GTU Department of Computer Engineering
CSE 222 - Homework 7 - Report

Doğukan Taştan 1901042627

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

Merge Sort:

We continuously divide the sequence in half. This leads to logarithmic time complexity (log n). Because each time we halve the size of the array, we reduce the size of the array down to 1. In the concatenate step, when concatenating the array, we need n elements in each concatenation step. This leads to a time complexity of n. The loops actually iterate over two different sections during the concatenation process, so even though it looks like 2 while nested, the effect is n.
Best: O(n log n)
Average: O(n log n)
Worst: O(n log n)

Selection Sort :

Selection Sort checks all elements even in the best case (when the array is already sorted). This is the reason why the time complexity is expressed as n squared. In each iteration, a search is performed on all remaining elements from the current position, leading to a time complexity of n^2
Best: O(n^2)
Average: O(n^2)
Worst: O(n^2)

Insertion Sort :

In the best case, i.e. when the array is already sorted at the beginning, Insertion Sort will have a time complexity of O(n). This is because in this case, the inner loop (while loop) will never run and the time complexity will be equal to the complexity of the outer loop (for loop), i.e. O(n).

However, in the Worst and Average case, where the array is initially completely in reverse order, n-1, n-2, ..., 1, 0 comparisons are made to place each element in its correct place. Since this is an arithmetic array, n*(n-1)/2 operations are performed. These operations lead to O(n^2) time complexity.

Best: O(n)
Average: O(n^2)
Worst: O(n^2)

Bubble Sort :

In the best case when the array is already sorted at the beginning, Bubble Sort will have O(n) time complexity. This is because in this case, there are no situations that require swapping and the entire array will be scanned in one round. However, in the Worst case and Average case, where the array is initially completely in reverse order, n-1, n-2, ..., 1, 0 comparisons are made to place each element in its correct place. Since this is an arithmetic array, n*(n-1)/2 operations are performed. These operations lead to O(n^2) time complexity.

Best: O(n)
Average: O(n^2)
Worst: O(n^2)

Quick Sort :

In the Best and Average case, i.e. when the array is initially properly shuffled, the Quick Sort algorithm will have a time complexity of O(n log n). This means that the algorithm sorts the data by splitting it into chunks and sorting each chunk separately (log n).

In the Worst case, where the array is initially fully sorted or fully reverse sorted, the Quick Sort algorithm will have a time complexity of O(n^2). This is the case where the pivot element is moved through the array each time and thus n operations are performed along the array. At each stage, the size of the array decreases by 1 and this means that n*(n-1)/2 = O(n^2) operations will be performed.

Best: O(n log n)
Average: O(n log n)
Worst: O(n^2)

B) Running time of each sorting algorithm for each input.

Algorithm	Best	Average	Worst
Bubble Sort	1700	2700	3400
Insertion Sort	1400	2600	3400
Merge Sort	1500	5900	3300
Quick Sort	1470	2400	8600
Selection Sort	2500	2900	3700

<u>These results can be misleading due to the operating system, hardware, other programs running and the small size of the data.</u>

C) Comparison of the sorting algorithms

Best Case: Insertion> Quick>Merge>Bubble>Selection

Average Case:Quick>Insertion>Bubble>Selection>Merge

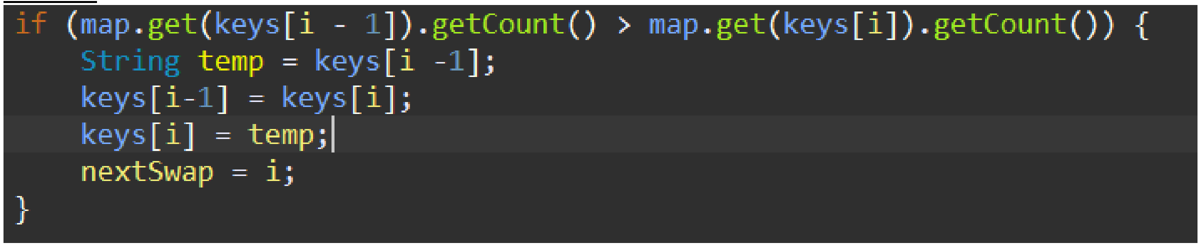
Worst Case:Merge>Bubble>=Insertion>Selection>Quick

In the best case, theoretically insertion and bubble would be expected to work best, in the average case merge and quick in. In the worst case, Merge should work best. In most of them, the agreement matches the experimental results, while the reasons for the parts that do not match can be said to be reasons such as working on a small data and system characteristics.

D) You are expected to analyze which algorithms keep the input ordering and which don't, along with the code snippet that causes/ensures this.

Since Selection Sort and Quick Sort are unstable sorting algorithms, they do not preserve the number of equal items.Merge,insertion,bubble keep the input ordering.

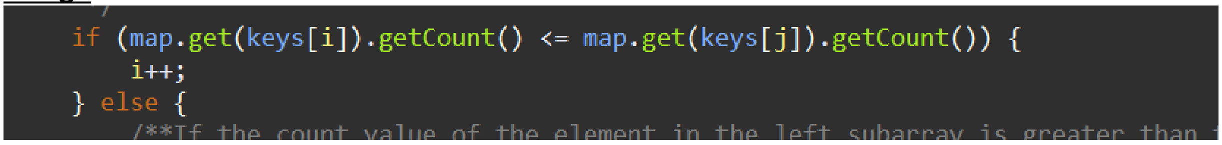
Bubble



```
if (map.get(keys[i - 1]).getCount() > map.get(keys[i]).getCount()) {
    String temp = keys[i -1];
    keys[i-1] = keys[i];
    keys[i] = temp;
    nextSwap = i;
}
```

This conditional expression is only valid if keys[i - 1] is greater than keys[i]. If the items have equal values, this conditional expression is invalid and the order of these items is not changed. Therefore, the original order of items with equal values is preserved.

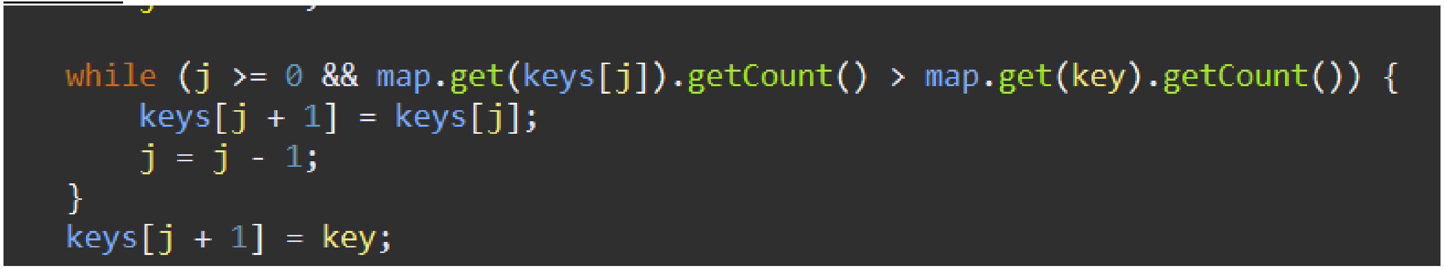
Merge



```
    if (map.get(keys[i]).getCount() <= map.get(keys[j]).getCount()) {
        i++;
    } else {
        /**If the count value of the element in the left subarray is greater than
```

This conditional expression is valid even when the getCount() values of keys[i] and keys[j] are equal. In this case, keys[i] (element of the left subarray) will come before keys[j] (element of the right subarray). This ensures that the original order of equal elements is preserved. That is, thanks to this expression, the first occurrence of equal-valued elements will also be first in the concatenated array. This ensures that the order of equal-valued elements is preserved.

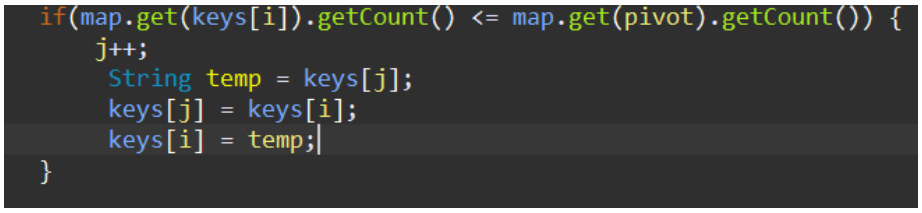
Insertion



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

During this comparison, if the getCount() values of the two keys are equal, this while loop stops and the current key remains in place. This ensures that the original order of equal items is preserved.
That is, thanks to this while loop, the first occurrence of equal-valued items will also be the first in the sorted array. This ensures that the order of equal-valued items is preserved.

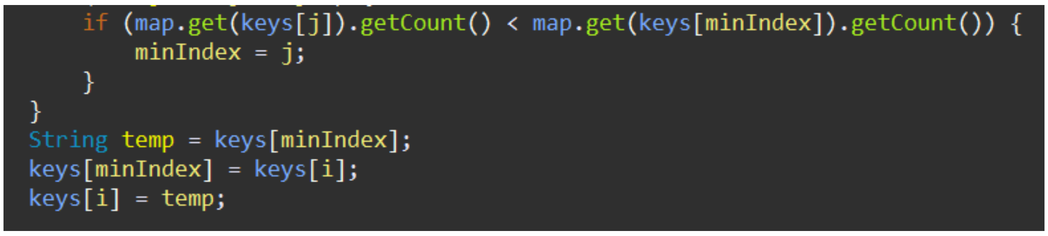
Quick Sort



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

Even when the value of the item being compared to the pivot is equal to the value of the pivot, a swap is performed. This swap between items of equal value causes the original ordering to be distorted.

Selection Sort



```
        if (map.get(keys[j]).getCount() < map.get(keys[minIndex]).getCount()) {
            minIndex = j;
        }
    }
    String temp = keys[minIndex];
    keys[minIndex] = keys[i];
    keys[i] = temp;
```

After finding the item with the smallest value in each iteration, we replace it with the first item. During this process, the original order of items of equal value is not preserved and the order between them is swapped.