GEBZE TECHNICAL UNIVERSITY
CSE222 DATA STRUCTURE
HOMEWORK 7 REPORT

225008003102

SÜHA BERK KUKUK

1) Time Complexity Analysis

Sorting Algorithms	Time Complexity				
	Merge	Selection	Insertion	Quick	Bubble
Best	O(N log N)	$O(N^2)$	O(N)	O(N log N)	O(N)
Average	O(N log N)	$O(N^2)$	$O(N^2)$	O(N log N)	$O(N^2)$
Worst	O(N log N)	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(N^2)$

For the merge algorithm, all case scenarios (best, average and worst) are not dependent on input therefore, the algorithm always divides log N levels and merge operation takes linear time.

For the selection algorithm, as the merge algorithm, all case scenarios are not dependent on input because the algorithm compares each element one by one. One loop to select an element of input another loop to compare with another element. So, total time complexity is $O(N^2)$.

For the insertion algorithm, if input is sorted already, the algorithm takes O(N) that is best scenario. The worst-case scenario occurs when the input is in reverse order or in a completely unsorted state. In the average case scenario, Insertion Sort requires comparing and shifting elements multiple times to find the correct position for each element.

For the quick algorithm, a worst-case scenario occurs when the pivot is selected first or last of input. If the pivot is selected in the middle of input, the algorithm works best scenario. In the average case scenario, Quick Sort also has a time complexity of O(N log N). The partitioning process divides the array into two parts, but the division may not always result in balanced subarrays.

For the bubble algorithm, if input is sorted already, the algorithm takes O(N) that is best scenario. The worst-case scenario occurs when the input is in reverse order or in a completely unsorted state. In the average case scenario, bubble sort requires comparing and shifting elements multiple times to find the correct position for each element.

2) Running Time Table

Sorting Algorithms	Running Time (second)				
	Merge	Selection	Insertion	Quick	Bubble
Best	6.395474E-5	6.914005E-5	9.9335E-6	1.641291E-5	4.63182E-6
Average	6.395474E-5	6.914005E-5	9.471557E-5	4.74271E-6	9.733482E-5
Worst	6.395474E-5	6.914005E-5	9.471557E-5	4.74271E-6	9.733482E-5

3) Compare Sorting Algorithms

For the best case scenario, Bubble Sort and Insertion Sort are the fastest algorithms. It is expected that if we look at time complexity of best case scenario. They are linear time complexity.

For the average case scenario, Quick Sort and Merge Sort are the faster algorithms. It is expected that if we look at time complexity of average case scenario. They are O(N log N) is faster quadratic time complexity.

For the worst-case scenario, Merge Sort and Quick Sort have similar performance, while Bubble Sort, Insertion Sort, and Selection Sort are slower.

NOTE

For the Quick Sort Scenario: I change pivot depending on best, worst and average cases. I just changed just select pivot step. The other steps are same.

It is middle of pivot implementation:

```
int middlePivot = low + (high - low) / 2; // Selecting middle element
as the pivot
String pivotKey = aux[middlePivot];
swap(middlePivot, high);
int pivot = orginalmyMap.getCount(pivotKey);
```

It is last of pivot implementation:

```
String pivotKey = aux[high];
int pivot = orginalmyMap.getCount(pivotKey);
```

4)

Merge sort algorithm is stable sort, which means that the order of elements with equal values is preserved during the sort. While the sorting process prevents index. But I added while comparing counts, if both elements are same count, I check index. You can see implementation in the below. I added because make sure the sorted map.

```
if(tempCount1<tempCount2)
            {
                aux[k++]=temp[i++];

            }
            else if(tempCount1==tempCount2)
            {
                if(i<j)
                {
                    aux[k++]=temp[i++];;

                }
                else
                {
                    aux[k++]=temp[j++];;

                }
            }
            else
            {
                aux[k++]=temp[j++];;
            }
```

The default implementation of the Selection Sort Algorithm is not stable. However, it can be made stable. Therefore, I added code when it comes to same counts. You can see implementation below:

```
for(j=i+1;j<n;j++)
            {
                int counti=orginalmyMap.getCount(aux[min_index]);
                int countj=orginalmyMap.getCount(aux[j]);
                if(counti>countj)
                {
                    min_index=j;
                }
                else if(counti==countj)
                {
                    if(j<min_index)
                    {
                        min_index = j;
                    }
                }
            }
            String temp = aux[min_index];
            aux[min_index] = aux[i];
            aux[i] = temp;
```

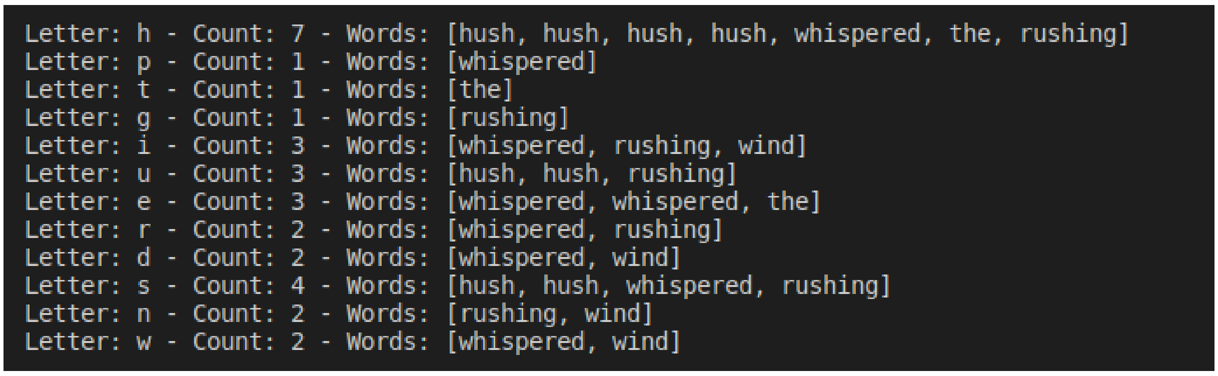
Bubble and Insertion sort are stable sorting algorithms. I don’t need implementation for check when it comes to same counts.

Quick sort is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot’s position. I add code for checking they are same counts in the partition part. You can see implementation in the below. But as you can see result in the below it did not work. Because the sorted algorithm works depend on pivot.

```
while(j<=high)
        {
            int element = orginalmyMap.getCount(aux[j]);
            if(element<pivot)
            {
                swap(i, j);
                i++;
            }
            else if(element==pivot)
            {
                swap(j, i+1);
                i++;
            }
            j++;
        }
```

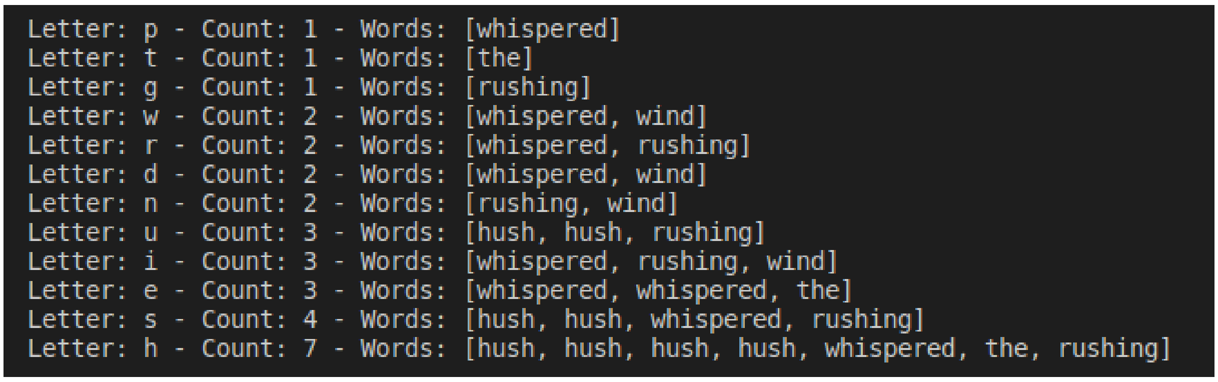
Example Result:

This result was obtained using pivot of the last element in the map.



```
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: w - Count: 2 - Words: [whispered, wind]
```

This result was obtained using pivot of the middle element in the map.



```
Letter: p - Count: 1 - Words: [whispered]
Letter: t - Count: 1 - Words: [the]
Letter: g - Count: 1 - Words: [rushing]
Letter: w - Count: 2 - Words: [whispered, wind]
Letter: r - Count: 2 - Words: [whispered, rushing]
Letter: d - Count: 2 - Words: [whispered, wind]
Letter: n - Count: 2 - Words: [rushing, wind]
Letter: u - Count: 3 - Words: [hush, hush, rushing]
Letter: i - Count: 3 - Words: [whispered, rushing, wind]
Letter: e - Count: 3 - Words: [whispered, whispered, the]
Letter: s - Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h - Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]
```