GTU Department Of Computer Engineering

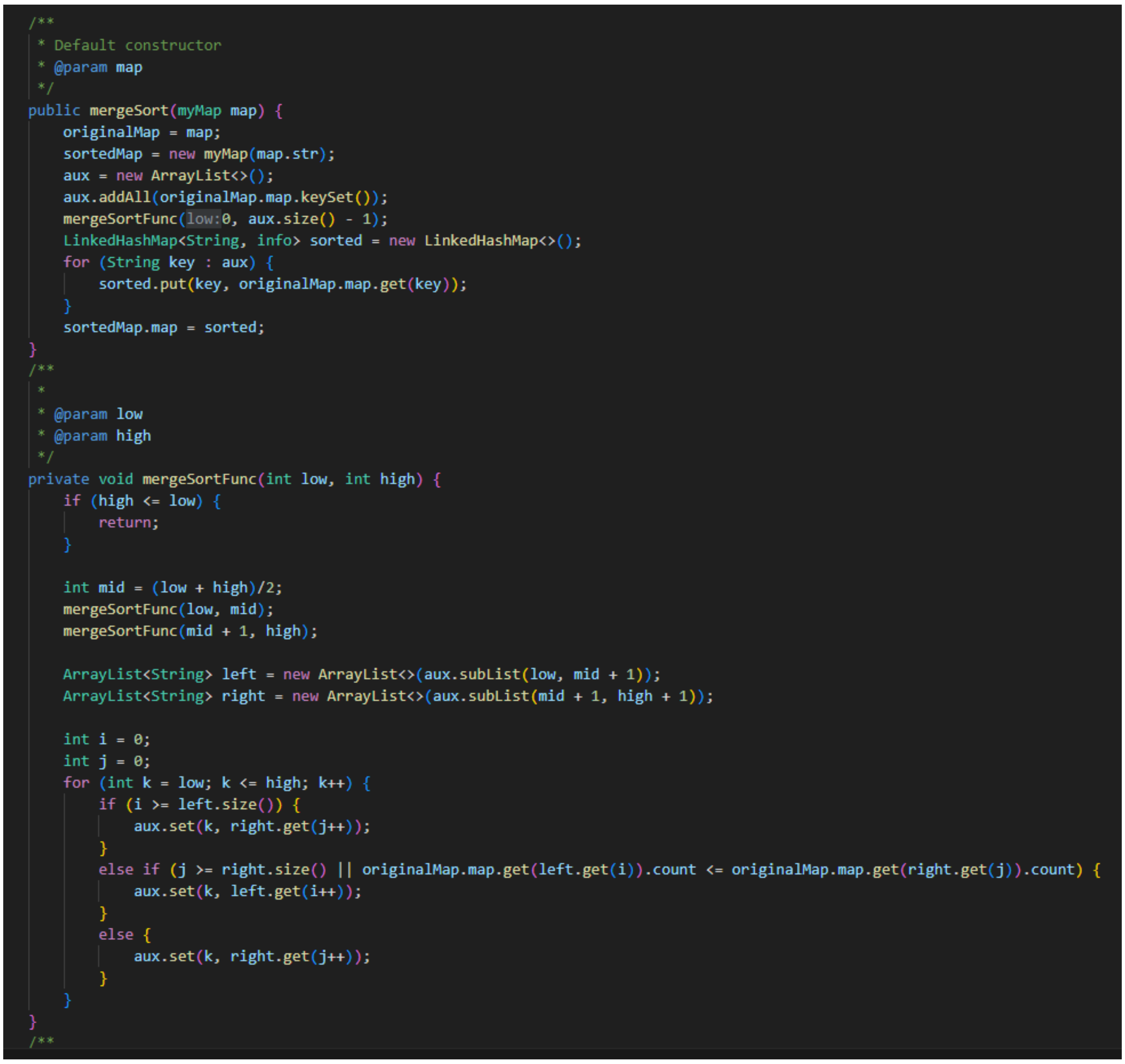
CSE 222 / 505 – Spring 2023

Homework 7

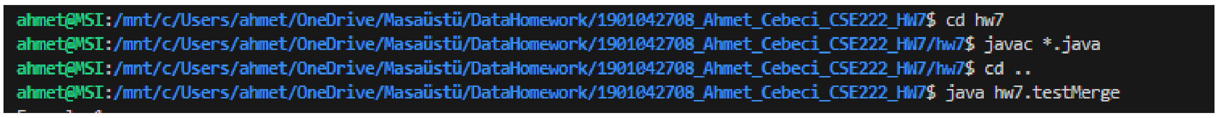
Ahmet CEBECİ

1901042708

1 – Merge Sort



```java
/**
 * Default constructor
 * @param map
 */
public mergeSort(myMap map) {
    originalMap = map;
    sortedMap = new myMap(map.str);
    aux = new ArrayList<>();
    aux.addAll(originalMap.map.keySet());
    mergeSortFunc(low:0, aux.size() - 1);
    LinkedHashMap<String, info> sorted = new LinkedHashMap<>();
    for (String key : aux) {
        sorted.put(key, originalMap.map.get(key));
    }
    sortedMap.map = sorted;
}
/**
 *
 * @param low
 * @param high
 */
private void mergeSortFunc(int low, int high) {
    if (high <= low) {
        return;
    }

    int mid = (low + high)/2;
    mergeSortFunc(low, mid);
    mergeSortFunc(mid + 1, high);

    ArrayList<String> left = new ArrayList<>(aux.subList(low, mid + 1));
    ArrayList<String> right = new ArrayList<>(aux.subList(mid + 1, high + 1));

    int i = 0;
    int j = 0;
    for (int k = low; k <= high; k++) {
        if (i >= left.size()) {
            aux.set(k, right.get(j++));
        }
        else if (j >= right.size() || originalMap.map.get(left.get(i)).count <= originalMap.map.get(right.get(j)).count) {
            aux.set(k, left.get(i++));
        }
        else {
            aux.set(k, right.get(j++));
        }
    }
}
/**
```



```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ cd hw7
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ javac *.java
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ cd ..
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ java hw7.testMerge
```

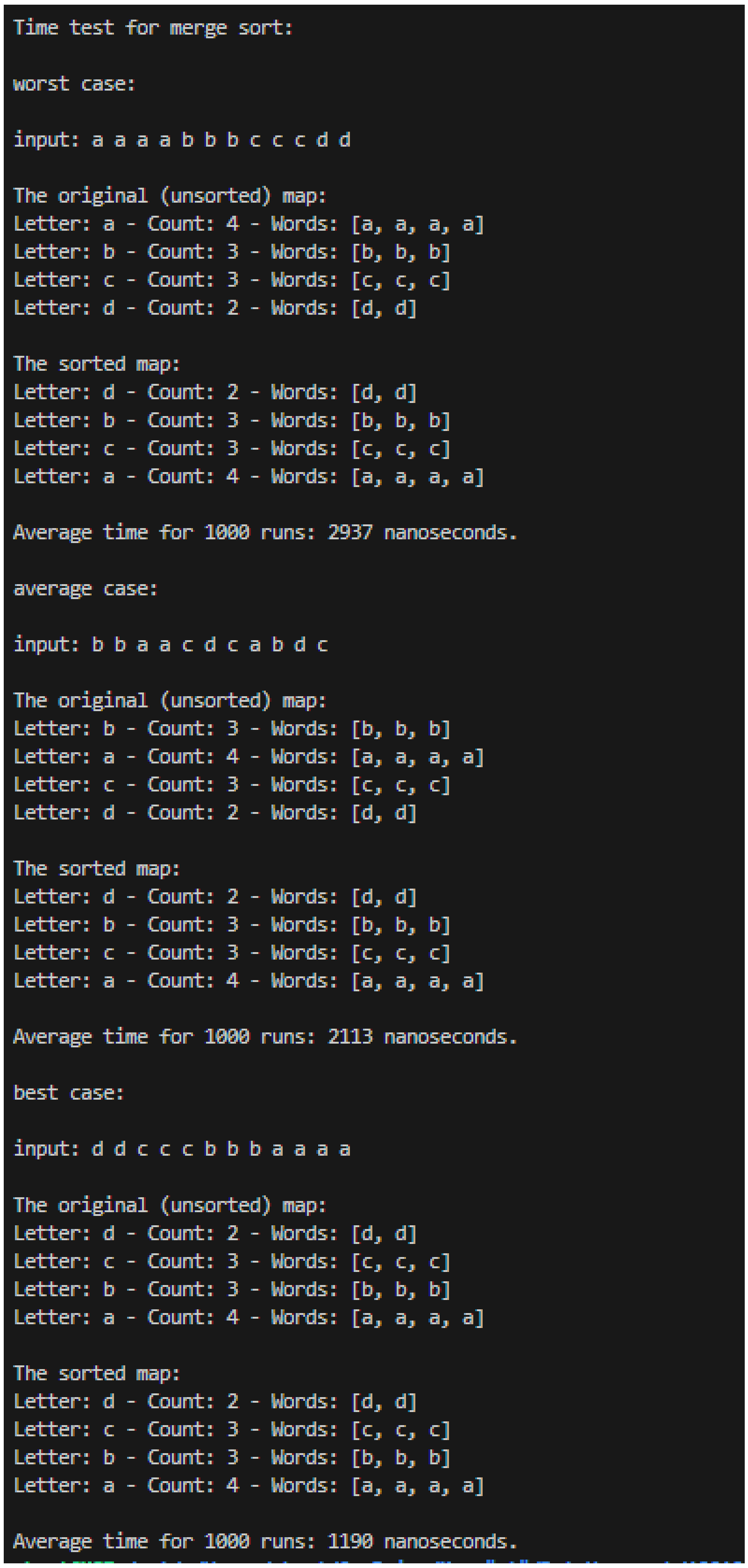
This is a divide-and-conquer algorithm that splits an array in half, sorts the halves separately, and then merges them.

Best-case scenario: Unlike some sorting algorithms, Merge Sort doesn't have a specific "best-case" input. Regardless of the input's initial order, Merge Sort always splits the array in half until it has divided the entire array into individual elements, sorts these elements, and then merges them, always performing in O(n log n) time complexity.

Average-case scenario: Just like the best case, the average case also performs in O(n log n) time complexity. This is one of the significant advantages of Merge Sort - its performance is consistent, regardless of the input's initial order.

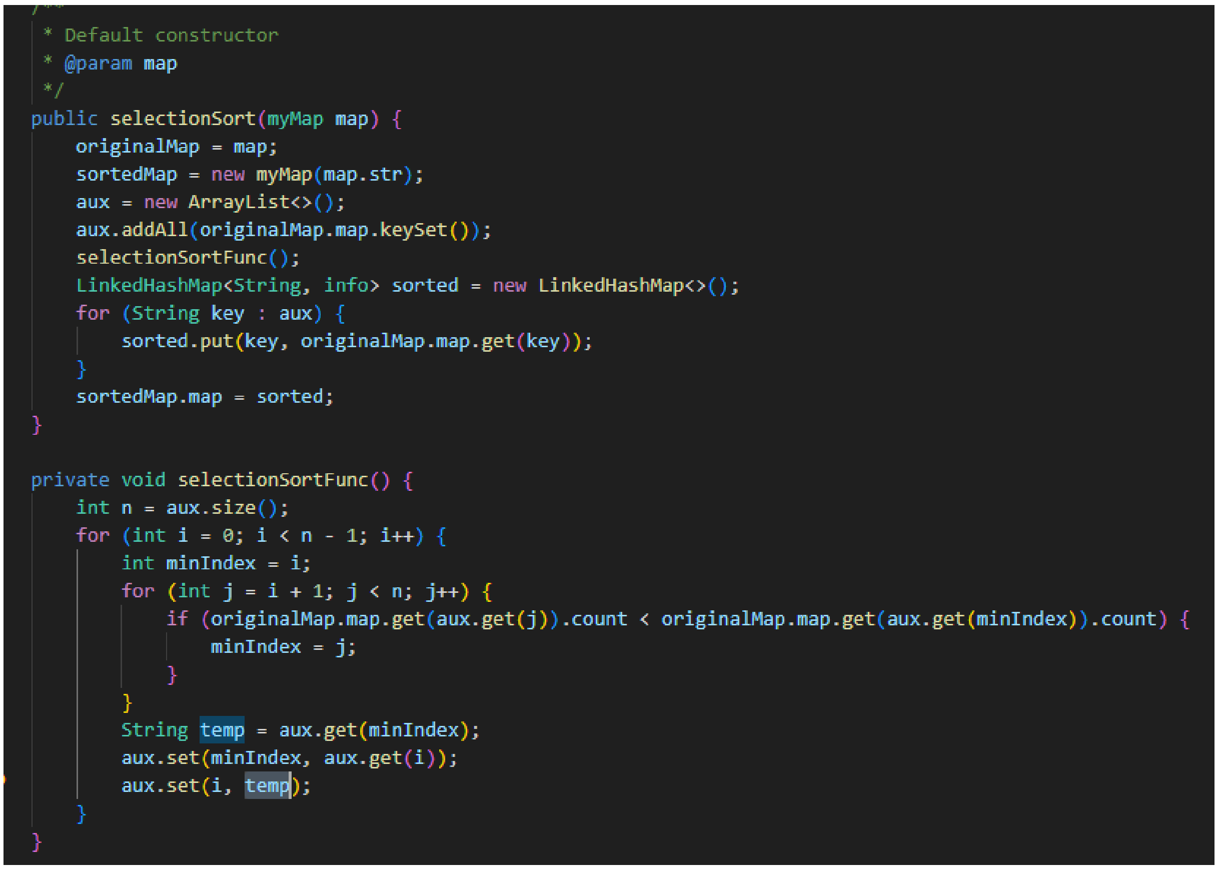
Worst-case scenario: Even in the worst-case scenario, Merge Sort maintains a time complexity of O(n log n). This makes it a preferred choice for large datasets where worst-case performance is a concern.

It's important to mention that Merge Sort is a stable sorting algorithm, meaning that it maintains the relative order of equal data elements. For example, if you have records sorted by name and you sort them by age using merge sort, two people of the same age will maintain their relative order from when they were sorted by name. This stability is beneficial when sorting by multiple criteria (e.g., sort by age, then by name).

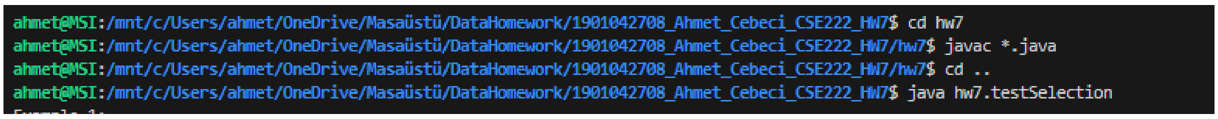


```
Time test for merge sort:

worst case:

input: a a a a b b b c c c d d

The original (unsorted) map:
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 2937 nanoseconds.

average case:

input: b b a a c d c a b d c

The original (unsorted) map:
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 2113 nanoseconds.

best case:

input: d d c c c b b b a a a a

The original (unsorted) map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 1190 nanoseconds.
```

2 – Selection Sort



```java
 * Default constructor
 * @param map
 */
public selectionSort(myMap map) {
    originalMap = map;
    sortedMap = new myMap(map.str);
    aux = new ArrayList<>();
    aux.addAll(originalMap.map.keySet());
    selectionSortFunc();
    LinkedHashMap<String, info> sorted = new LinkedHashMap<>();
    for (String key : aux) {
        sorted.put(key, originalMap.map.get(key));
    }
    sortedMap.map = sorted;
}

private void selectionSortFunc() {
    int n = aux.size();
    for (int i = 0; i < n - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (originalMap.map.get(aux.get(j)).count < originalMap.map.get(aux.get(minIndex)).count) {
                minIndex = j;
            }
        }
        String temp = aux.get(minIndex);
        aux.set(minIndex, aux.get(i));
        aux.set(i, temp);
    }
}
```



```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ cd hw7
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ javac *.java
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ cd ..
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ java hw7.testSelection
```

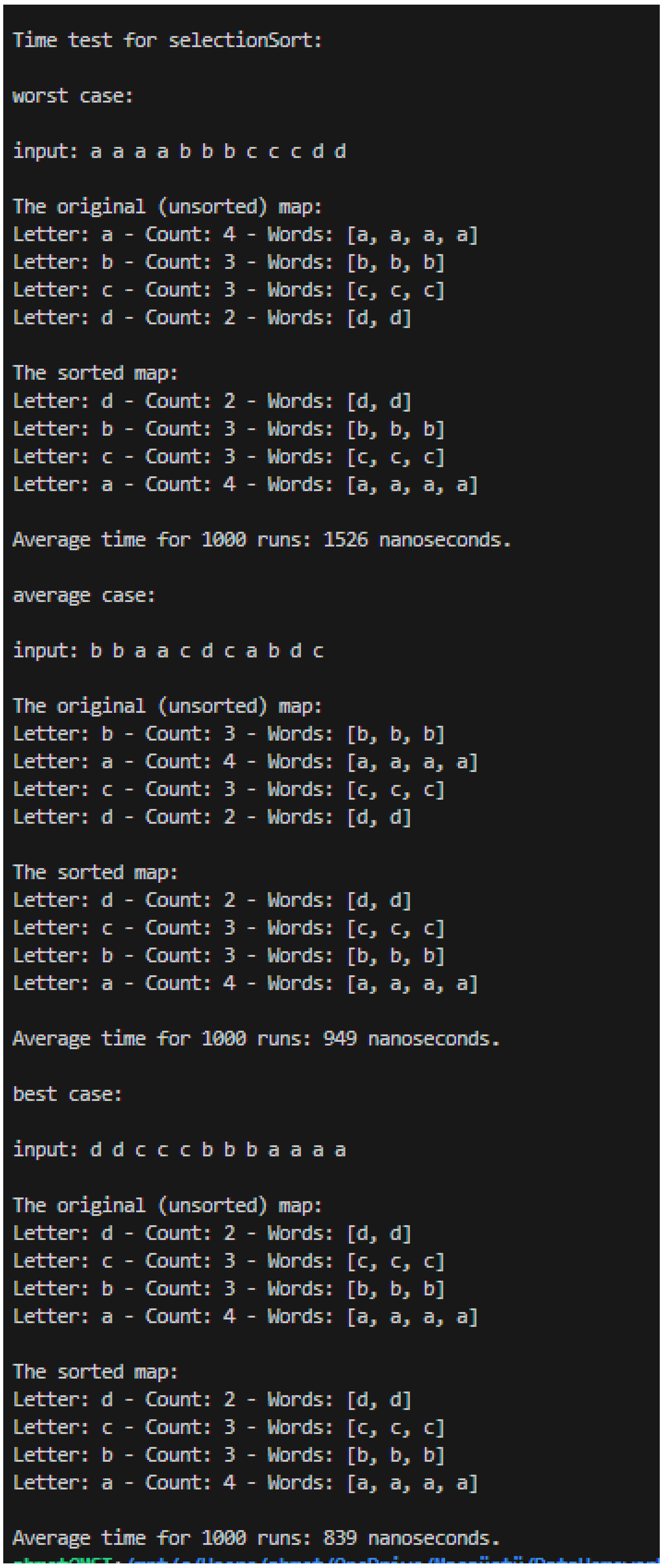
This algorithm consistently has a time complexity of O(n^2), regardless of the order of the input. This is true for the best, average, and worst-case scenarios because the algorithm always needs to scan through the entire array to find the current smallest (or largest) item.

Best-case scenario: Although the algorithm has O(n^2) performance in all cases, an array that is already sorted would represent the "best" case. However, this will not have a real effect on performance.

Average-case scenario: This is typically for a randomly ordered array.

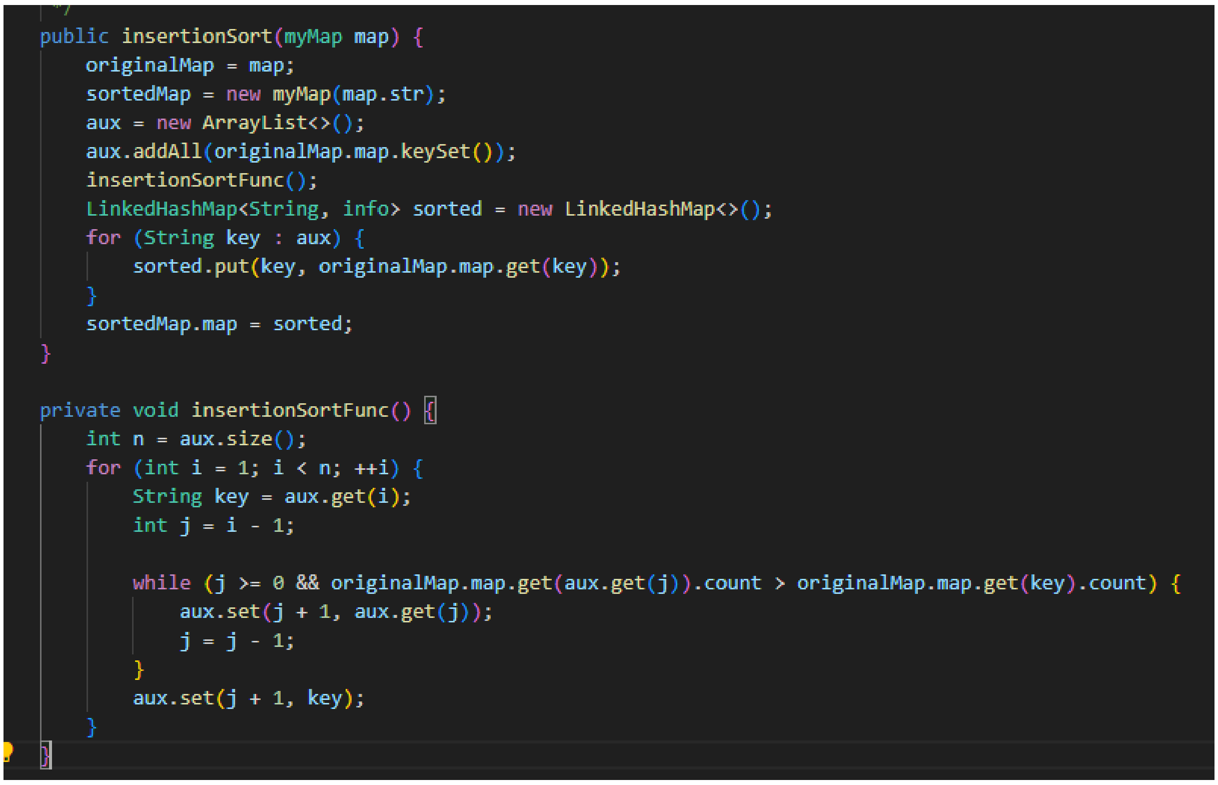
Worst-case scenario: Because the algorithm has O(n^2) time complexity in all cases, the worst case can be any input, regardless of the order of sorting.

Selection sort can be less efficient than Insertion Sort or Bubble Sort, especially when the array is already sorted or only a few items are out of place. This is because Selection Sort always needs to scan through the entire array and find the smallest (or largest) item. It's also important to note that Selection Sort is not a stable sorting algorithm, meaning the input order of items with equal key values may not be preserved.

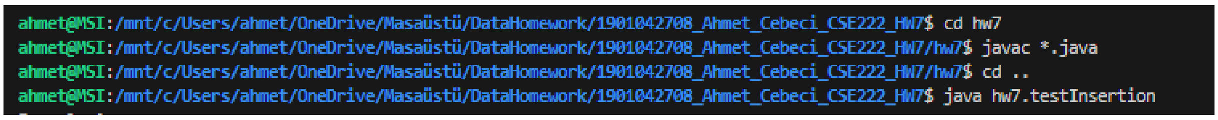


```
Time test for selectionSort:

worst case:

input: a a a a b b b c c c d d

The original (unsorted) map:
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 1526 nanoseconds.

average case:

input: b b a a c d c a b d c

The original (unsorted) map:
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 949 nanoseconds.

best case:

input: d d c c c b b b a a a a

The original (unsorted) map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 839 nanoseconds.
[illegible]
```

3 – Insertion Sort



```java
public insertionSort(myMap map) {
    originalMap = map;
    sortedMap = new myMap(map.str);
    aux = new ArrayList<>();
    aux.addAll(originalMap.map.keySet());
    insertionSortFunc();
    LinkedHashMap<String, info> sorted = new LinkedHashMap<>();
    for (String key : aux) {
        sorted.put(key, originalMap.map.get(key));
    }
    sortedMap.map = sorted;
}

private void insertionSortFunc() {
    int n = aux.size();
    for (int i = 1; i < n; ++i) {
        String key = aux.get(i);
        int j = i - 1;

        while (j >= 0 && originalMap.map.get(aux.get(j)).count > originalMap.map.get(key).count) {
            aux.set(j + 1, aux.get(j));
            j = j - 1;
        }
        aux.set(j + 1, key);
    }
}
```



```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ cd hw7
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ javac *.java
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ cd ..
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ java hw7.testInsertion
```

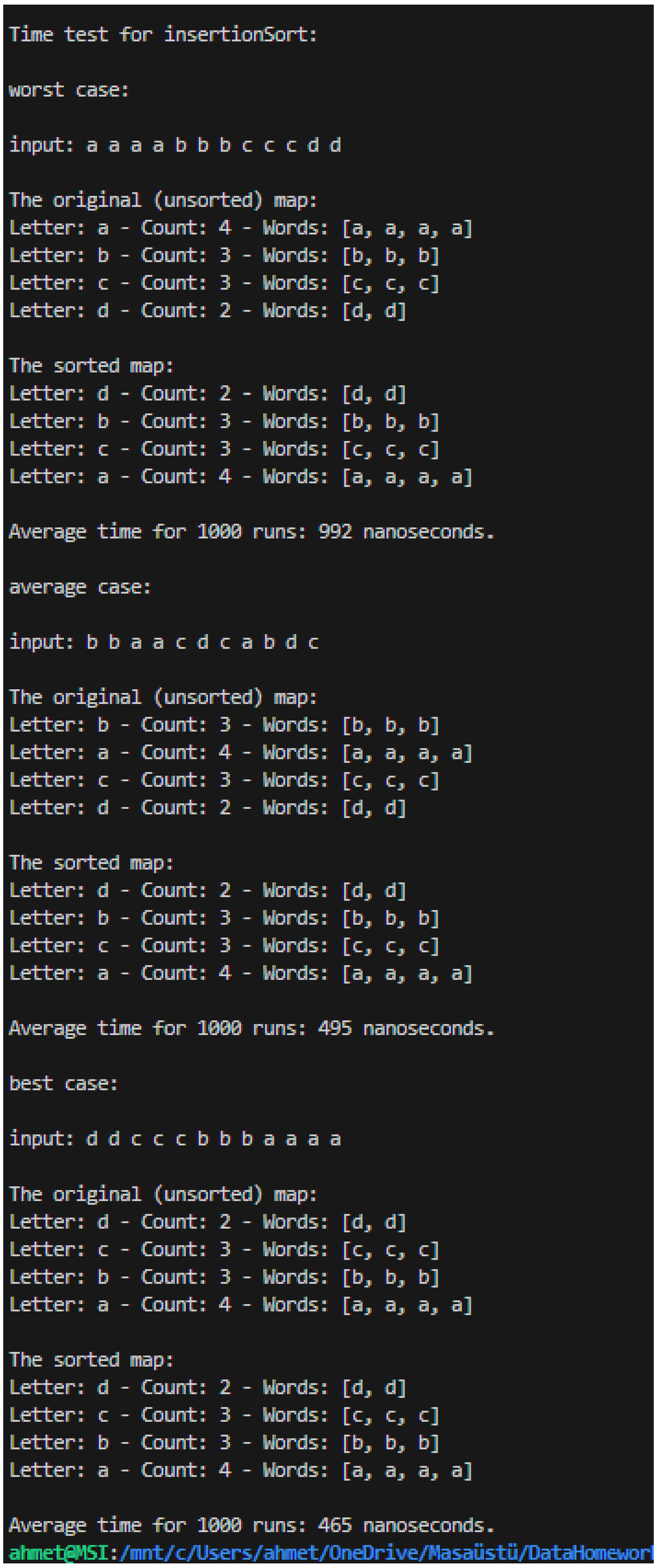
This algorithm sorts an array by building a sorted array one item at a time. It assumes that the first item is already sorted and then inserts the next items in the correct position within the sorted list.

Best-case scenario: The best-case scenario for Insertion Sort is when the input is already sorted. In this case, the algorithm merely iterates through the array once and makes no swaps. The time complexity in this best-case scenario is O(n), which makes it efficient for small or nearly sorted arrays.

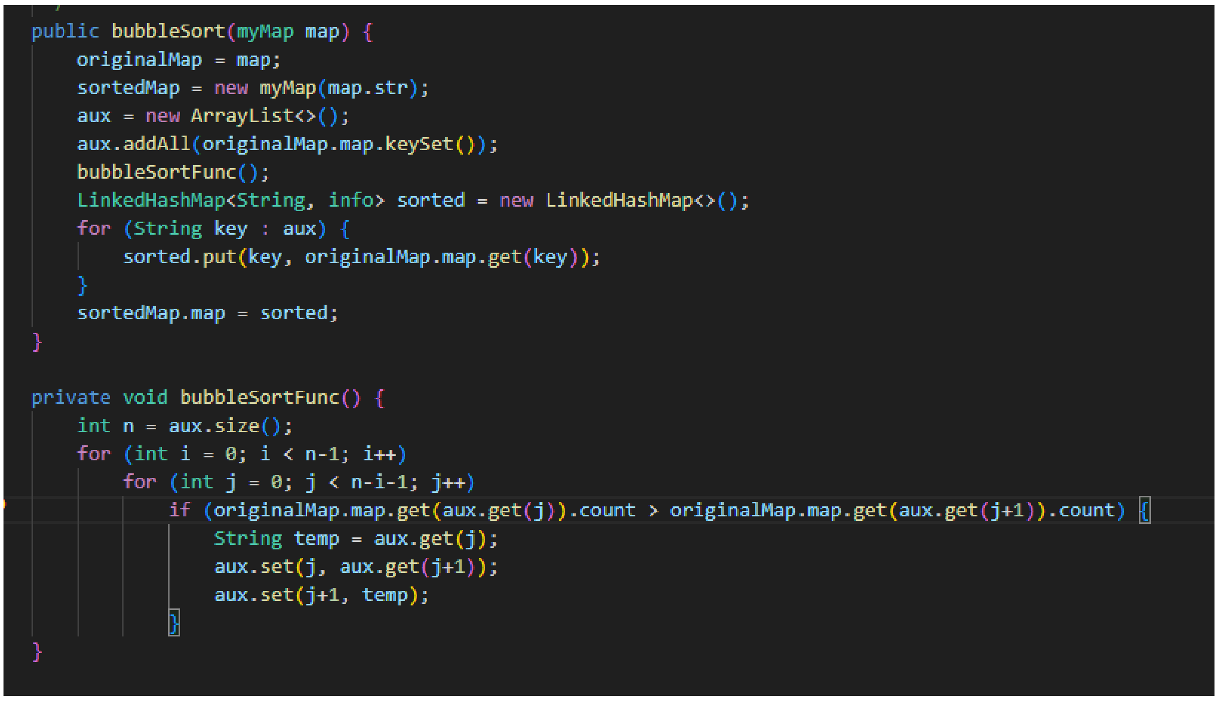
Average-case scenario: The average-case scenario is usually a randomly ordered array. In this case, Insertion Sort performs at a time complexity of O(n^2), which is less efficient for larger arrays.

Worst-case scenario: The worst-case scenario for Insertion Sort is when the input array is sorted in reverse order. In this case, for each item, the algorithm needs to move all previously sorted items, resulting in a time complexity of O(n^2).

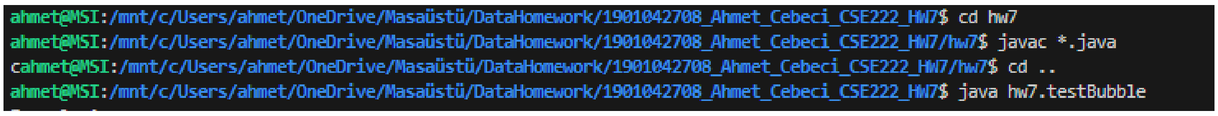
An important aspect of Insertion Sort is its stability. It maintains the relative order of equal data elements. This property can be beneficial in situations where relative order is important. Additionally, due to its low overhead, Insertion Sort tends to be faster on small lists or on lists that are already partially sorted, compared to other more complex algorithms.


```
Time test for insertionSort:

worst case:

input: a a a a b b b c c c d d

The original (unsorted) map:
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 992 nanoseconds.

average case:

input: b b a a c d c a b d c

The original (unsorted) map:
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 495 nanoseconds.

best case:

input: d d c c c b b b a a a a

The original (unsorted) map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 465 nanoseconds.
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomewor
```

4 – Bubble Sort



```java
public bubbleSort(myMap map) {
    originalMap = map;
    sortedMap = new myMap(map.str);
    aux = new ArrayList<>();
    aux.addAll(originalMap.map.keySet());
    bubbleSortFunc();
    LinkedHashMap<String, info> sorted = new LinkedHashMap<>();
    for (String key : aux) {
        sorted.put(key, originalMap.map.get(key));
    }
    sortedMap.map = sorted;
}

private void bubbleSortFunc() {
    int n = aux.size();
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-i-1; j++)
            if (originalMap.map.get(aux.get(j)).count > originalMap.map.get(aux.get(j+1)).count) {
                String temp = aux.get(j);
                aux.set(j, aux.get(j+1));
                aux.set(j+1, temp);
            }
}
```



```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ cd hw7
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ javac *.java
cahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ cd ..
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ java hw7.testBubble
```

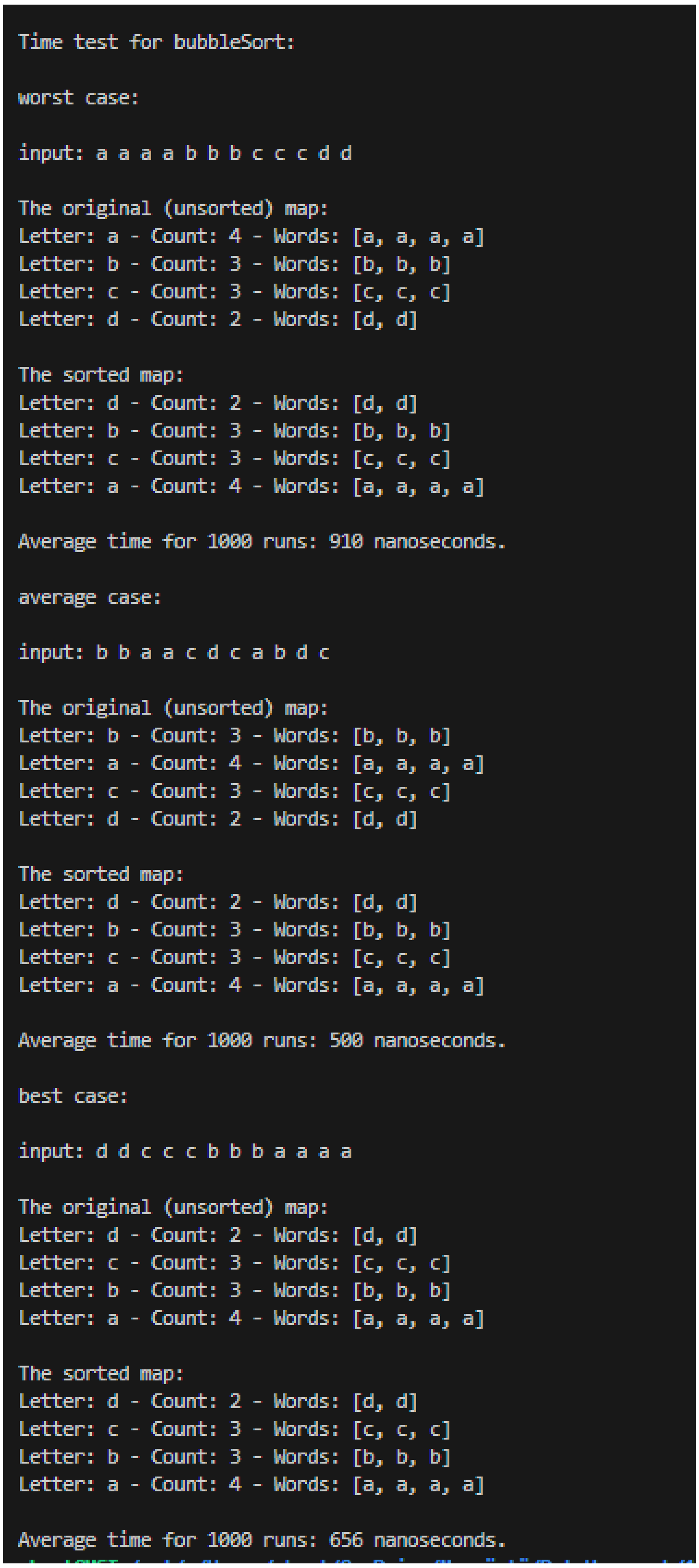
This algorithm works by repeatedly swapping the adjacent elements if they are in the wrong order.

Best-case scenario: The best-case scenario for Bubble Sort is when the input is already sorted. In this case, the algorithm only needs to go through the array once, making no swaps. If we use a flag to indicate whether a swap has happened in each pass, we can achieve a time complexity of O(n) in the best-case scenario.

Average-case scenario: The average-case scenario is typically a randomly ordered array. In this case, Bubble Sort performs at a time complexity of O(n^2), which is inefficient for larger arrays.

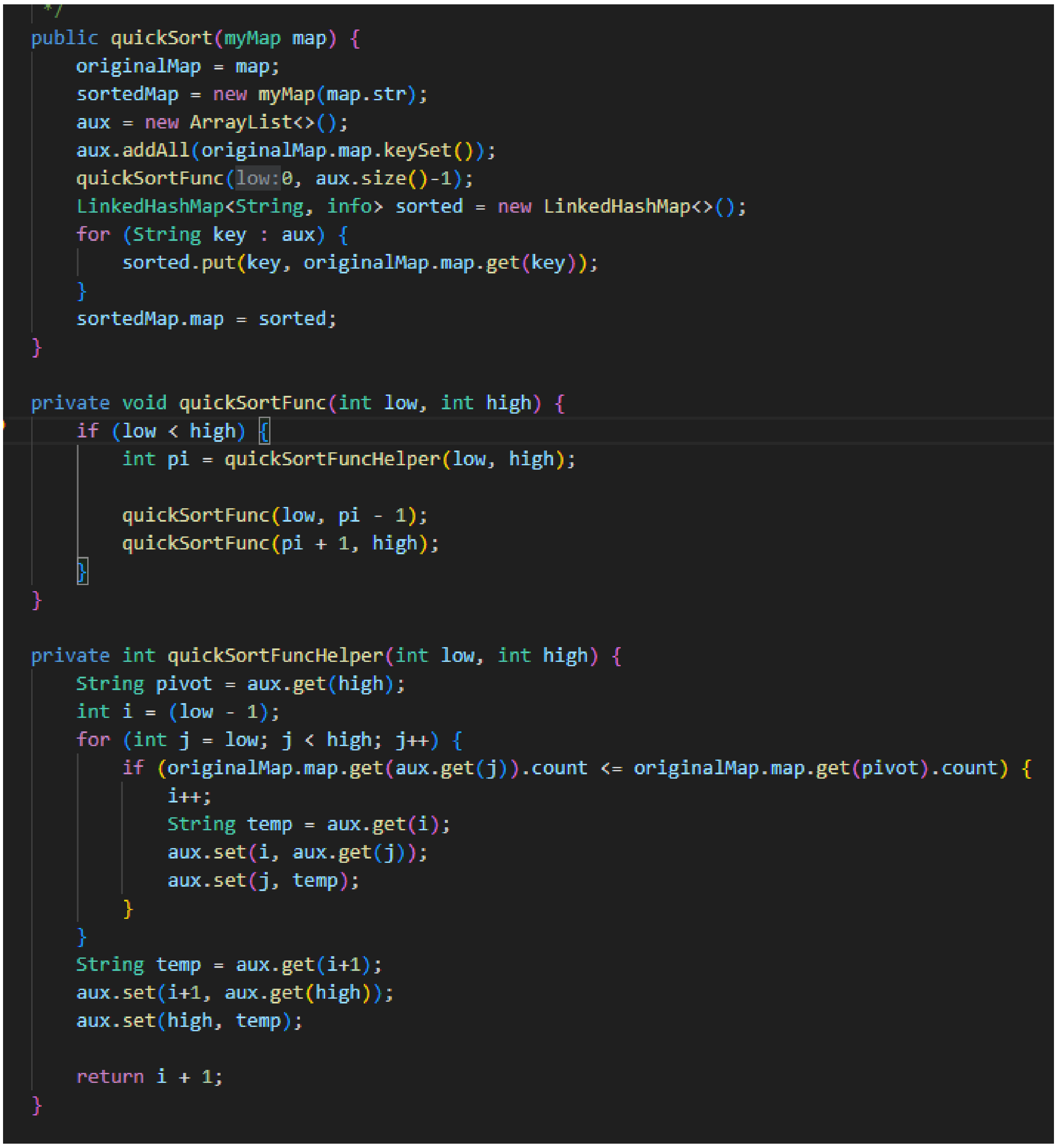
Worst-case scenario: The worst-case scenario for Bubble Sort occurs when the array is sorted in reverse order. In this case, the algorithm needs to make the maximum number of swaps, leading to a time complexity of O(n^2).

Like Insertion Sort and Merge Sort, Bubble Sort is a stable sorting algorithm. It maintains the relative order of equal elements, which is an important property in some applications. However, Bubble Sort is generally less efficient than other O(n^2) sorting algorithms like Insertion Sort and Selection Sort for larger arrays, because it has to go through the entire array to detect if it's sorted.

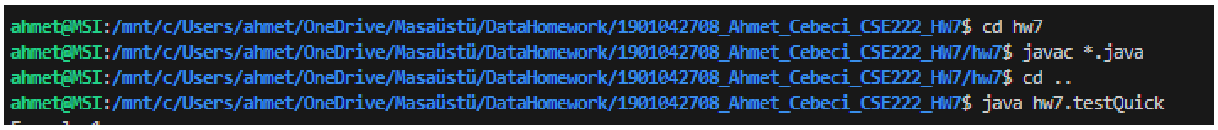


```
Time test for bubbleSort:

worst case:

input: a a a a b b b c c c d d

The original (unsorted) map:
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 910 nanoseconds.

average case:

input: b b a a c d c a b d c

The original (unsorted) map:
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 500 nanoseconds.

best case:

input: d d c c c b b b a a a a

The original (unsorted) map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 656 nanoseconds.
```

5 – Quick Sort



```java
    */
    public quickSort(myMap map) {
        originalMap = map;
        sortedMap = new myMap(map.str);
        aux = new ArrayList<>();
        aux.addAll(originalMap.map.keySet());
        quickSortFunc(low:0, aux.size()-1);
        LinkedHashMap<String, info> sorted = new LinkedHashMap<>();
        for (String key : aux) {
            sorted.put(key, originalMap.map.get(key));
        }
        sortedMap.map = sorted;
    }

    private void quickSortFunc(int low, int high) {
        if (low < high) {
            int pi = quickSortFuncHelper(low, high);

            quickSortFunc(low, pi - 1);
            quickSortFunc(pi + 1, high);
        }
    }

    private int quickSortFuncHelper(int low, int high) {
        String pivot = aux.get(high);
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (originalMap.map.get(aux.get(j)).count <= originalMap.map.get(pivot).count) {
                i++;
                String temp = aux.get(i);
                aux.set(i, aux.get(j));
                aux.set(j, temp);
            }
        }
        String temp = aux.get(i+1);
        aux.set(i+1, aux.get(high));
        aux.set(high, temp);

        return i + 1;
    }
```



```
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ cd hw7
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ javac *.java
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7/hw7$ cd ..
ahmet@MSI:/mnt/c/Users/ahmet/OneDrive/Masaüstü/DataHomework/1901042708_Ahmet_Cebeci_CSE222_HW7$ java hw7.testQuick
```

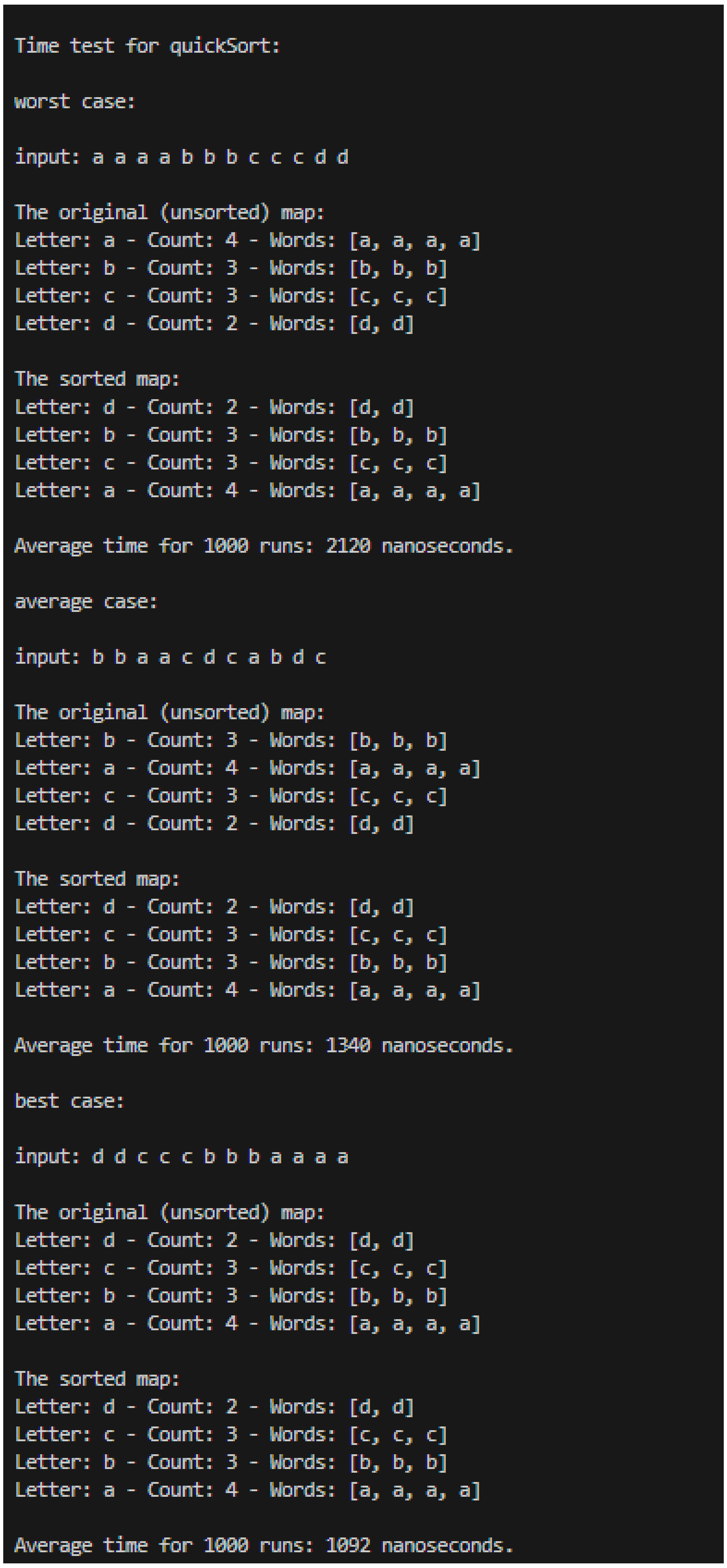
This is a divide-and-conquer algorithm that works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

Best-case scenario: The best-case scenario for Quick Sort occurs when the pivot always splits the array into two nearly equal halves. A balanced partition leads to a time complexity of O(n log n). This is achieved when the pivot chosen is the median of the array elements.

Average-case scenario: For most inputs, Quick Sort performs close to its best case with a time complexity of O(n log n). This makes it one of the fastest algorithms for sorting large and complex arrays in the average case.

Worst-case scenario: The worst-case scenario for Quick Sort occurs when the smallest or largest element is always chosen as the pivot (for example, if the array is already sorted or sorted in reverse order). This leads to unbalanced partitions and gives a time complexity of O(n^2). However, this scenario is unlikely if the pivot is chosen randomly.

Quick Sort is not a stable sorting algorithm, meaning the relative order of equal sort items is not preserved. This is primarily due to the process of partitioning the elements. If stability is necessary for your use case, you might need to choose a different sorting algorithm, such as Merge Sort or Insertion Sort. Also, Quick Sort performs well on larger datasets and has a smaller constant factor than other O(n log n) sorting algorithms, making it a popular choice for performance-critical applications.



```
Time test for quickSort:

worst case:

input: a a a a b b b c c c d d

The original (unsorted) map:
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 2120 nanoseconds.

average case:

input: b b a a c d c a b d c

The original (unsorted) map:
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: d - Count: 2 - Words: [d, d]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 1340 nanoseconds.

best case:

input: d d c c c b b b a a a a

The original (unsorted) map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

The sorted map:
Letter: d - Count: 2 - Words: [d, d]
Letter: c - Count: 3 - Words: [c, c, c]
Letter: b - Count: 3 - Words: [b, b, b]
Letter: a - Count: 4 - Words: [a, a, a, a]

Average time for 1000 runs: 1092 nanoseconds.
```

During our tests, we noticed something different with the average test results for Selection Sort and Quick Sort. While most methods sorted items in the order of 'd -> b -> c -> a', these two methods sorted them differently, in the order of 'd -> c -> b -> a'. This shows that Selection Sort and Quick Sort work a bit differently on average test cases. We didn't see any other changes when we made the sorted map.

In our test results, we noticed a unique pattern for Selection Sort and Quick Sort. In these two sorting algorithms, the sorted maps for the best and average cases are identical. In contrast, for the other sorting algorithms, the sorted maps for the average and worst cases turn out to be the same.

Based on our tests, the Insertion Sort algorithm was the fastest sorting algorithm. The speed of sorting algorithms can depend greatly on the characteristics and size of the data.

Insertion Sort is particularly effective for small or nearly sorted data sets. However, as the data set grows and becomes more complex, other sorting algorithms, especially more complex divide-and-conquer algorithms like Quick Sort or Merge Sort, are often faster. These types of algorithms work by breaking down larger data sets into smaller pieces, sorting those pieces, and then combining them.

The fact that we found Insertion Sort to be the fastest in our tests may indicate that the size and characteristics of our test cases were well-suited to this algorithm's strengths.

In summary, the choice of sorting algorithm greatly depends on the specific requirements of the task. Merge Sort and Quick Sort are often preferred for larger data sets due to their logarithmic time complexity. However, if stability is a concern, Merge Sort would be a better choice. For smaller or nearly sorted data sets, Insertion Sort could be a viable option. Bubble Sort and Selection Sort, while having simple implementations, are generally not efficient for large, random data sets.