Gebze Technical University
CSE222 – HW7 Report

Name: Emre Oytun
Student ID: 200104004099

a) Analysis of the Sorting Algorithms:

Merge Sort:

Best-case: O(nlogn)
Average-case: O(nlogn)
Worst-case: O(nlogn)
Merge sort does not depend on the order of the elements like if the given elements are nearly sorted or not; or if they are same etc. It always splits the current sub-array into two sub-arrays and this results in "logn" sort method calls and O(logn) complexity. In each time after the calls are returned for the two sub-arrays, it merges the resulting arrays in O(n) time. So the total complexity is O(nlogn) for all cases.

Selection Sort:

Best-case: O(n^2)
Average-case: O(n^2)
Worst-case: O(n^2)
Selection sort does not depend on the order of the elements like if the given elements are nearly sorted or not; or if they are same etc. It iterates through the array "n" times, and each time it finds the minimum element's index by iterating through the remaining elements and swap the first element of the remaining array with the minimum element. So there are total: n*(n+1)/2 processes, and it results in O(n^2) for all cases.

Insertion Sort:

Best-case: O(n)
Average-case: O(n^2)
Worst-case: O(n^2)
Insertion sort depends on the order of the elements. If the elements are sorted already then it runs in O(n) complexity. The left side of the array is considered as sorted, and the next element from the rest of the array is inserted into this sorted side until there are no elements unsorted.
Firstly, the first element is considered as the only element in the sorted array and the next elements are inserted into the sorted array each time. Each time, the next element is compared with the previous one, and they are swapped if it is needed and the process

continues until there is no need to swap. For n elements, the outer loop works “n-1” times and it is fixed but the total process inside the inner loop changes from element to element.
For the best-case, when all elements are sorted; the inner loop works only “1” time for each of the elements to compare with the previous one. Therefore, there are total “n-1” processes and it results in O(n) as O(n) for comparision and O(1) for swap.
For the worst-case, when elements are sorted in reverse order; the processes for each elements are as 1 + 2 + … + n-1, so it results in O(n^2) as O(n^2) for comparision and O(n^2) for swaps.
For the average-case, when elements are not sorted or nearly-sorted but elements are not sorted in reverse order in the same time; it works as in the worst-case scenario except it could require less comparisions for some of the elements but time complexity results in O(n^2).

Bubble Sort:

Best-case: O(n)
Worst-case: O(n^2)
Average-case: O(n^2)
Bubble sort algorithm sorts the elements like bubbles, in each time it iterates through all elements except the last one, and checks if the current element and the next element should be swapped or not; if it does then it swaps the elements, otherwise it does not swap them. This process is done “n” times as long as there is at least one exchange that is made.
For the best-case, when all elements are sorted; it iterates through all the elements once and since there is no exchange, it stops after this iteration. Therefore, it takes O(n) time.
For the worst-case, when all elements are sorted in reverse order, the outer loop does not stop in the middle and runs “n” times while each time making “n-1” comparisions and swaps. Therefore, it results in O(n^2).
For the average-case, when all elements are not sorted, and there is at least one swap for most of the time; the outer loop works “n” times or nearly “n” times while each time making “n-1” comparisions and “n-1” or nearly “n-1” swaps. Therefore, it results in O(n^2).

Quick Sort:

Best-case: O(nlogn)
Worst-case: O(n^2)
Average-case: O(nlogn)
Quick sort algorithm splits the array into two sub-arrays each time by partitioning the current sub-array using a pivot. In my algorithm, the pivot is selected as the middle element each time, firstly the pivot element is swapped with the last element and the

new pivot index is initialized as the first index. Then the remaining elements are compared with the pivot element, if an element is smaller or equal than the pivot element then it is swapped by the new pivot index, and the new pivot index is incremented by one.
For the best-case, the current sub-array is split from the middle such that the pivot element is the middle element in sorting order each time like in the case when all the elements are sorted. This results in “logn” process/recursive calls, and in each process, the partitioning takes firstly n times, than n/2 times and so on. Therefore, the recursive calls/iterations take O(logn) and each call takes O(n) time. It results in O(nlogn).
For the worst-case, the current sub-array cannot be split into two sub-array because the pivot is always the smallest element and it results in the way that the array size is only decremented by one(pivot element) each time. So, it causes “n” recursive calls. Since each recursive call takes O(n) to partition the array, it takes O(n^2).
For the average-case, the current sub-array is split into two sub-arrays whose sizes are equal or nearly-equal with small differences. Therefore, the quicksort algorithm is able to split the current sub-array into two sub-arrays and so the recursive calls take O(logn). Since again each call takes O(n) time to partition, this results in O(nlogn).

b) Running Time of Sorting Algorithms:

Algorithms/Times	Best-case (ns)	Worst-case (ns)	Average-case (ns)
Merge Sort	26600	27000	25100
Selection Sort	73100	47500	53400
Insertion Sort	9700	71000	38100
Bubble Sort	10300	110300	92600
Quick Sort	23500	37700	17700

c) Comparision of Sorting Algorithms:

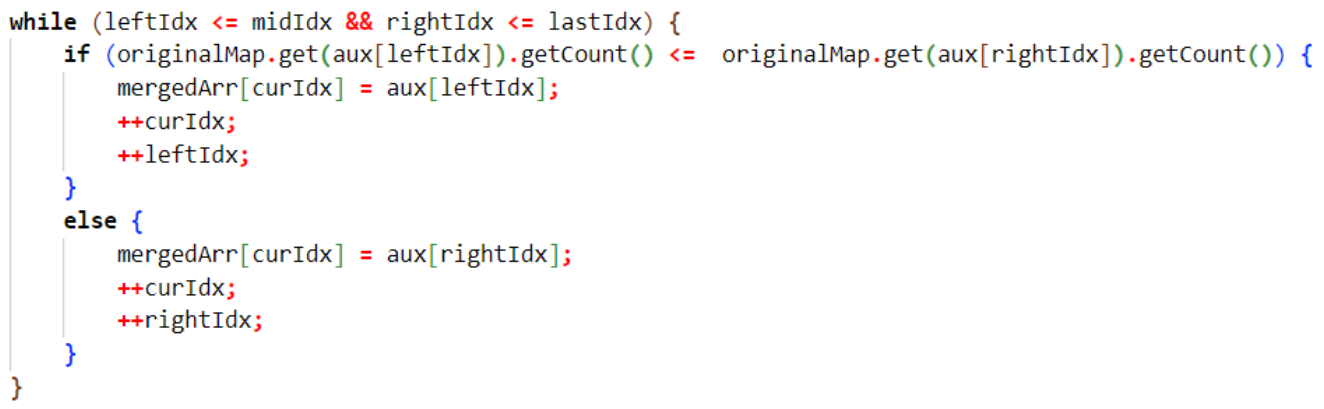
Best-case: In best-case, insertion sort and bubble sort algorithms are the fastest algorithms since their complexities is less than the others with the O(n) complexity and from the running time table it can be seen that insertion sort is faster then the bubble sort even though they are almost similar.

Worst-case: In worst-case, the merge sort is the fastest algorithm since its complexity is less than the others with O(nlogn) complexity while others’ complexity is O(n^2) and from the running time table it can be seen that the merge sort is faster than the others.

Average-case: In average-case, the quick sort and the merge sort algorithms are faster than the others since their complexities are less than the others wigh O(nlogn) complexity while others' complexity is quadratic O(n^2) and from the running time table it can be seen that the quick sort is faster than the merge sort also.

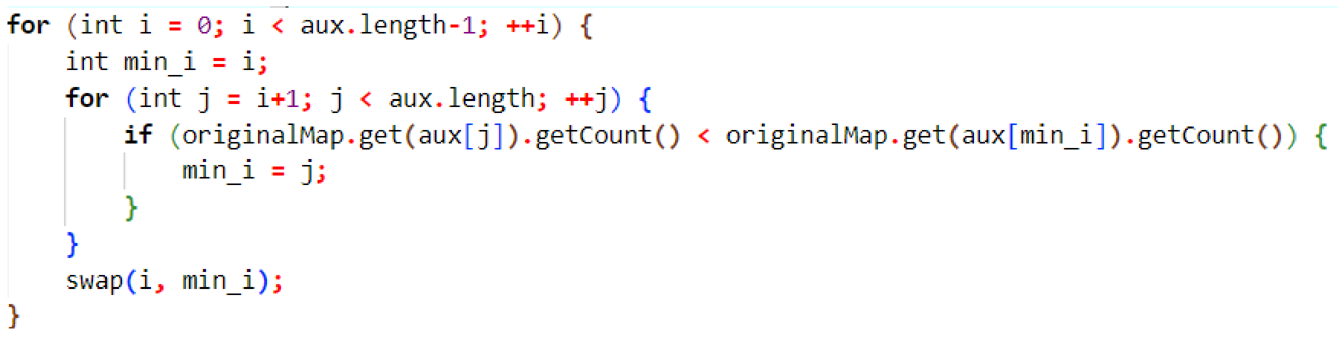
d) Analysis of Sorting Stability:

Merge Sort: Merge sort keeps the relative order of the same elements since it puts the element in the left side into the array before putting the element which in the right side as seen in the code snippet below from merge method.



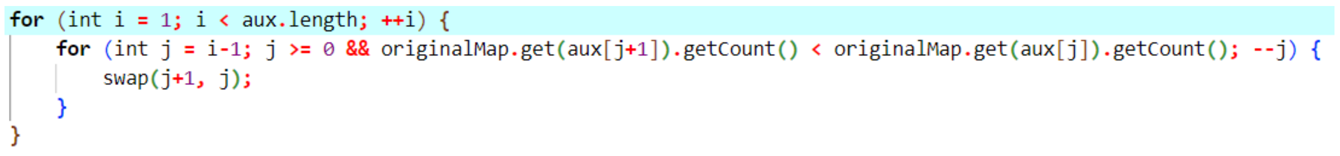
```
while (leftIdx <= midIdx && rightIdx <= lastIdx) {
    if (originalMap.get(aux[leftIdx]).getCount() <=  originalMap.get(aux[rightIdx]).getCount()) {
        mergedArr[curIdx] = aux[leftIdx];
        ++curIdx;
        ++leftIdx;
    }
    else {
        mergedArr[curIdx] = aux[rightIdx];
        ++curIdx;
        ++rightIdx;
    }
}
```

Selection Sort: Selection sort does not keep the relative order of the same elements since it swaps the minimum element and the first element of the current sub-array as seen in the code snippet below from the sort method.



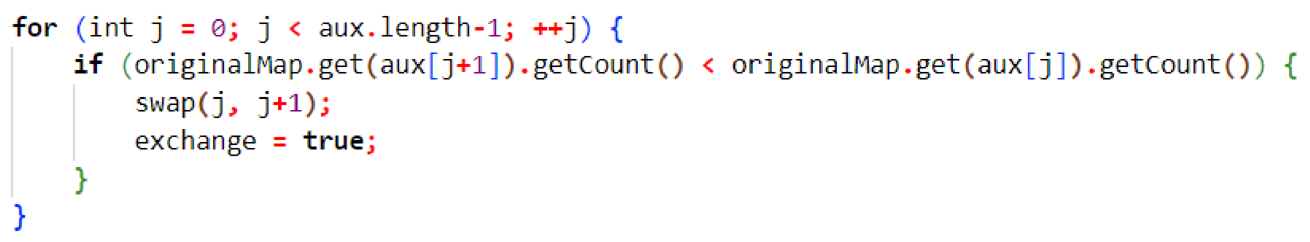
```
for (int i = 0; i < aux.length-1; ++i) {
    int min_i = i;
    for (int j = i+1; j < aux.length; ++j) {
        if (originalMap.get(aux[j]).getCount() < originalMap.get(aux[min_i]).getCount()) {
            min_i = j;
        }
    }
    swap(i, min_i);
}
```

Insertion Sort: Insertion sort keeps the relative order of the same elements since it stops to insert operation for the current element if the next element is not smaller than the current element as seen in the code snippet below.



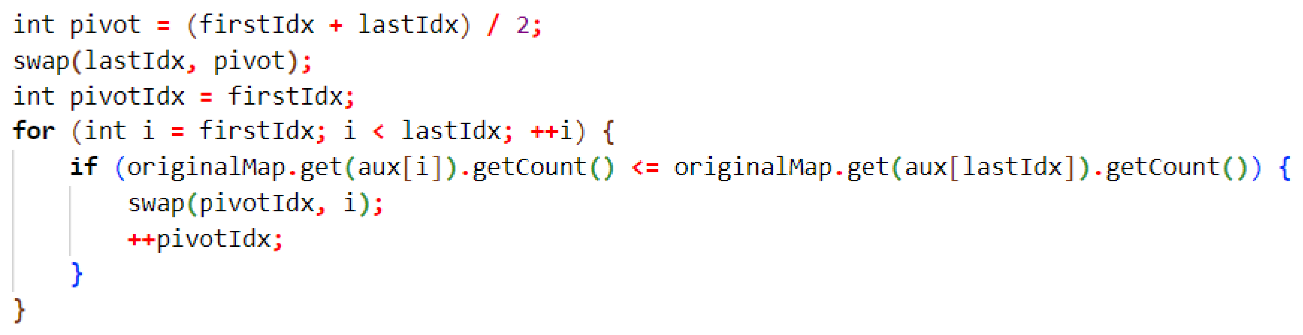
```
for (int i = 1; i < aux.length; ++i) {
    for (int j = i-1; j >= 0 && originalMap.get(aux[j+1]).getCount() < originalMap.get(aux[j]).getCount(); --j) {
        swap(j+1, j);
    }
}
```

Bubble Sort: Bubble sort keeps the relative order of the same elements since it does not swap the elements if the next element is not smaller than the current one as seen in the code snippet below.



```
for (int j = 0; j < aux.length-1; ++j) {
    if (originalMap.get(aux[j+1]).getCount() < originalMap.get(aux[j]).getCount()) {
        swap(j, j+1);
        exchange = true;
    }
}
```

Quick Sort: Quick sort does not keep the relative order of the same elements since it first puts the pivot at the end, and compares all the rest elements with the pivot. If an element is smaller or equal than the pivot, it swaps this element such that it will be before the new pivot index. This process can change the relative order of the same elements in the case that an element which is the same with the pivot element can be put before the pivot element while it is actually after the pivot element. The code snippet is given below.



```
int pivot = (firstIdx + lastIdx) / 2;
swap(lastIdx, pivot);
int pivotIdx = firstIdx;
for (int i = firstIdx; i < lastIdx; ++i) {
    if (originalMap.get(aux[i]).getCount() <= originalMap.get(aux[lastIdx]).getCount()) {
        swap(pivotIdx, i);
        ++pivotIdx;
    }
}
```

d) Explanation of the Testing in Main Class:

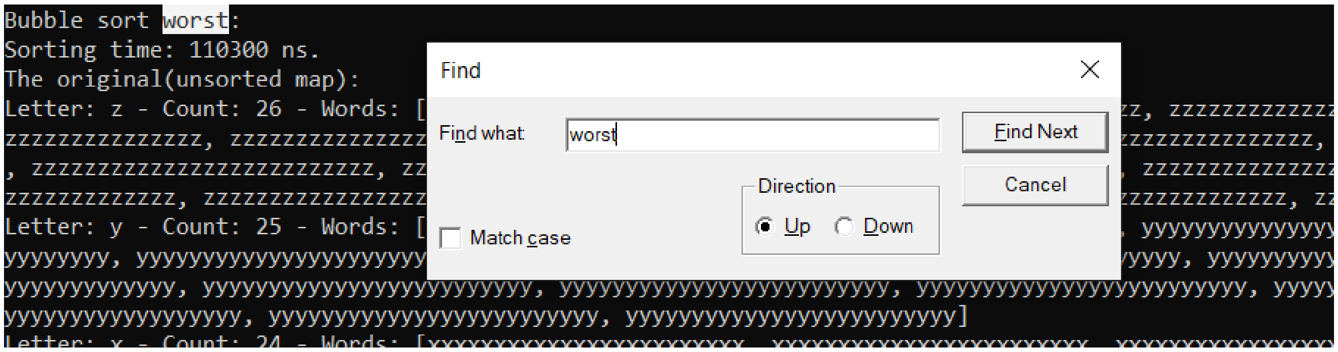
Main class tests all five sorting algorithms for the 3 scenarios as best, worst and average case. In order to increase consistency, I used max number of elements to create my map for all scenarios but this makes hard to examine the output of the program. You can search an output for a scenario of a sorting algorithm by searching like “Quick sort best” as you can see in the screenshot below.

Running Commands:

```
javac *.java
java Main
```

** Since the output is long, it is sometimes not seen fully from some ides. It’s recommended to observe the output by running the program from a terminal like cmd or linux terminal.

Screenshot to search for a scenario:



```
Bubble sort worst:
Sorting time: 110300 ns.
The original(unsorted map):
Letter: z - Count: 26 - Words: [
Letter: y - Count: 25 - Words: [
Letter: x - Count: 24 - Words: [
```