GTU Department of Computer Engineering
CSE 222/505 - Spring 2023
Homework #07 Report

Selim Aynigül
200104004004

Class Structure

In this homework we are asked to sort a given input String with different sorting algorithms: Bubble Sort, Quick Sort, Insertion Sort, Merge Sort and Selection Sort.

We have two packages for sorting, printing sorted maps, and testing running times.

package sort
-Sort.java
-MergeSort.java
-BubbleSort.java
-InsertionSort.java
-SelectionSort.java
-QuickSort.java
-MyMap.java
-info.java

package main
-Main.java
-TestClass.java

• I wanted to create an OOP class structure to save me from writing same methods and variables multiple times in sorting classes. To do this we have ***Sort*** abstract class and have all the other 5 sort class as subclasses of sort class. In this Sort class we have our common protected variables ***originalMap, sortedMap***, and protected methods ***swap()***,and ***getEntryAt()***. We also have abstract ***sort()*** method to overwrite in all sorting classes.

• In main class we can print sorted maps after sorted original map with all sorting algorithms.

• In test class we can test running times for all sorting algorithms with 3 different inputs for the best, worst and average cases.

Time Complexity Analysis

	Best Case	***Average Case***	***Worst Case***
Merge Sort	O(n.logn)	O(n.logn)	O(n.logn)

Description: Even if it works with the maps it still has same time complexities with the original merge sort algorithm since it does not use any other helper method with O(n) time as add or push methods. It uses recursive calls with subarrays that split in two. So it has a logn in the time complexity calculation.

Insertion Sort	O(n^2)	O(n^3)	O(n^3)

Description: It uses nested for, while, and and swap (which has O(n) time complexity) in average and worst cases. But for best case since we do not need to use swap method, it will has better time complexity.

Bubble Sort	O(n^3)	O(n^3)	O(n^3)

Description: Since it uses getCountAt() method (Which has O(n) time complexity) to compare map values in all situations, the time complexity does not change in best or worst case.

Selection Sort	O(n^3)	O(n^3)	O(n^3)

Description: Same as the bubble sort it uses getCountAt() method to compare values so time complexity does not change with different cases.

Quick Sort	O(n^2)	O(n^2)	O(n^3)

Description: Since the partition() method has O(n^2) time complexity with maps other than the original algorithm, it will dominate the other parts for best case and it will become O(n^2) from O(n^2.logn). In here n^2 comes from partition and logn comes from recursive calls with subarrays.

Comparison of Algorithms

Three different inputs used for the calculations.

Worst Case: Reverse sorted

Best Case: Already Sorted (in increasing order)

Average Case: Random

Results can slightly change for every run but we have an appropriate example print of the running times.

As we can see on the printed screen bubble sort and quick sort gave the fastest running time results in the best case. Merge sort gave a slower result even if it has a better time complexity mathematically. This might have happened because it uses too much space and many copying operations with the temporary maps.

But for the worst case, merge sort gave better results with quick sort. Because all the other methods has O(n^3) time complexity in the worst case and this resulted in slower running times.

Quick sort has the best running time in average case as well. So we can say that in this homework while doing sorting operations with maps, quick sort is the most efficient algorithm in terms of the running time.

```
====================
Merge Sort:
----------
Worst Case:   0.3528
Best Case:    0.2638
Average Case: 0.2881
====================
Selection Sort:
--------------
Worst Case:   0.3731
Best Case:    0.1911
Average Case: 0.2178
====================
Insertion Sort:
--------------
Worst Case:   0.8539
Best Case:    0.1689
Average Case: 0.6073
====================
Bubble Sort:
------------
Worst Case:   0.3789
Best Case:    0.0419
Average Case: 0.1917
====================
Quick Sort:
----------
Worst Case:   0.0928
Best Case:    0.0764
Average Case: 0.0821
```

Stability Analysis

- ***Merge Sort:*** The merge step in merge sort involves comparing elements from two sorted subarrays and merging them into a single sorted array. If there are elements with equal keys, the algorithm ensures that the element from the first subarray is placed before the element from the second subarray.

```
while (!left.isEmpty() && !right.isEmpty()) {
    if (getFirstEntry(left).getValue().getCount() <= getFirstEntry(right).getValue().getCount()) {
        mergedMap.put(getFirstEntry(left).getKey(), getFirstEntry(left).getValue());
        left.remove(getFirstEntry(left).getKey());
    }
    else {
        mergedMap.put(getFirstEntry(right).getKey(), getFirstEntry(right).getValue());
        right.remove(getFirstEntry(right).getKey());
    }
}
```

- ***Selection Sort:*** If there are two or more elements with the same count value selection sort does not consider their original order. As a result, the relative order of elements with equal values may change after the swap operations, leading to instability.

```
for(int i = 0; i < n - 1; i++) {
    int minIndex = i;
    for(int j = i + 1; j < n; j++) {
        if(getCountAt(j) <= getCountAt(minIndex))
            minIndex = j;
    }
    if(minIndex != i) swap(i, minIndex);
```

- ***Insertion Sort:*** In insertion sort when inserting an element, if there are already elements with equal count values in the sorted portion, the algorithm ensures that the element being inserted is placed after the existing elements with equal values.

```
for (int i = 1; i < n; i++) {
    Map.Entry<Character, info> key = getEntry(i);
    int j = i - 1;
    while (j >= 0 && getCountAt(j) > key.getValue().getCount()) {
        swap(j + 1, j--);
    }
    insert(j + 1, key);
}
```

- ***Bubble Sort:*** In bubble sort, adjacent elements are compared and swapped if they are in the wrong order. During each pass, if two adjacent elements have equal count values they are not swapped, thereby preserving their original relative order.

```
for(int i = 0; i < n; i++)
    for(int j = 1; j < n - i; j++)
        if(getCountAt(j - 1) > getCountAt(j))
            swap(j - 1, j);
```

- ***Quick Sort:*** During the partitioning step of quicksort, elements are rearranged around a pivot element based on their comparison to the pivot. Equal elements may be swapped or moved to different partitions during this process.

```
int i = low - 1;
for (int j = low; j < high; j++) {
    if (getCountAt(j) < getCountAt(high))
        swap(++i, j);
}
swap(i + 1, high);
```

Running Command and Results

Compiling commands:

cd homework7
javac sort/*.java main/*.java test/*.java

Running commands:

java main/Main
java main/TestClass

java main/Main

```
C:\Users\Administrator\Desktop\deneme>java main/Main

Input: 'Hush, hush!' whispered the rushing wind.

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

Merge Sort:
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]

Quick Sort:
Letter: g -  Count: 1 - Words: [rushing]
Letter: t -  Count: 1 - Words: [the]
Letter: p -  Count: 1 - Words: [whispered]
Letter: n -  Count: 2 - Words: [rushing, wind]
Letter: w -  Count: 2 - Words: [whispered, wind]
Letter: d -  Count: 2 - Words: [whispered, wind]
Letter: r -  Count: 2 - Words: [whispered, 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]


Bubble Sort:
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]

Selection Sort:
Letter: g -  Count: 1 - Words: [rushing]
Letter: t -  Count: 1 - Words: [the]
Letter: p -  Count: 1 - Words: [whispered]
Letter: n -  Count: 2 - Words: [rushing, wind]
Letter: w -  Count: 2 - Words: [whispered, wind]
Letter: d -  Count: 2 - Words: [whispered, wind]
Letter: r -  Count: 2 - Words: [whispered, 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: s -  Count: 4 - Words: [hush, hush, whispered, rushing]
Letter: h -  Count: 7 - Words: [hush, hush, hush, hush, whispered, the, rushing]

Insertion Sort:
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]
```

java main/TestClass

```
Worst Case Input:
====================
oooooooooooooo nnnnnnnnnnnnnn mmmmmmmmmmmmm lll
lllllllll kkkkkkkkkk jjjjjjjjjj iiiiiiiii hhhhh
hhh ggggggg ffffff eeeee dddd ccc bb a

Best Case Input:
====================
a bb ccc dddd eeeee ffffff ggggggg hhhhhhh iiii
iiiii jjjjjjjjjj kkkkkkkkkk lllllllllll mmmmmm
mmmmmmm nnnnnnnnnnnnn ooooooooooooooo

Average Case Input:
====================
eeeee ggggggg ccc nnnnnnnnnnnnn lllllllllll bb
 ooooooooooooooo a kkkkkkkkkk jjjjjjjjjj ffffff
 iiiiiiii mmmmmmmmmmmmm dddd


====================
Merge Sort:
-----------
Worst Case:   0.3528
Best Case:    0.2638
Average Case: 0.2881
====================
Selection Sort:
---------------
Worst Case:   0.3731
Best Case:    0.1911
Average Case: 0.2178
====================
Insertion Sort:
---------------
Worst Case:   0.8539
Best Case:    0.1689
Average Case: 0.6073
====================
Bubble Sort:
------------
Worst Case:   0.3789
Best Case:    0.0419
Average Case: 0.1917
====================
Quick Sort:
-----------
Worst Case:   0.0928
Best Case:    0.0764
Average Case: 0.0821
```