GEBZE TECHNICAL UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CSE 222 - SPRING 2023

HOMEWORK 7 REPORT

ETHEM MENGAASLAN

200104004016

a) **Best, average, and worst-case time complexities analysis of each sorting algorithm. Make sure you explain your answer.**

Merge Sort:

Best Case: Assuming the array is already sorted, the merge operations are still performed, but no swapping is occurred. Dividing the array into two pieces until only one element left causes logarithmic time, so the time complexity is O(n log n).

***Average Case*:** Divides the array into two pieces and sorts it partially. The time complexity is again O(n log n).

***Worst Case*:** The array is divided into two pieces and put together in a sorted form. The time complexity is O(n log n).

Selection Sort:

Best Case: Assuming the array is already sorted, the algorithm will still traverse through the whole array in 2 nested loops to find minimum item at each iteration, but no swaps will be performed. So the time complexity is O(n^2).

Average Case: The algorithm traverses the whole array through 2 nested loops and does the necessary swappings. The time complexity is still O(n^2).

***Worst Case*:** The algorithm traverses the whole array through 2 nested loops and swaps every item until it is sorted. The time complexity is still O(n^2).

Insertion Sort:

Best Case: Assuming the array is already sorted, for each element the algorithm compares it with the elements in the already sorted portion and finds its correct position without any swaps. In this case, the time complexity of the algorithm is O(n).

Average Case: Insertion Sort iterates through the array, compares each element with the elements in the already sorted portion, and shifts them if necessary. Since the array is traverse multiple times until it is completely sorted, the time complexity of the algorithm is O(n^2).

Worst Case: For each element, the algorithm needs to compare each element with all previous elements in the sorted portion and perform a swap. This operations cost traversing the array approximately n*n times. So the time complexity is O(n^2).

Bubble Sort:

Best Case: Assuming the array is already sorted, the algorithm will iterate over the array n times and won't swap any values. So, the time complexity is O(n).

Average Case: The array is traversed n times by the outer loop and n times by the inner loop, so it is traversed n^2 times. The time complexity is O(n^2).

Worst Case: The array is traversed n times by the outer loop and n times by the inner loop, so it is traversed n^2 times. The time complexity is O(n^2).

Quick Sort:

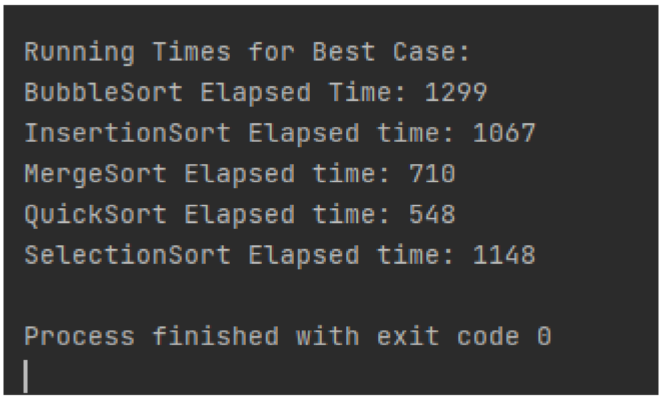
Best Case: In this scenario, the pivot chosen each time is the median element of the array (or sub-array). And the array is divided into two pieces at each recursive call, without swap operations being needed. The time complexity is therefore O(n log n).

Average Case: Performing quick sort, the array is divided into two pieces recursively and sometimes swaps are made. The dividing operations lead us to a time complexity of O(n log n).

Worst Case: In the worst case, the pivot element is always chosen as either the smallest or the largest element in the array(and sub-arrays) being worked on. In this case, the left and right subarrays will always be of size 1, and the algorithm will make O(log n) recursive calls, each of which will take O(n) time. So the time complexity is O(n^2).

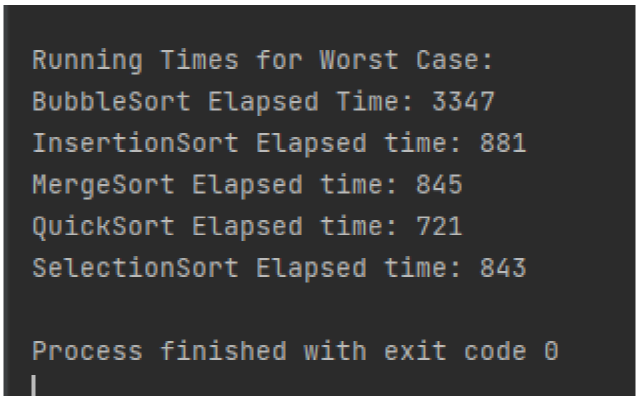
b) (10 pts) Running time of each sorting algorithm for each input.

Best Case Measurements:



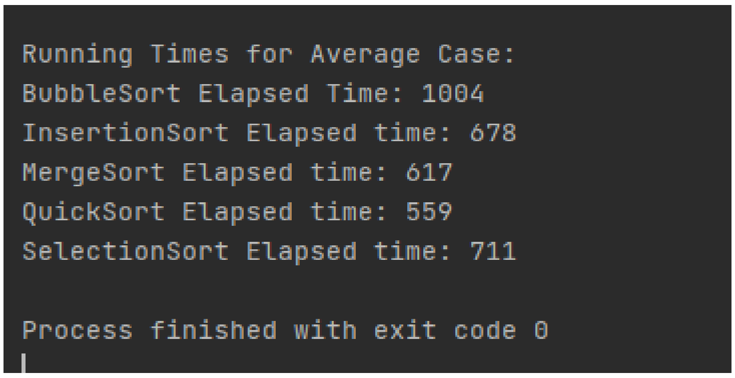
```
Running Times for Best Case:
BubbleSort Elapsed Time: 1299
InsertionSort Elapsed time: 1067
MergeSort Elapsed time: 710
QuickSort Elapsed time: 548
SelectionSort Elapsed time: 1148

Process finished with exit code 0
```

Worst Case Measurements:



```
Running Times for Worst Case:
BubbleSort Elapsed Time: 3347
InsertionSort Elapsed time: 881
MergeSort Elapsed time: 845
QuickSort Elapsed time: 721
SelectionSort Elapsed time: 843

Process finished with exit code 0
```

Average Case Measurements



```
Running Times for Average Case:
BubbleSort Elapsed Time: 1004
InsertionSort Elapsed time: 678
MergeSort Elapsed time: 617
QuickSort Elapsed time: 559
SelectionSort Elapsed time: 711

Process finished with exit code 0
```

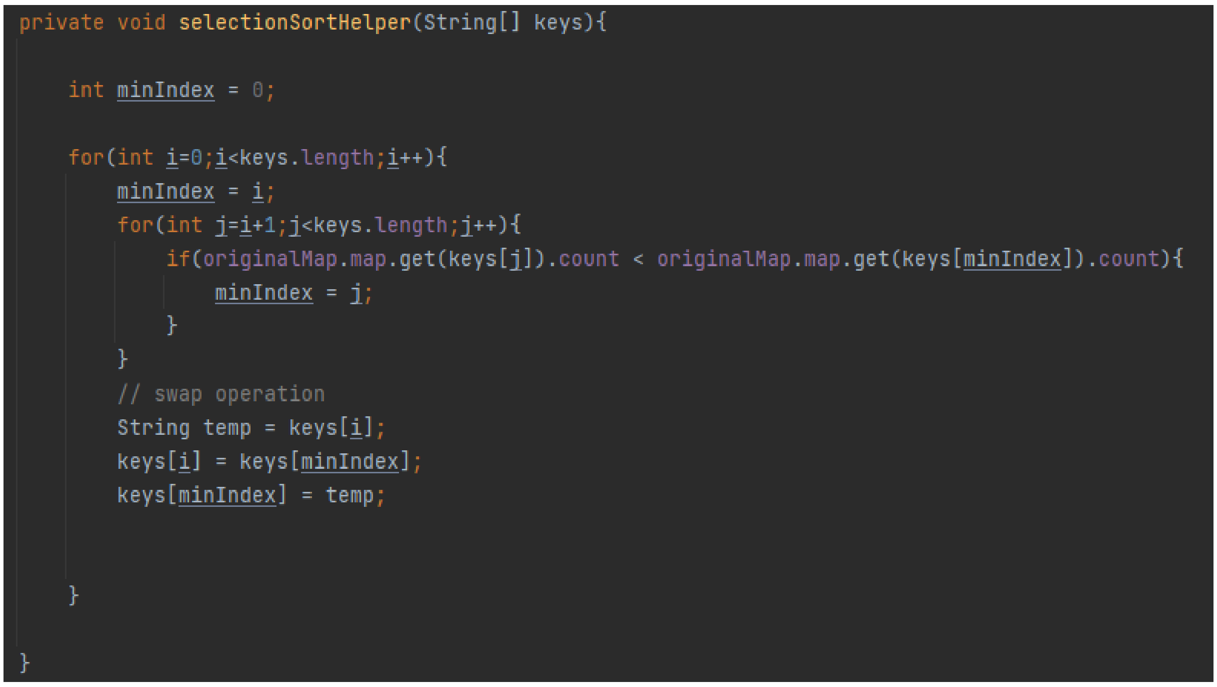
c) **Comparison of the sorting algorithms (by using the information from Part2-a and Part2-b) Which algorithm is faster in which case?**

At **best case, theoretically, Insertion Sort is the fastest** since it traverses the array n times and does not do any swapping operations. Assuming we had 1 million randomly ordered elements, the Quick Sort algorithm would perform better since it recursively divides the array and minimizes the sorting operation. **So Quick Sort is the one with the best performance average cases.** For the worst case, merge sort guarantees a worst case complexity of n log n. Compared with quick sort, when quick sort's pivot element is chosen to be the smallest or largest element in the array, because of the unbalanced partitions sourced from this, the time complexity of the quick sort can go up to O(n^2). But this is an uncommon case. But to guarantee success**, the merge sort algorithm is the fastest** including the edge cases **for the worst case scenario.**

**d) In HW6, it was highlighted that the ordering of the letters with same count value should be the same as their addition order to myMap object. However, for these 4 sorting algorithms,
the case might not be the same. You are expected to analyze which algorithms keep the input
ordering and which don't, along with the code snippet that causes/ensures this.**

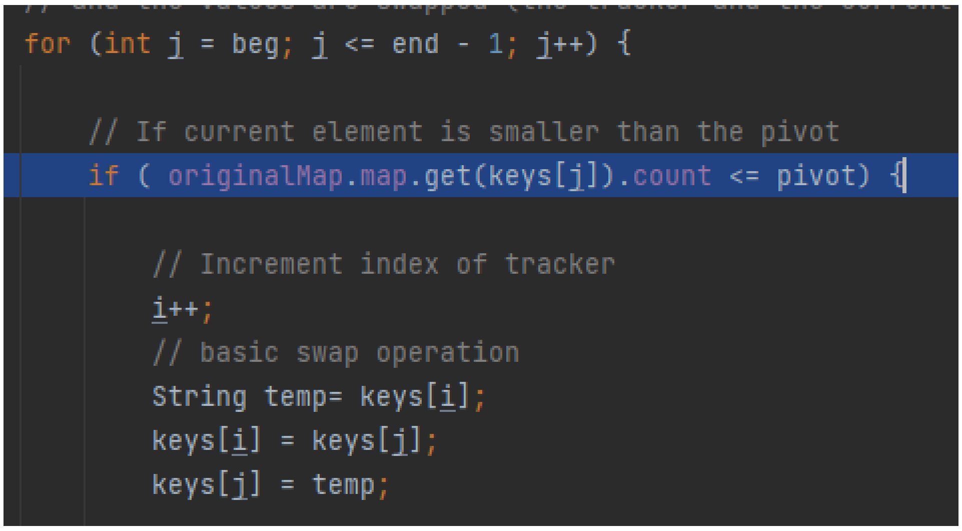
First of all, let's talk about **Stability** aspect of sorting algorithms.
Many sorting algorithms, when encountered with equal values, do not swap the values in order to preserve the original order in the sorted output. But **algorithms like Selection Sort and Quick Sort are not** Stable Sorting Algorithms by nature**. Here is why:**

Selection Sort: Selection Sort works as the following: it repedeatly finds the minimum element from the unsorted part of the array, and moves it to the sorted part of the array. While doing this, the array is shifted many times, therefore the original order of the array is not maintained even it's now sorted.



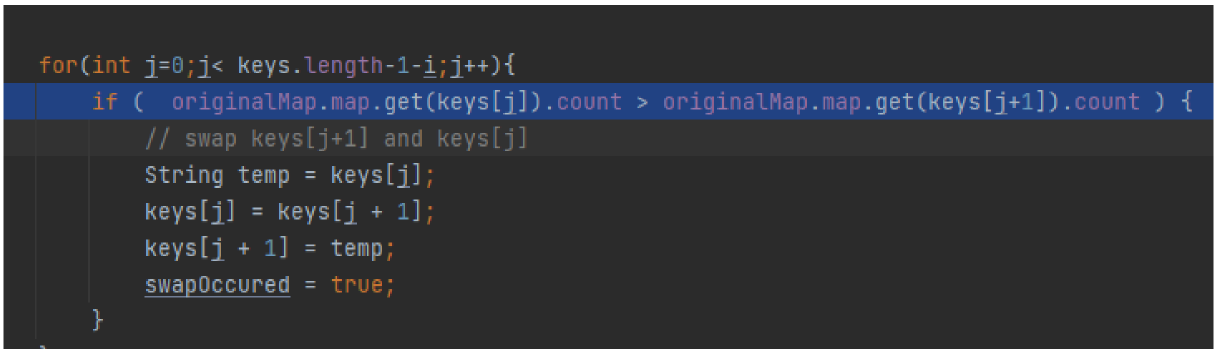
```java
private void selectionSortHelper(String[] keys){

    int minIndex = 0;

    for(int i=0;i<keys.length;i++){
        minIndex = i;
        for(int j=i+1;j<keys.length;j++){
            if(originalMap.map.get(keys[j]).count < originalMap.map.get(keys[minIndex]).count){
                minIndex = j;
            }
        }
        // swap operation
        String temp = keys[i];
        keys[i] = keys[minIndex];
        keys[minIndex] = temp;


    }

}
```

Quick Sort: The partitioning process in Quick Sort focuses on rearranging elements based on their relationship to the chosen pivot value, without considering the original order of equal elements. As a result, equal elements can still be swapped during the partitioning process, leading to a change in their original order.



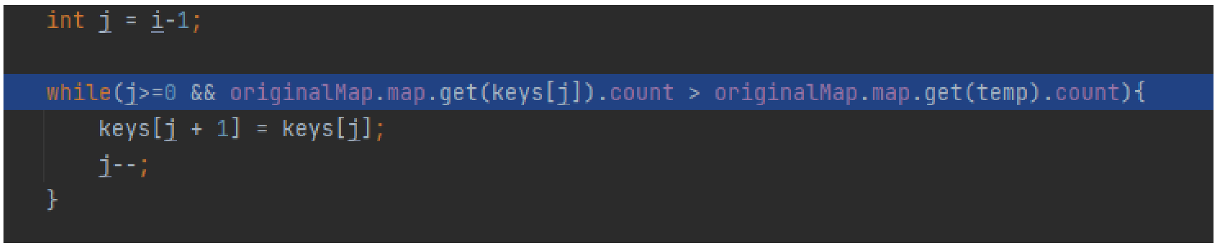
```
for (int j = beg; j <= end - 1; j++) {

    // If current element is smaller than the pivot
    if ( originalMap.map.get(keys[j]).count <= pivot) {

        // Increment index of tracker
        i++;
        // basic swap operation
        String temp= keys[i];
        keys[i] = keys[j];
        keys[j] = temp;
```

Bubble Sort: Bubble sort keeps the original order since it only swaps when two adjoining elements are not in order, and does not perform swapping operation when the order is correct.



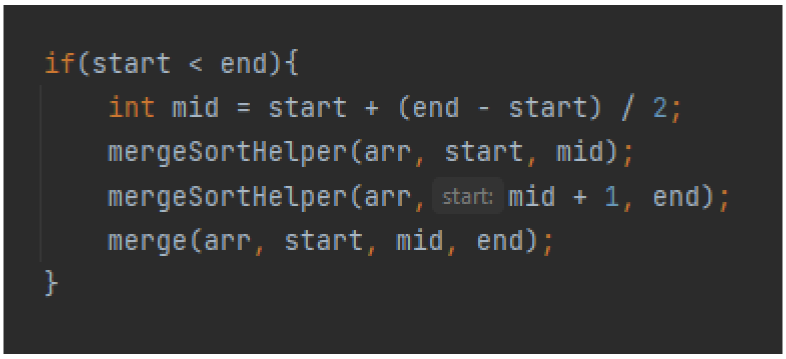
```
for(int j=0;j< keys.length-1-i;j++){
    if (  originalMap.map.get(keys[j]).count > originalMap.map.get(keys[j+1]).count ) {
        // swap keys[j+1] and keys[j]
        String temp = keys[j];
        keys[j] = keys[j + 1];
        keys[j + 1] = temp;
        swapOccured = true;
    }
```

Insertion Sort: Insertion sort keeps the order because it only inserts elements which causes the leftside and rightside items to be shifted, and shifting operation do not swap any equal values. Therefore, the original order is preserved.



```
int j = i-1;

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

Merge Sort: Merge Sort keeps the original order when the items are equal because it divides the array into smaller sub-arrays and sorts them recursively, then merges them back to form one single sorted array in a way that maintains the original order.



```
if(start < end){
    int mid = start + (end - start) / 2;
    mergeSortHelper(arr, start, mid);
    mergeSortHelper(arr, start: mid + 1, end);
    merge(arr, start, mid, end);
}
```