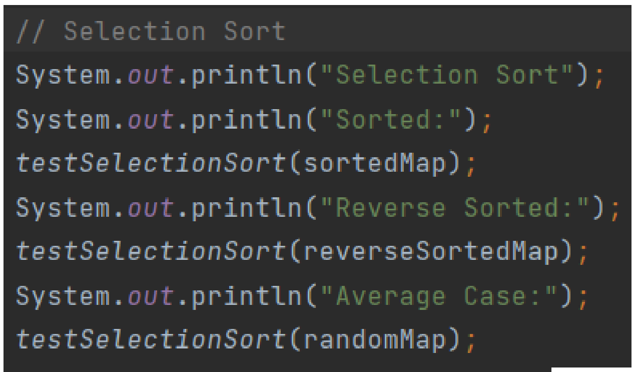


```
Worst Case:
Insertion Sort Time: 45400 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter l - 11
Letter k - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

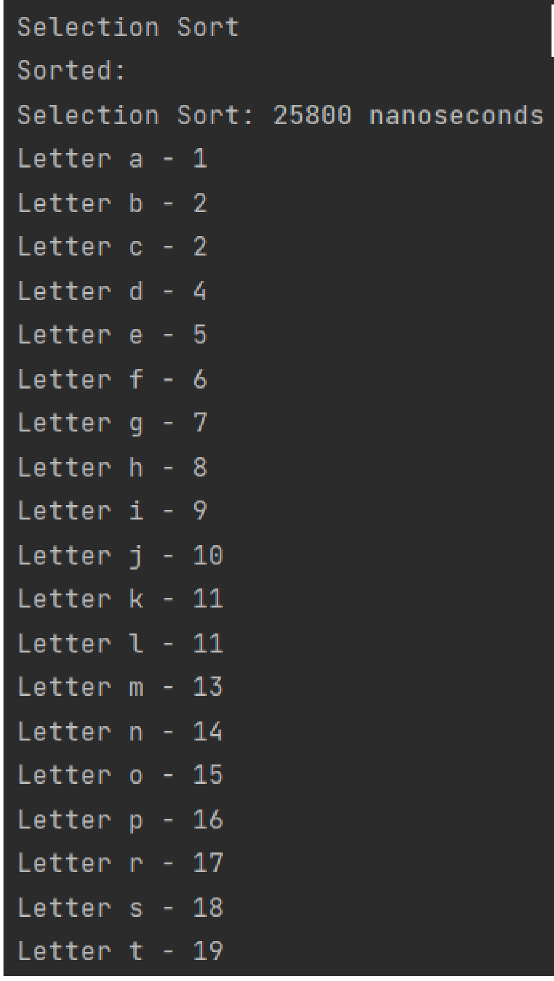


```
Average Case:
Insertion Sort Time: 24800 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter l - 11
Letter k - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

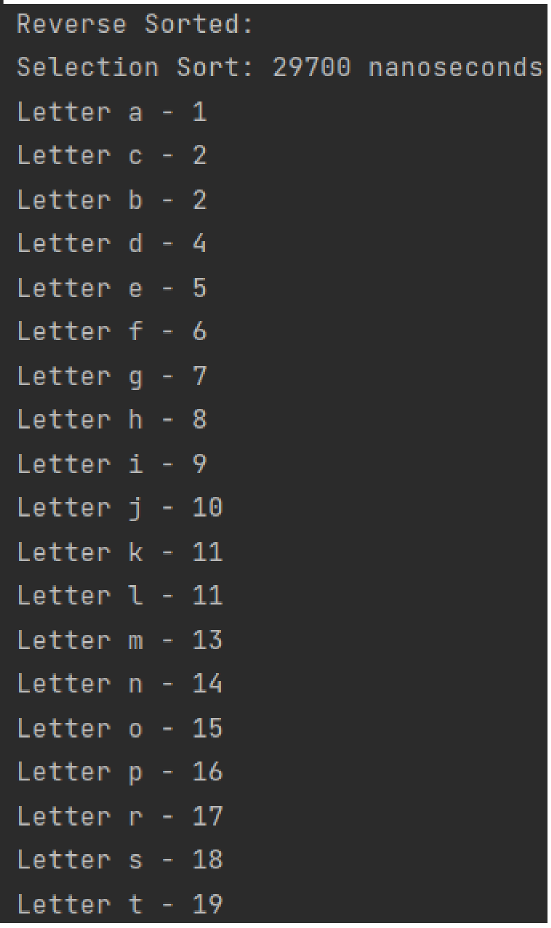
Selection Sort Input and Output:



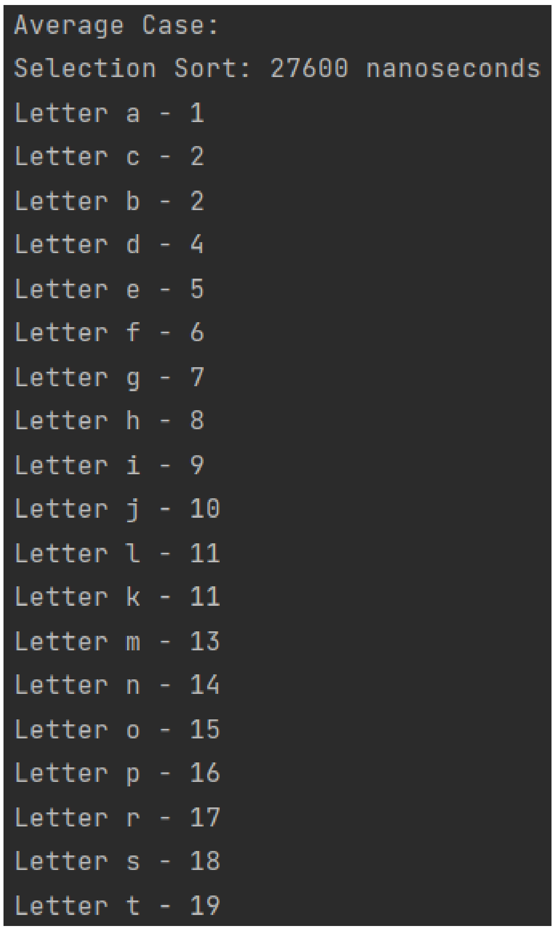
```
// Selection Sort
System.out.println("Selection Sort");
System.out.println("Sorted:");
testSelectionSort(sortedMap);
System.out.println("Reverse Sorted:");
testSelectionSort(reverseSortedMap);
System.out.println("Average Case:");
testSelectionSort(randomMap);
```



```
Selection Sort
Sorted:
Selection Sort: 25800 nanoseconds
Letter a - 1
Letter b - 2
Letter c - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter k - 11
Letter l - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

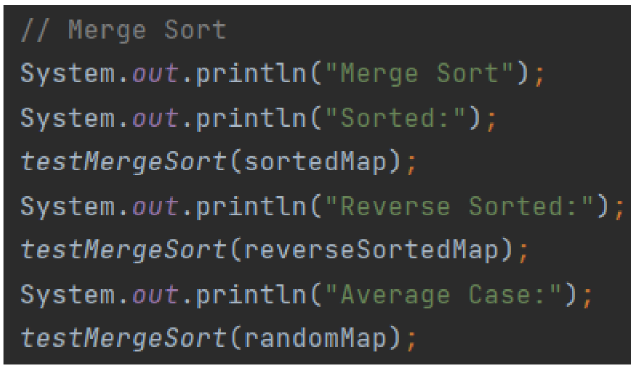


```
Reverse Sorted:
Selection Sort: 29700 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter k - 11
Letter l - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

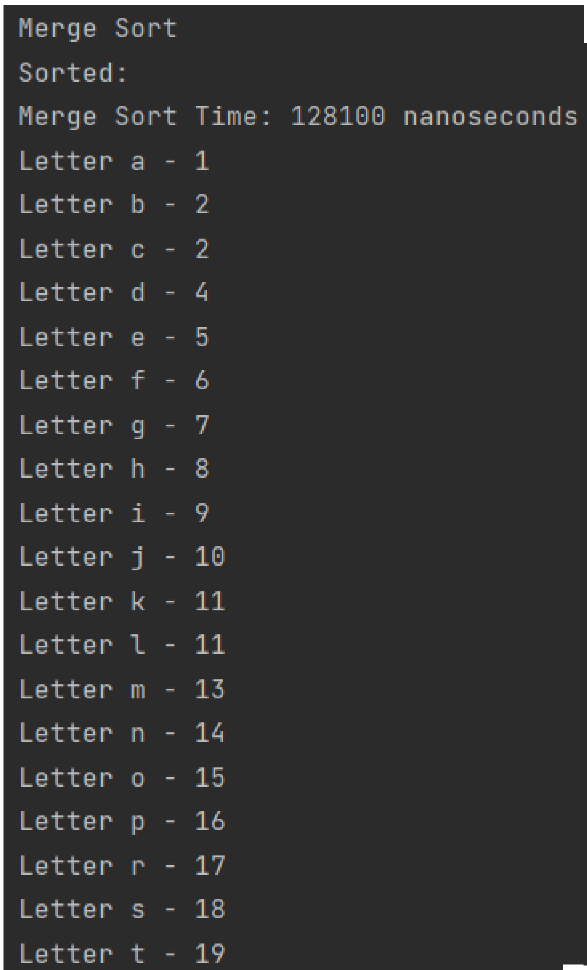


```
Average Case:
Selection Sort: 27600 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter l - 11
Letter k - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

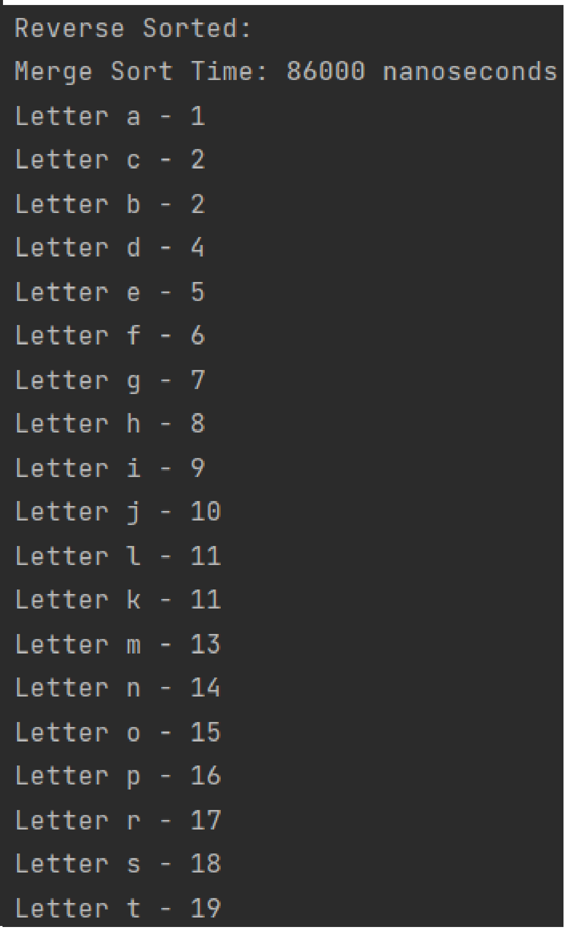
Merge Sort Input and Output:



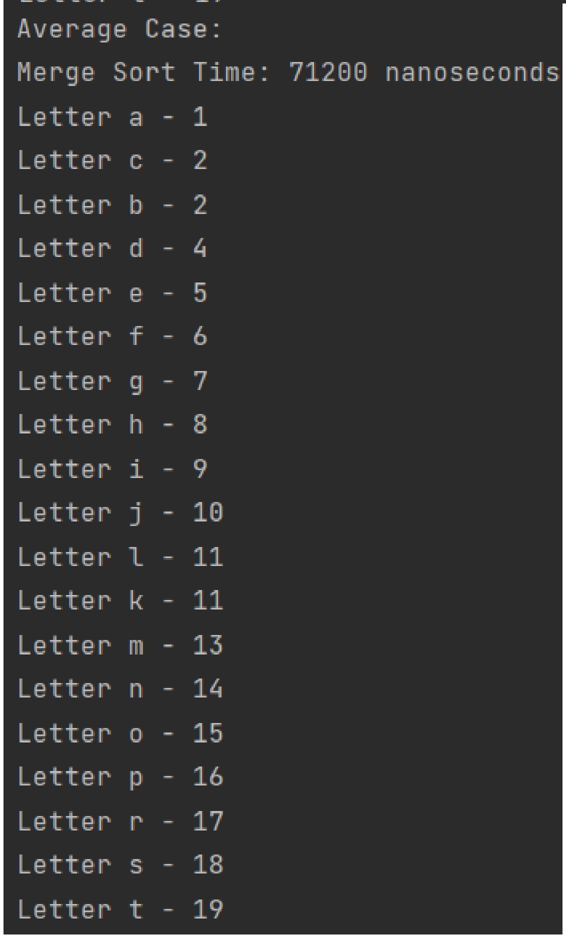
```
// Merge Sort
System.out.println("Merge Sort");
System.out.println("Sorted:");
testMergeSort(sortedMap);
System.out.println("Reverse Sorted:");
testMergeSort(reverseSortedMap);
System.out.println("Average Case:");
testMergeSort(randomMap);
```



```
Merge Sort
Sorted:
Merge Sort Time: 128100 nanoseconds
Letter a - 1
Letter b - 2
Letter c - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter k - 11
Letter l - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

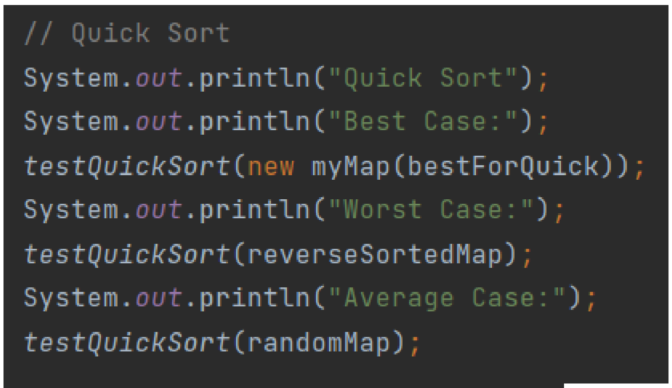


```
Reverse Sorted:
Merge Sort Time: 86000 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter l - 11
Letter k - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

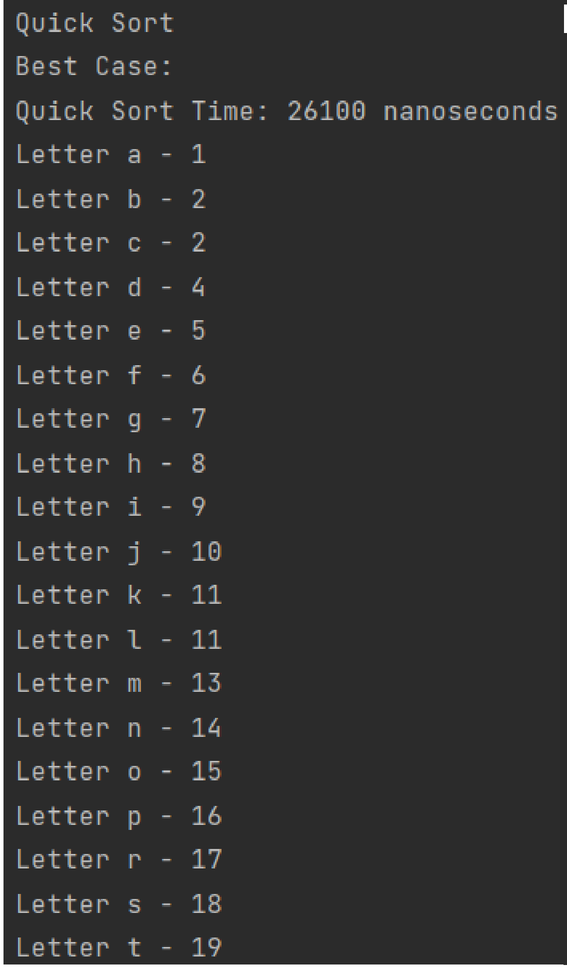


```
Average Case:
Merge Sort Time: 71200 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter l - 11
Letter k - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

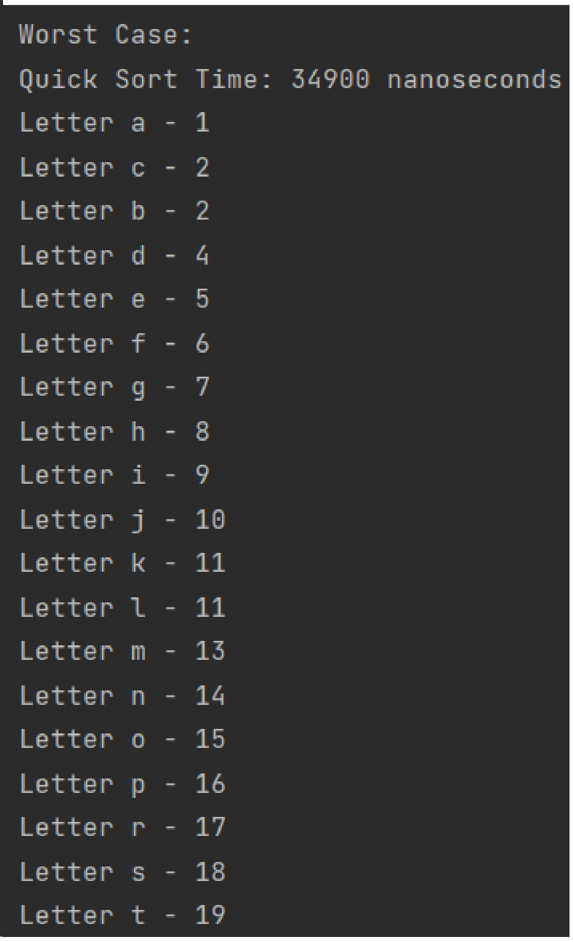
Quick Sort Input and Output



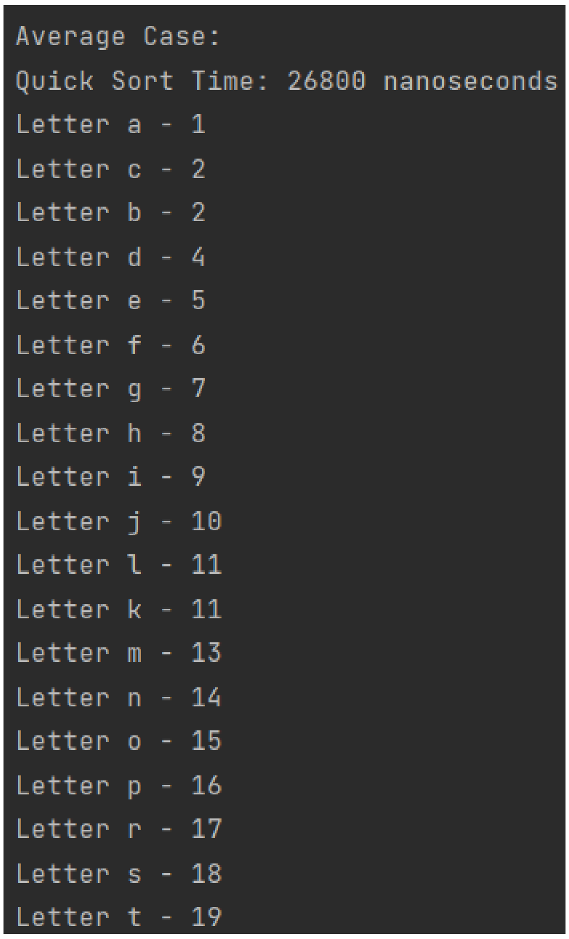
```
// Quick Sort
System.out.println("Quick Sort");
System.out.println("Best Case:");
testQuickSort(new myMap(bestForQuick));
System.out.println("Worst Case:");
testQuickSort(reverseSortedMap);
System.out.println("Average Case:");
testQuickSort(randomMap);
```



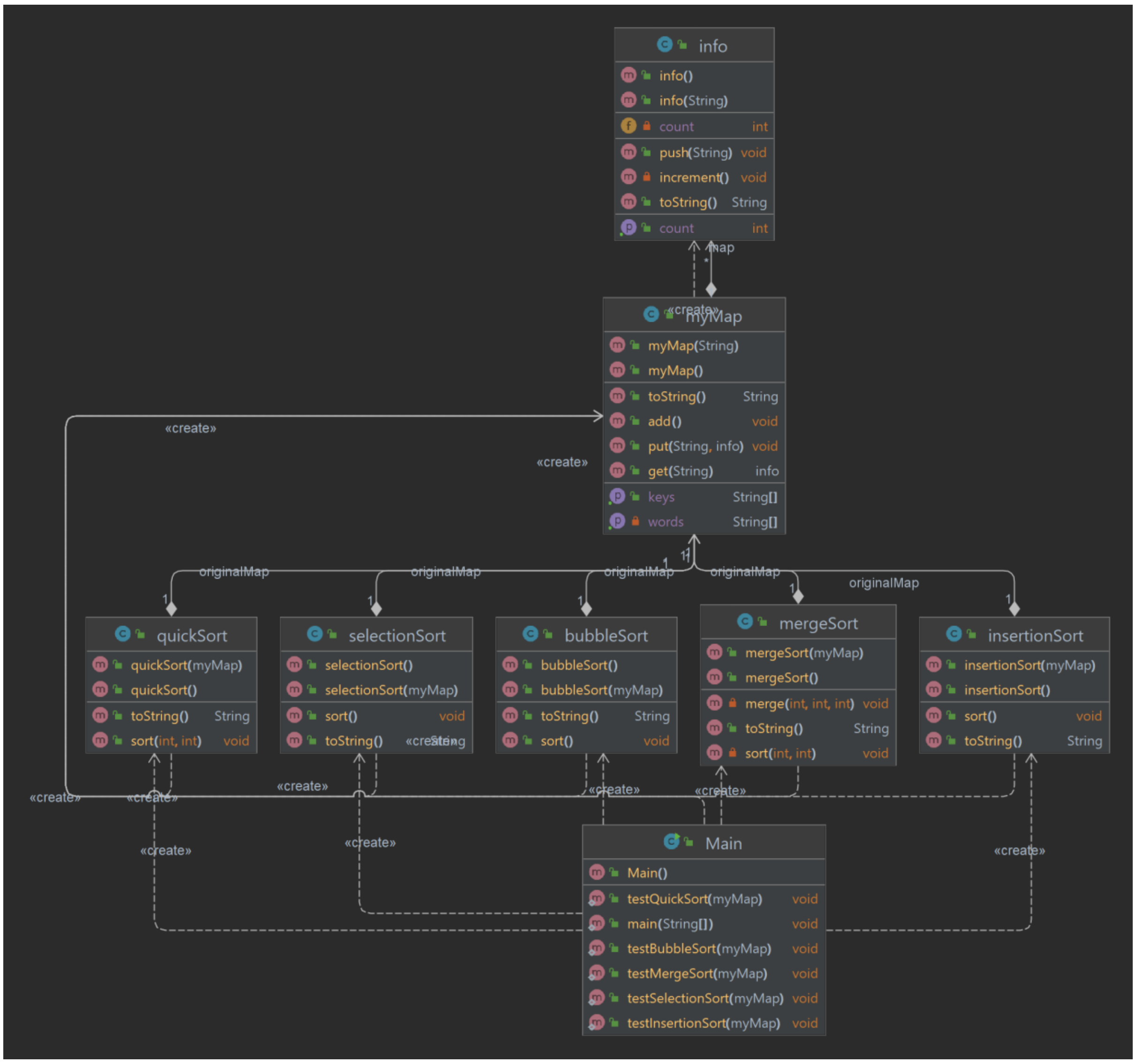
```
Quick Sort
Best Case:
Quick Sort Time: 26100 nanoseconds
Letter a - 1
Letter b - 2
Letter c - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter k - 11
Letter l - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```



```
Worst Case:
Quick Sort Time: 34900 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter k - 11
Letter l - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```


```
Average Case:
Quick Sort Time: 26800 nanoseconds
Letter a - 1
Letter c - 2
Letter b - 2
Letter d - 4
Letter e - 5
Letter f - 6
Letter g - 7
Letter h - 8
Letter i - 9
Letter j - 10
Letter l - 11
Letter k - 11
Letter m - 13
Letter n - 14
Letter o - 15
Letter p - 16
Letter r - 17
Letter s - 18
Letter t - 19
```

5.CLASS DIAGRAM



6. ANSWERS FOR PART 2

Part A

Sort/Case	Best Case	Average Case	Worst Case
Merge	O(nlogn)	O(nlogn)	O(nlogn)
Selection	O(n^2)	O(n^2)	O(n^2)
Insertion	O(n)	O(n^2)	O(n^2)
Bubble	O(n)	O(n^2)	O(n^2)
Quick	O(nlogn)	O(nlogn)	O(n^2)

In merge sort we divide the array into small pieces then sort and merge them until we get the full array. Merge sort uses divide and conquer strategy that's the reason it's faster than others in big arrays (n>1000). Dividing the array takes O(logn) time and merging takes O(n) time which gives us the result of O(nlogn).

Selection sort finds the smallest value and places it starting from the start of the array. Iterating through an array takes O(n) time and we are doing this for every element in the array. This means that it takes O(n^2) time to sort in every case.

Insertion sort, sorts an array one by one. An element is being shifted to left if it's smaller than the element in the left. In the best case, the array is already sorted so it doesn't shift anything but still checks every element. Because of this check, in best case it's O(n). The worst case is that the array is reverse sorted. In that case, it will shift every element n times. So, it will do n^2 operations. In average case it will make n/2 comparisons and n/2 shifts which is still equal to O(n^2).

Bubble sort, sorts an array by placing the greatest value to the end first while shifting others into their positions. The best case for it is that the array is already sorted. It will only make comparisons n times and not going to shift anything which gives us the result of O(n). The worst case is that the array is reverse sorted. It will shift n elements and every element will be shifted at average of (n-1)/2. This gives us the O(n^2). At average we are going to shift n/2 elements by (n-1)/4 times which means notation is O(n^2).

In quick sort we use the divide and conquer strategy. We divide the array into two according to chosen pivot. Elements that are less than pivot are shifted to left and rest is in the right. In the best case, pivot is right in the middle every time we divide the array or subarrays. We will divide it logn times and every subarray will take n time to sort in average. That gives us the result of O(nlogn). Average case is close to this so it is also O(nlogn). At worst case, the pivot is chosen poorly every time which results in having unbalanced divisions. That will make this sort useless. We will have n subarrays which takes n time to sort in average. That's why at worst case it takes O(n^2).

Part B (in nanoseconds)

Sort/Case	Best	Average	Worst
Merge	128100	71200	86000
Selection	25800	27600	29700
Insertion	22200	24800	45400
Bubble	17000	79200	104200
Quick	26100	26800	34900

Part C

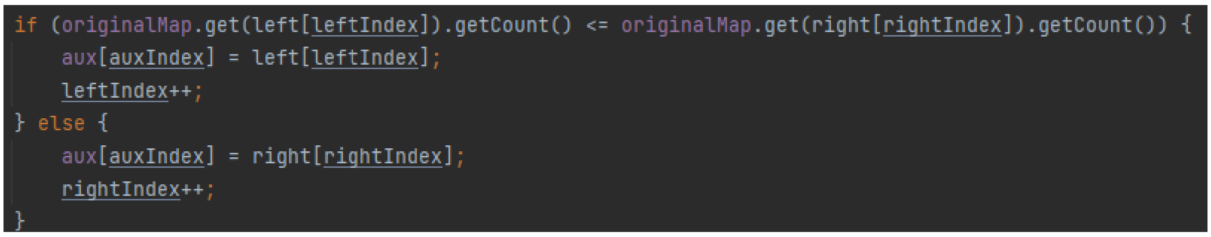
Bubble and insertion sort are only better when the array is sorted which is rarely the case. In other cases, they are very bad. It can only be used while sorting small arrays (n<1000).

Insertion sort is clearly worst to use out of these 5 sort algorithms. Since it's simplicity, maybe using it for educational purposes be viable. Also, using it for small arrays is not great but ok.

Merge and quick sort are generally slower in small arrays but for large arrays they are much better.

Part D

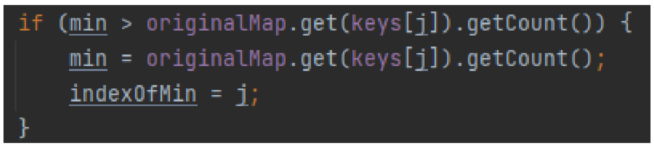
Merge Sort:



```java
if (originalMap.get(left[leftIndex]).getCount() <= originalMap.get(right[rightIndex]).getCount()) {
    aux[auxIndex] = left[leftIndex];
    leftIndex++;
} else {
    aux[auxIndex] = right[rightIndex];
    rightIndex++;
}
```

The "<=" operator gives priority to left element over right when they both have the same value.

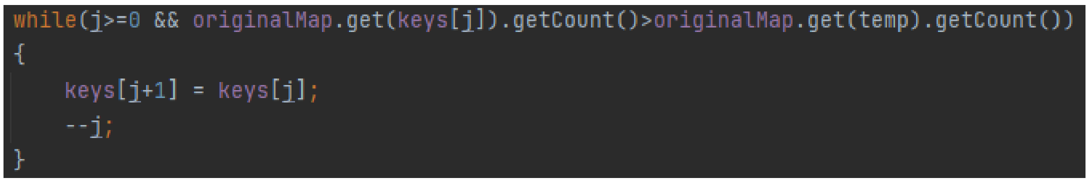
Selection Sort:



```java
if (min > originalMap.get(keys[j]).getCount()) {
    min = originalMap.get(keys[j]).getCount();
    indexOfMin = j;
}
```

Since we try to find the smaller value than selected one, we don't have to put ">=" instead of ">". So, the value in the left will stay in the left.

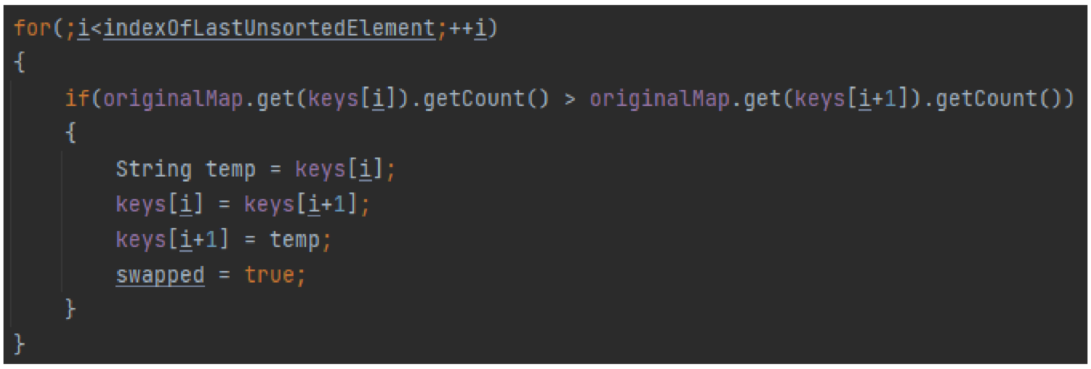
Insertion Sort:



```
while(j>=0 && originalMap.get(keys[j]).getCount()>originalMap.get(temp).getCount())
{
    keys[j+1] = keys[j];
    --j;
}
```

Since we have to shift until we find a smaller value then the selected value, there is no point for the same right value to shift to left. Instead, it just stands in the right.

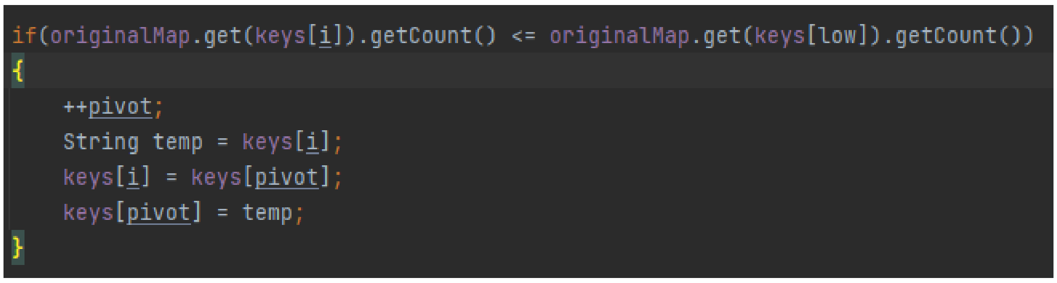
Bubble Sort:



```
for(;i<indexOfLastUnsortedElement;++i)
{
    if(originalMap.get(keys[i]).getCount() > originalMap.get(keys[i+1]).getCount())
    {
        String temp = keys[i];
        keys[i] = keys[i+1];
        keys[i+1] = temp;
        swapped = true;
    }
}
```

If statement works only when the left value is greater than right and not when it's equal. It prevents for an extra shift operation.

Quick Sort:



```
if(originalMap.get(keys[i]).getCount() <= originalMap.get(keys[low]).getCount())
{
    ++pivot;
    String temp = keys[i];
    keys[i] = keys[pivot];
    keys[pivot] = temp;
}
```

Since I used "<=" operator, and picked the pivot using lowest index; only the pivot will be on the right and every other value that's same with pivot will preserve their positions. So, it's unstable and values may or may not keep their positions.