GTU Department of Computer Engineering

CSE 222/505 – Spring 2023

Homework 7 Report

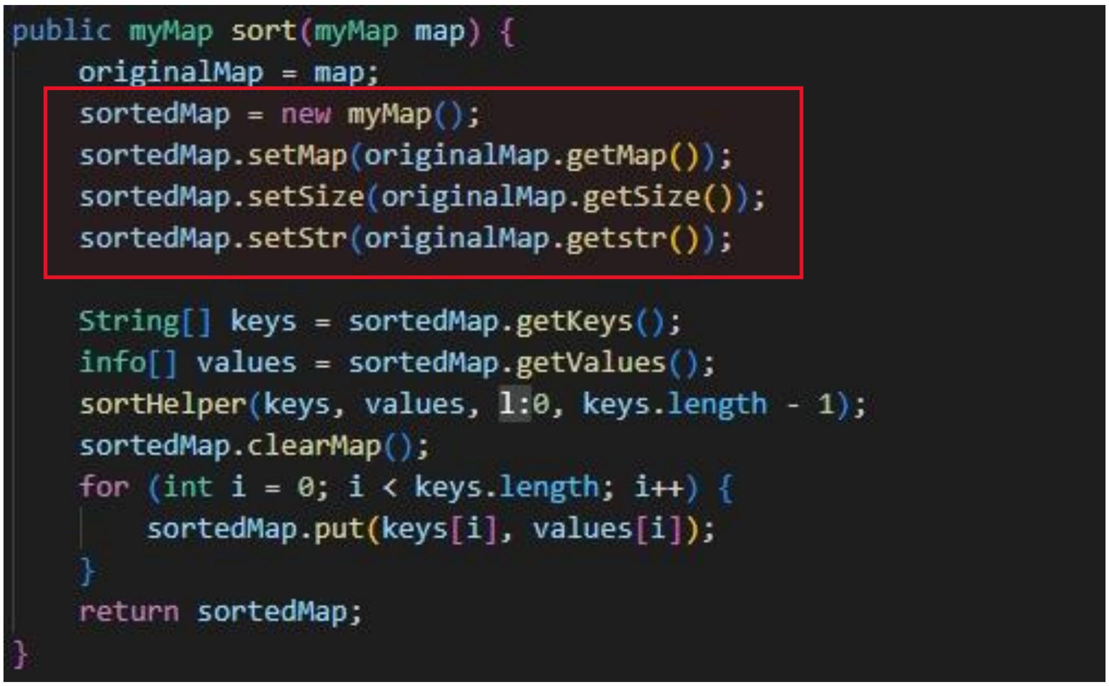
Barış Batuhan Bolat

210104004029

Time Complexity Analysis(Part2-a)

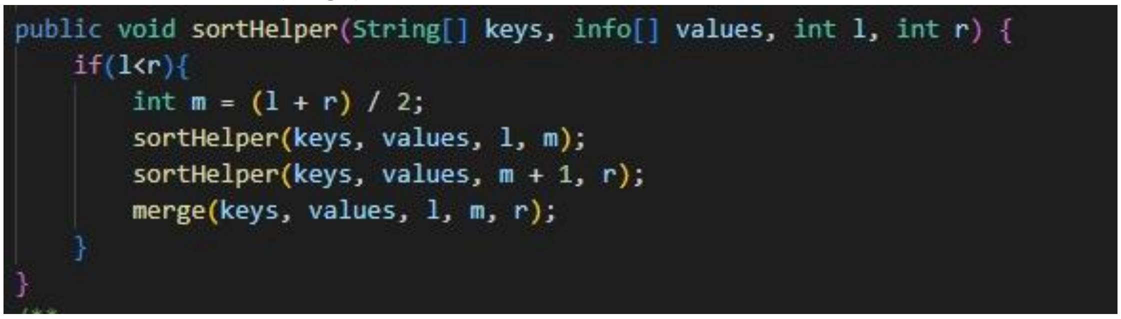
1. mergeSort

- This method initializes variables and assigns values from the original map to the sorted map. These operations and getter and setter methods (getMap(), getSize(), getStr(), setMap(), setSize(), setStr()) have constant time complexity. And also clear method that has O(n) time complexity. Sorthelper method'S time complexity is O(nlogn)(Explained below) . In the and of the method reconstruct the sorted map with the sorted keys and values using the put() method in a loop. The put() method has a constant time complexity. So overall time complexity is O(nlogn).
- In the HW6, I changed an operation that increases the time complexity in the highlighted section as below.



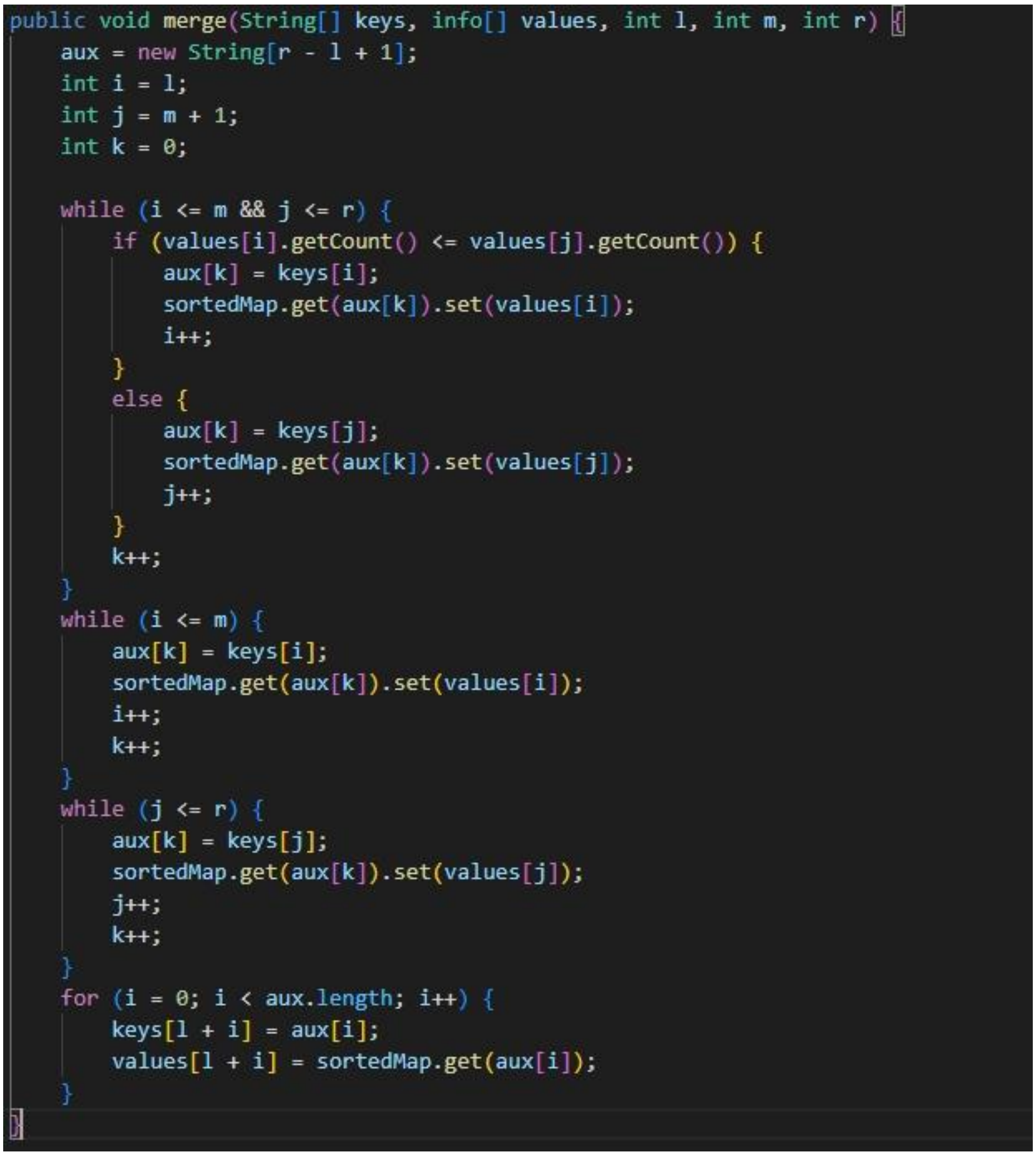
```
public myMap sort(myMap map) {
    originalMap = map;
    sortedMap = new myMap();
    sortedMap.setMap(originalMap.getMap());
    sortedMap.setSize(originalMap.getSize());
    sortedMap.setStr(originalMap.getstr());

    String[] keys = sortedMap.getKeys();
    info[] values = sortedMap.getValues();
    sortHelper(keys, values, l:0, keys.length - 1);
    sortedMap.clearMap();
    for (int i = 0; i < keys.length; i++) {
        sortedMap.put(keys[i], values[i]);
    }
    return sortedMap;
}
```

- This method makes a recursive call logn times. And each of them has a merge method that has O(n) time complexity(Explained below). So overall time complexity of this method is O(nlogn).



```
public void sortHelper(String[] keys, info[] values, int l, int r) {
    if(l<r){
        int m = (l + r) / 2;
        sortHelper(keys, values, l, m);
        sortHelper(keys, values, m + 1, r);
        merge(keys, values, l, m, r);
    }
}
```

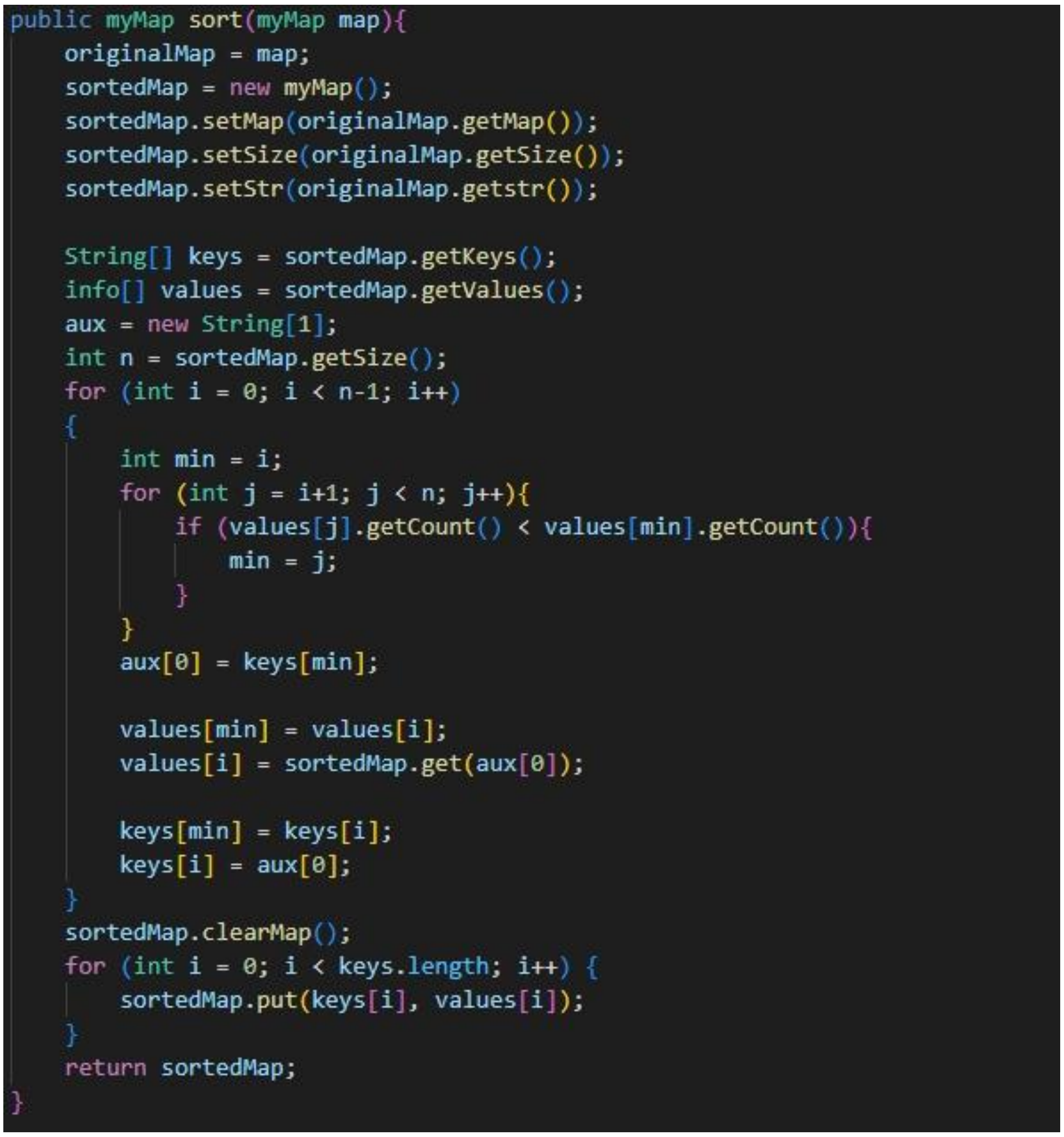
- The get and set operations on the sortedMap have a constant time complexity. The merging loop iterates through the elements in the ranges from l to m and from m + 1 to r, where m is the middle index. The loop compares the values of the elements and assigns them to the aux array and the sortedMap. The loop executes a total of r - l + 1 times. This parts time complexity is O(n). After the merging loop, there are two additional loops that copy the elements from the aux array back to the keys and values arrays. These loops execute a total of aux.length times, which is equal to the number of elements being merged. This parts time complexity is calculated as O(n) to. So overall time complexity of this method is O(n).



```java
public void merge(String[] keys, info[] values, int l, int m, int r) {
    aux = new String[r - l + 1];
    int i = l;
    int j = m + 1;
    int k = 0;

    while (i <= m && j <= r) {
        if (values[i].getCount() <= values[j].getCount()) {
            aux[k] = keys[i];
            sortedMap.get(aux[k]).set(values[i]);
            i++;
        }
        else {
            aux[k] = keys[j];
            sortedMap.get(aux[k]).set(values[j]);
            j++;
        }
        k++;
    }
    while (i <= m) {
        aux[k] = keys[i];
        sortedMap.get(aux[k]).set(values[i]);
        i++;
        k++;
    }
    while (j <= r) {
        aux[k] = keys[j];
        sortedMap.get(aux[k]).set(values[j]);
        j++;
        k++;
    }
    for (i = 0; i < aux.length; i++) {
        keys[l + i] = aux[i];
        values[l + i] = sortedMap.get(aux[i]);
    }
}
```

2. selectionSort

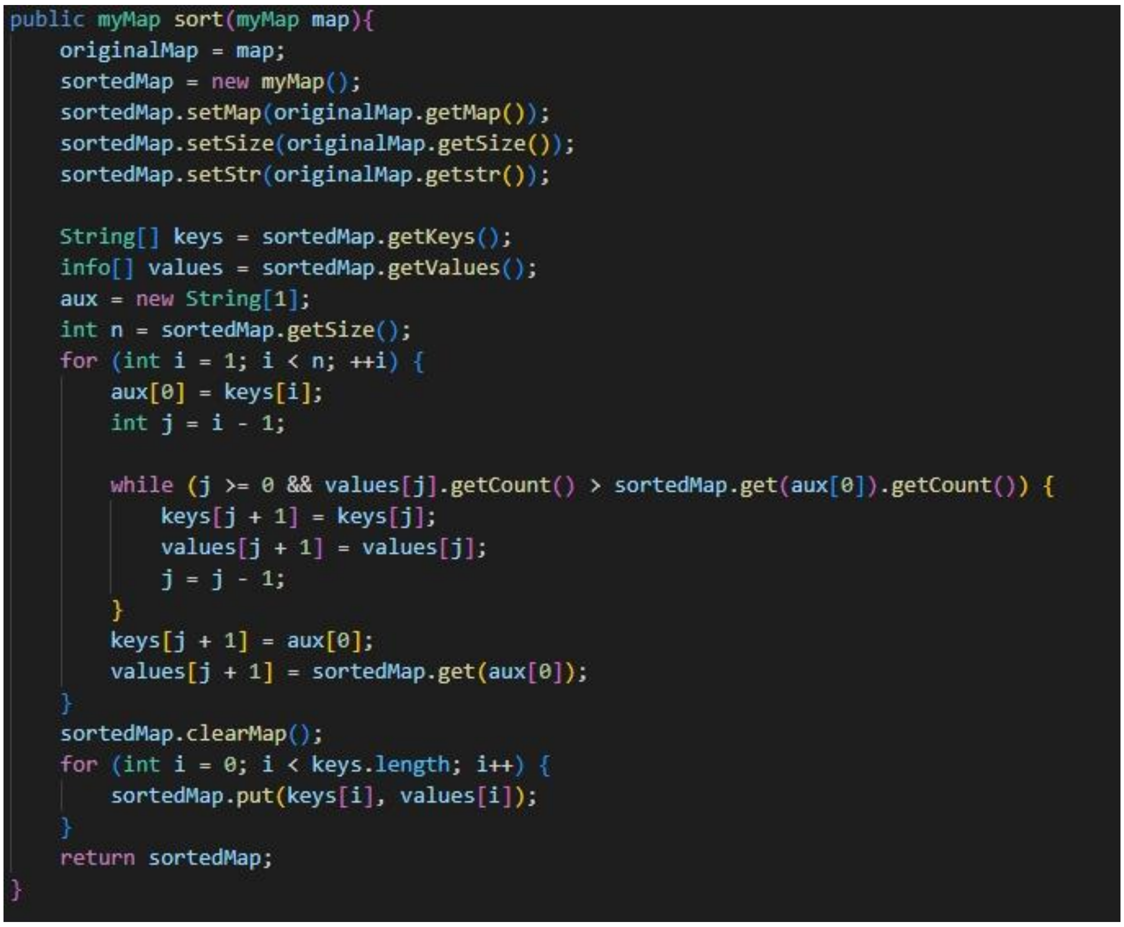
- This method initializes variables and assigns values from the original map to the sorted map. These operations and getter and setter methods (getMap(), getSize(), getStr(), setMap(), setSize(), setStr()) have constant time complexity. The outer loop iterates n-1 times. The inner loop iterates n-i-1 times in each iteration of the outer loop, where i is the current iteration index of the outer loop. It searches for the minimum count among the remaining values. And this method clears the sorted map using the clearMap() method, which has a time complexity of O(n). Then, reconstruct the sorted map with the sorted keys and values using the put() method in a loop. The put() method has a constant time complexity. So the overall time complexity is O(n^2).



```java
public myMap sort(myMap map){
    originalMap = map;
    sortedMap = new myMap();
    sortedMap.setMap(originalMap.getMap());
    sortedMap.setSize(originalMap.getSize());
    sortedMap.setStr(originalMap.getstr());

    String[] keys = sortedMap.getKeys();
    info[] values = sortedMap.getValues();
    aux = new String[1];
    int n = sortedMap.getSize();
    for (int i = 0; i < n-1; i++)
    {
        int min = i;
        for (int j = i+1; j < n; j++){
            if (values[j].getCount() < values[min].getCount()){
                min = j;
            }
        }
        aux[0] = keys[min];

        values[min] = values[i];
        values[i] = sortedMap.get(aux[0]);

        keys[min] = keys[i];
        keys[i] = aux[0];
    }
    sortedMap.clearMap();
    for (int i = 0; i < keys.length; i++) {
        sortedMap.put(keys[i], values[i]);
    }
    return sortedMap;
}
```

3. insertionSort

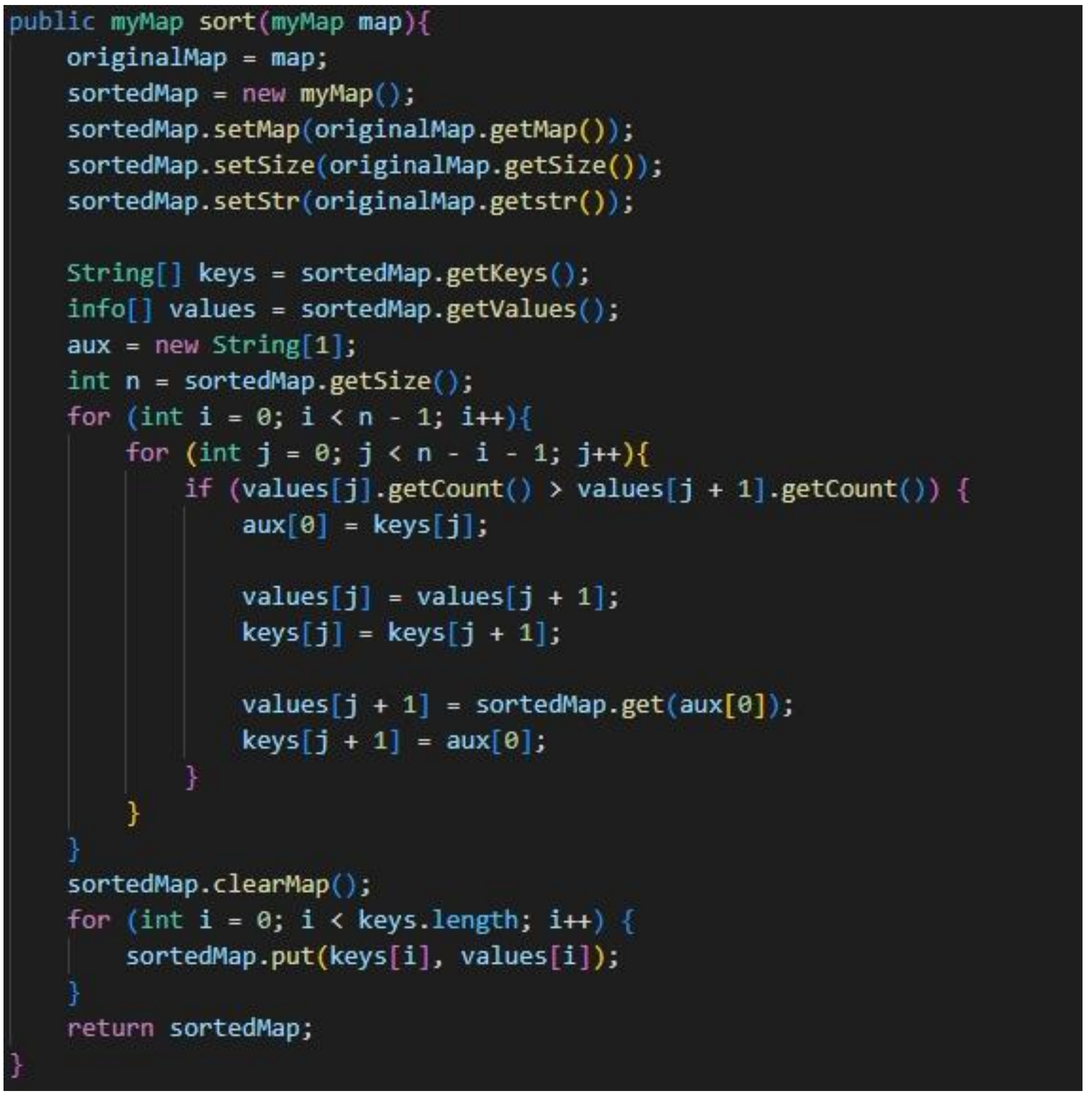
- This method initializes variables and assigns values from the original map to the sorted map. These operations and getter and setter methods (getMap(), getSize(), getStr(), setMap(), setSize(), setStr()) have constant time complexity. The outer loop iterates from the second element to the last element (n-1 iterations). The inner loop moves elements to the right until finding the correct position for the current element. The inner loop iterates until reaching the beginning of the array or finding an element with a lower count. In the end this method clears the sorted map using the clearMap() method, which has a time complexity of O(n), where n is the number of entries in the map. Then reconstruct the sorted map with the sorted keys and values arrays using the put() method in a loop. The put() method has a constant time complexity. So the overall time complexity of this method is O(n^2) in the worst case scenario. However, in the best case scenario (when the array is already sorted), the time complexity reduces to O(n).



```
public myMap sort(myMap map){
    originalMap = map;
    sortedMap = new myMap();
    sortedMap.setMap(originalMap.getMap());
    sortedMap.setSize(originalMap.getSize());
    sortedMap.setStr(originalMap.getstr());

    String[] keys = sortedMap.getKeys();
    info[] values = sortedMap.getValues();
    aux = new String[1];
    int n = sortedMap.getSize();
    for (int i = 1; i < n; ++i) {
        aux[0] = keys[i];
        int j = i - 1;

        while (j >= 0 && values[j].getCount() > sortedMap.get(aux[0]).getCount()) {
            keys[j + 1] = keys[j];
            values[j + 1] = values[j];
            j = j - 1;
        }
        keys[j + 1] = aux[0];
        values[j + 1] = sortedMap.get(aux[0]);
    }
    sortedMap.clearMap();
    for (int i = 0; i < keys.length; i++) {
        sortedMap.put(keys[i], values[i]);
    }
    return sortedMap;
}
```

4. bubbleSort

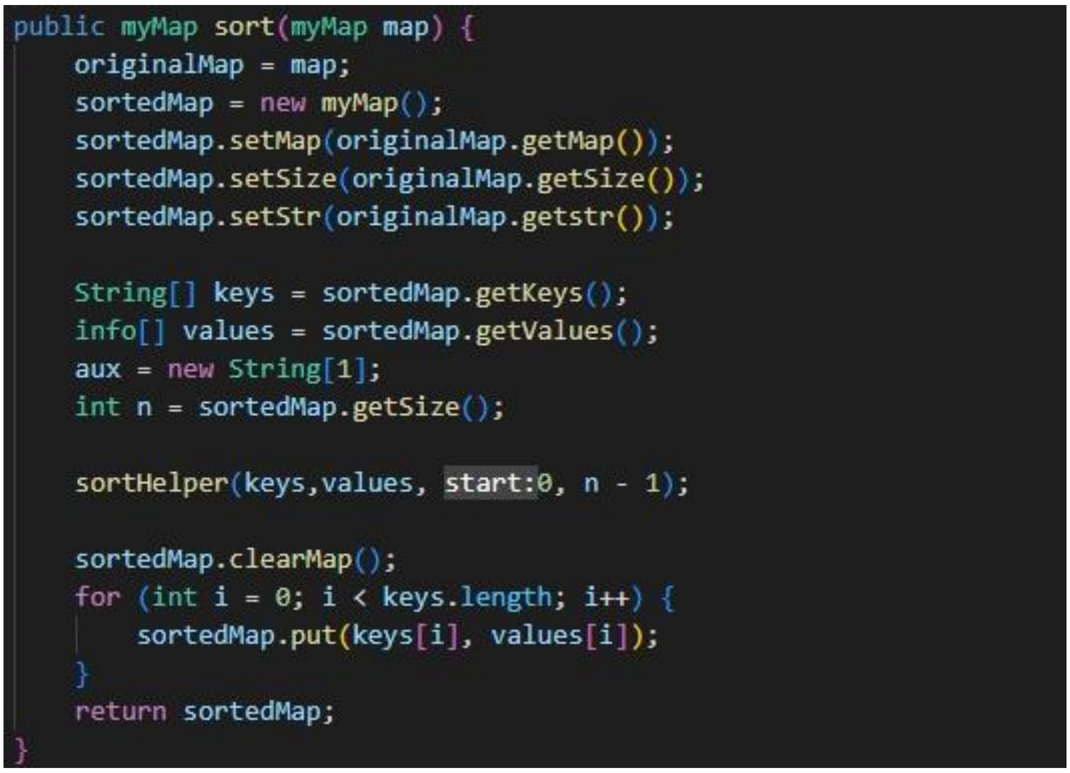
- This method initializes variables and assigns values from the original map to the sorted map. These operations and getter and setter methods (getMap(), getSize(), getStr(), setMap(), setSize(), setStr()) have constant time complexity. The outer loop iterates n-1 times (where n is the size of the map). The inner loop iterates from 0 to n-i-1 in each iteration of the outer loop, where i is the current iteration of the outer loop. It compares counts of values and swaps them if necessary to move the larger element towards the end of the array. This method clears the sorted map using the clearMap() method, which has a time complexity of O(n). Then reconstruct the sorted map with the sorted keys and values arrays using the put() method in a loop. The put() method has a constant time complexity. So the overall time complexity of the sort method can be approximated as O(n^2) in the worst and average case. In the best case (already sorted map), the time complexity becomes O(n).



```
public myMap sort(myMap map){
    originalMap = map;
    sortedMap = new myMap();
    sortedMap.setMap(originalMap.getMap());
    sortedMap.setSize(originalMap.getSize());
    sortedMap.setStr(originalMap.getstr());

    String[] keys = sortedMap.getKeys();
    info[] values = sortedMap.getValues();
    aux = new String[1];
    int n = sortedMap.getSize();
    for (int i = 0; i < n - 1; i++){
        for (int j = 0; j < n - i - 1; j++){
            if (values[j].getCount() > values[j + 1].getCount()) {
                aux[0] = keys[j];

                values[j] = values[j + 1];
                keys[j] = keys[j + 1];

                values[j + 1] = sortedMap.get(aux[0]);
                keys[j + 1] = aux[0];
            }
        }
    }
    sortedMap.clearMap();
    for (int i = 0; i < keys.length; i++) {
        sortedMap.put(keys[i], values[i]);
    }
    return sortedMap;
}
```

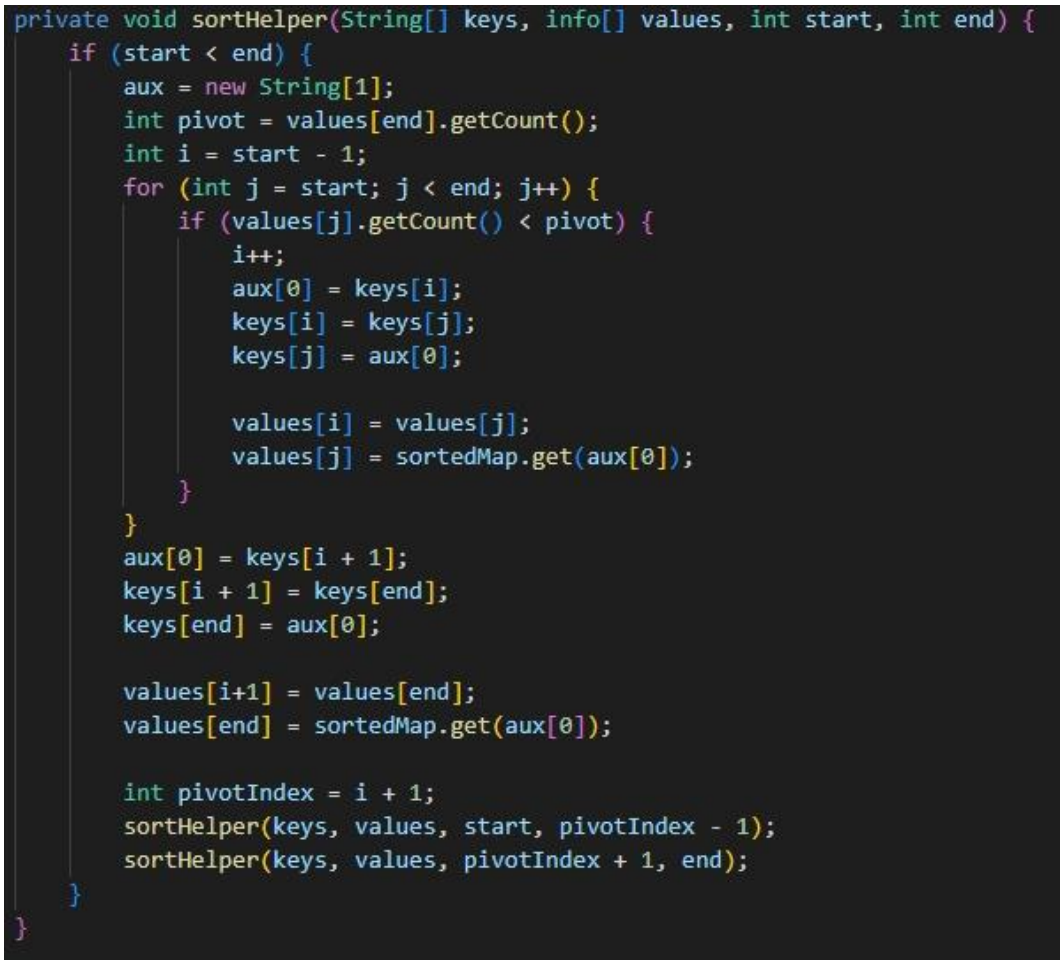
5. quickSort

- This method initializes variables and assigns values from the original map to the sorted map. These operations and getter and setter methods (getMap(), getSize(), getStr(), setMap(), setSize(), setStr()) have constant time complexity. sorthelper method has different time complexities for each case. Best case and average case has O(nlogn) and worst case has O(n^2)(Explained below). In the last part of the method clears the sorted map using the clearMap() method, which has a time complexity of O(n). Then reconstruct the sorted map with the sorted keys and values arrays using the put() method in a loop. The put() method has a constant time complexity. So the overall time complexity of the this method can be approximated as O(nlogn) in the average and best case scenarios. However, in the worst case scenario, the time complexity can be O(n^2).



```java
public myMap sort(myMap map) {
    originalMap = map;
    sortedMap = new myMap();
    sortedMap.setMap(originalMap.getMap());
    sortedMap.setSize(originalMap.getSize());
    sortedMap.setStr(originalMap.getstr());

    String[] keys = sortedMap.getKeys();
    info[] values = sortedMap.getValues();
    aux = new String[1];
    int n = sortedMap.getSize();

    sortHelper(keys,values, start:0, n - 1);

    sortedMap.clearMap();
    for (int i = 0; i < keys.length; i++) {
        sortedMap.put(keys[i], values[i]);
    }
    return sortedMap;
}
```

- This method has getter and setter methods (getMap(), getSize(), getStr(), setMap(), setSize(), setStr()) that have constant time complexity.
  This method can be analyzed in two parts:
- Partitioning Part : This step takes O(n) time, where n is the number of elements in the sub-array being partitioned.
- Recursive Part : The recursive calls to sort the sub-arrays have a total time complexity of O(log n) due to the binary nature of the partitioning.
- So the total time complexity of this method is O(nlogn) in the average and best case scenarios. However, in the worst case scenario (when the array is already sorted or reverse sorted), the partitioning may result in highly unbalanced sub-arrays, leading to a time complexity of O(n^2).



```java
private void sortHelper(String[] keys, info[] values, int start, int end) {
    if (start < end) {
        aux = new String[1];
        int pivot = values[end].getCount();
        int i = start - 1;
        for (int j = start; j < end; j++) {
            if (values[j].getCount() < pivot) {
                i++;
                aux[0] = keys[i];
                keys[i] = keys[j];
                keys[j] = aux[0];

                values[i] = values[j];
                values[j] = sortedMap.get(aux[0]);
            }
        }
        aux[0] = keys[i + 1];
        keys[i + 1] = keys[end];
        keys[end] = aux[0];

        values[i+1] = values[end];
        values[end] = sortedMap.get(aux[0]);

        int pivotIndex = i + 1;
        sortHelper(keys, values, start, pivotIndex - 1);
        sortHelper(keys, values, pivotIndex + 1, end);
    }
}
```

Time Complexity Table

	mergeSort	*selectionSort*	*insertionSort*	*bubbleSort*	*quickSort*
Best Case	O(nlogn)	O(n^2)	O(n)	O(n)	O(nlogn)
Average Case	O(nlogn)	O(n^2)	O(n^2)	O(n^2)	O(nlogn)
Worst Case	O(nlogn)	O(n^2)	O(n^2)	O(n^2)	O(n^2)

Running Time(Nanoseconds)(Part2-b)

Inputs

Best Case : a bb ccc dddd eeeee
Average Case : bb eeeee a dddd ccc
Worst Case : eeeee dddd ccc bb a

	mergeSort	*selectionSort*	*insertionSort*	*bubbleSort*	*quickSort*
Best Case	12936	7753	8576	6759	9747
Average Case	11504	9285	9181	10454	10494
Worst Case	14131	10299	11682	12831	13106

Comparison(Part2-c)

- First, let's briefly summarize the sorting algorithms.

Merge Sort

- Time Complexity: O(nlog n) in all cases (best, worst, and average).
- It is generally faster than the quadratic-time sorting algorithms (selection sort, insertion sort, bubble sort) for worst case.

Selection Sort

- Time Complexity: O(n^2) in all cases (best, worst, and average).
- Selection sort has a simple but is generally slower than other sorting algorithms for large datasets.

Insertion Sort

- Time Complexity: O(n) in the best case and O(n^2) for worst and average cases
- Insertion sort performs well on small or nearly sorted datasets due to its efficient best-case time complexity. However, it becomes inefficient for worst cases as it requires shifting elements multiple times.

Bubble Sort

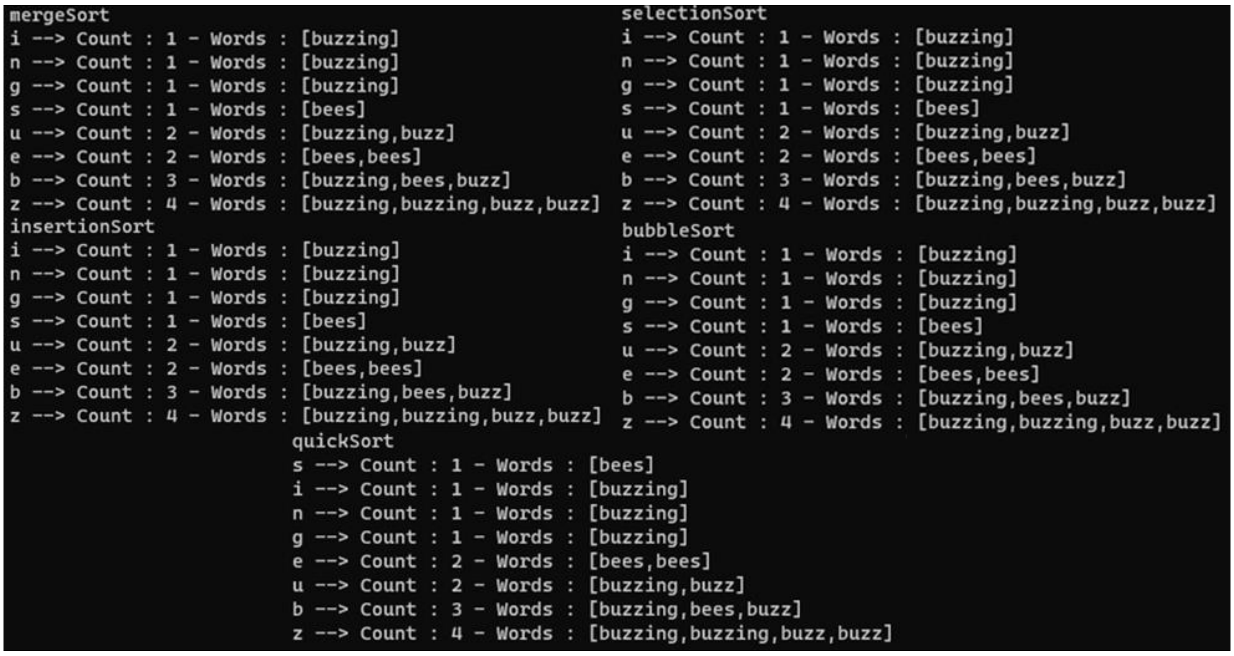
- Time Complexity: O(n^2) in the worst and average cases, O(n) in the best case.
- Bubble sort is simple but generally slower than other sorting algorithms.

Quick Sort

- Time Complexity: O(nlogn) in the average and best cases, O(n^2) in the worst case.
- Quick sort is generally faster than the quadratic-time sorting algorithms for average and best cases. However, in the worst case it can have a time complexity similar to selection sort or bubble sort.

- As a result, sorting algorithm depends on the input. Merge sort and quick sort are generally more efficient for worst cases, insertion sort is suitable for best cases, while selection sort and bubble sort are less efficient but have simple implementations.
- In this homework best case is already sorted map,average case is nearly sorted or unsorted map and worst case is reverse sorted map.

Part2-d

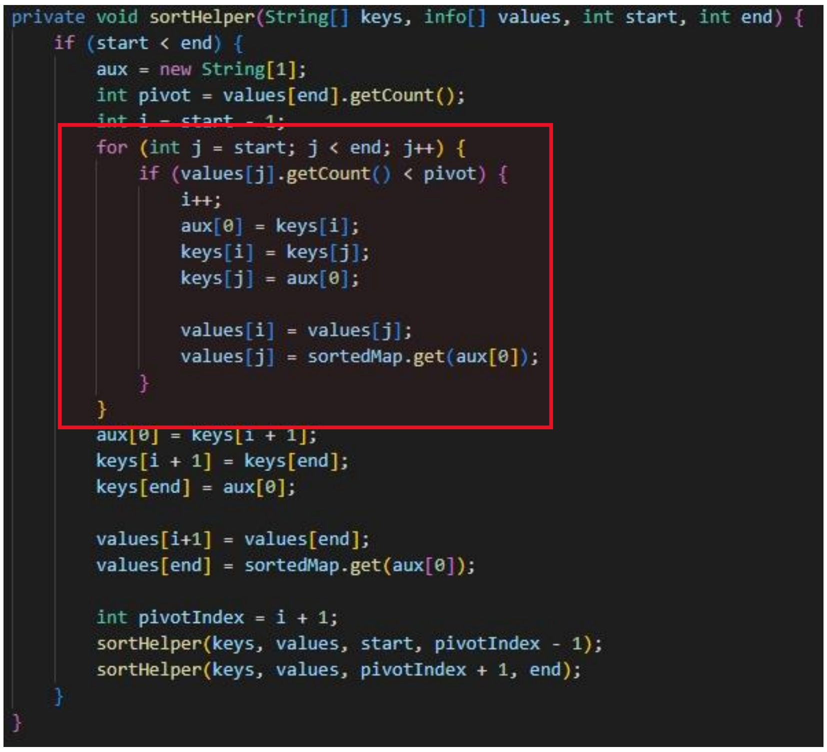


```
mergeSort
i --> Count : 1 - Words : [buzzing]
n --> Count : 1 - Words : [buzzing]
g --> Count : 1 - Words : [buzzing]
s --> Count : 1 - Words : [bees]
u --> Count : 2 - Words : [buzzing,buzz]
e --> Count : 2 - Words : [bees,bees]
b --> Count : 3 - Words : [buzzing,bees,buzz]
z --> Count : 4 - Words : [buzzing,buzzing,buzz,buzz]
insertionSort
i --> Count : 1 - Words : [buzzing]
n --> Count : 1 - Words : [buzzing]
g --> Count : 1 - Words : [buzzing]
s --> Count : 1 - Words : [bees]
u --> Count : 2 - Words : [buzzing,buzz]
e --> Count : 2 - Words : [bees,bees]
b --> Count : 3 - Words : [buzzing,bees,buzz]
z --> Count : 4 - Words : [buzzing,buzzing,buzz,buzz]
```

```
selectionSort
i --> Count : 1 - Words : [buzzing]
n --> Count : 1 - Words : [buzzing]
g --> Count : 1 - Words : [buzzing]
s --> Count : 1 - Words : [bees]
u --> Count : 2 - Words : [buzzing,buzz]
e --> Count : 2 - Words : [bees,bees]
b --> Count : 3 - Words : [buzzing,bees,buzz]
z --> Count : 4 - Words : [buzzing,buzzing,buzz,buzz]
bubbleSort
i --> Count : 1 - Words : [buzzing]
n --> Count : 1 - Words : [buzzing]
g --> Count : 1 - Words : [buzzing]
s --> Count : 1 - Words : [bees]
u --> Count : 2 - Words : [buzzing,buzz]
e --> Count : 2 - Words : [bees,bees]
b --> Count : 3 - Words : [buzzing,bees,buzz]
z --> Count : 4 - Words : [buzzing,buzzing,buzz,buzz]
```

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

- As we see above all sorting algorithms except quickSort keep input ordering. The main reason why quickSort does not keep input ordering its partitioning step. Elements are moved around throughout the partitioning process according on how they relate to the pivot element. Larger elements are positioned to the right, whereas smaller elements are positioned to the left. However, the partitioning part does not maintain the relative order of elements with the same value.



```java
private void sortHelper(String[] keys, info[] values, int start, int end) {
    if (start < end) {
        aux = new String[1];
        int pivot = values[end].getCount();
        int i = start - 1;
        for (int j = start; j < end; j++) {
            if (values[j].getCount() < pivot) {
                i++;
                aux[0] = keys[i];
                keys[i] = keys[j];
                keys[j] = aux[0];

                values[i] = values[j];
                values[j] = sortedMap.get(aux[0]);
            }
        }
        aux[0] = keys[i + 1];
        keys[i + 1] = keys[end];
        keys[end] = aux[0];

        values[i+1] = values[end];
        values[end] = sortedMap.get(aux[0]);

        int pivotIndex = i + 1;
        sortHelper(keys, values, start, pivotIndex - 1);
        sortHelper(keys, values, pivotIndex + 1, end);
    }
}
```

- Marked part is the reason for this problem.