GTU Department of Computer Engineering
CSE 222 - Spring 2023
Homework 7 Report

BEYZA ACAR

200104004065

19 .05.2023

1-System Requirements

First of all, in this assignment, we are expected to improve the previous assignment, so below, I should mention the previous assignment requirements.

• PART 1:
An input string was taken, declared within the code. The string should only contain lowercase letters and no non-alphabetic characters. The original and modified strings were printed, where the modification was done using regex.

• PART 2:
A myMap object was constructed with the modified string. Each letter should only be added once, and if a letter is repeated, the count of the Info object should be increased, and the word should be added.

• PART 3:
A mergeSort object was constructed with the myMap object. The myMap object was sorted using the merge sort algorithm via the mergeSort method. The original myMap object and the sorted myMap object were printed.

OTHER RULES :

- myMap class should contain :
  - LinkedHashMap<String, Info> map
  - int mapSize
  - String str
- Info class should contain :
  - int count
  - String [] words
- mergeSort class should contain :
  - myMap originalMap
  - myMap sortedMap
  - a helper data type for key values (may be string, but I use a nested class for it named keyValuePair that I am going to mention later.

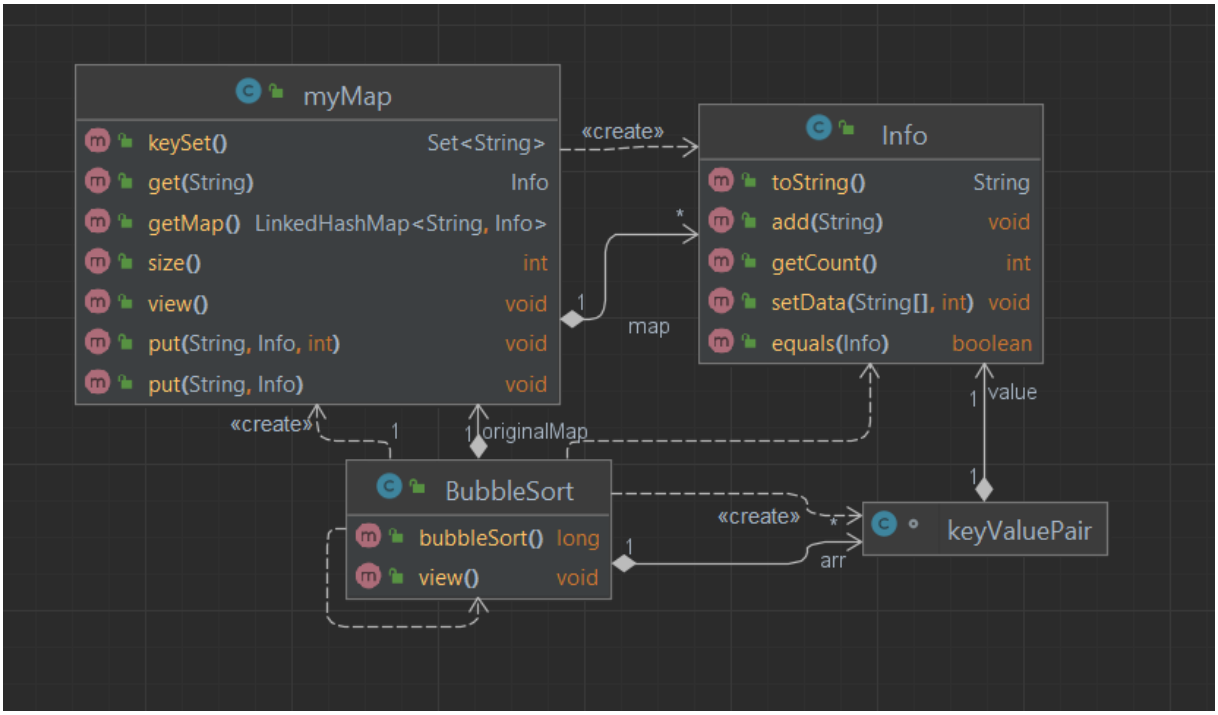
In this homework;

We are expected to implement various sorting algorithms including merge sort, quick sort, bubble sort, insertion sort, selection sort with the LinkedHashMap data type; calculate and print the running time of these algorithms.

- Every sort algorithm must take place in a class
- These sorting purposed classes must include 2 myMap object. One for original map and other one is for sorted version of the original array.
- Allowed libraries are listed below:
    - Java.util.List
    - Java.util.ArrayList
    - Java.util.Set
    - Java.util.Map
    - Java.util.LinkedHashMap

2- ***Uml Diagram***

This is the uml diagram of Bubble sort algorithm but it is the same for all other sorting classes. So I just added BubbleSort Class' UML diagram because homework's uml diagram is extremely chaotic.



3- *Problem Solution Approach*

Note: keyValuePair class has a String and Info in it to hold an element of LinkedHashMap.

I have implemented all the sorting algorithms in the known forms, with some modifications to fit the LinkedHashMap data type. Firstly, I copied the data from the LinkedHashMap structure into a KeyValuePair array using the fillTheArray() method. I performed the sorting operations on this array, without modifying the original map. When the sorting process is done, I used the copyToSortedMap() method to transfer the resulting sorted KeyValuePair array into the sortedMap instance variable of type LinkedHashMap. Therefore, the sort method called by the user is actually a helper method that sequentially calls fillTheArray(), the real sorting algorithm (which varies for each class), and copyToSortedMap() methods. At the end the returned time only represents the time spent during the sorting algorithm itself, which means, without including the time taken for the fill and copy operations. I have followed this process consistently in every other sorting classes.

4- *Time Complexities*

a. *Merge Sort*

In the merge sort algorithm, we have two part : mergeSort and merge parts. mergeSort method calls merge method, so total complexity must be :

time complexity of mergeSort() * time complexity of merge()

PART 1 : time complexity of mergeSort

MergeSort method divides array by two and recursively calls itself to sort every divided part. So time complexity is logn. For example if we have a 16 sized array:

1. 16 divided by two array : 8 – 8
2. These arrays divided by two again : 4 – 4 – 4 – 4
3. 2 – 2 – 2 – 2 – 2 – 2 – 2 – 2
4. 1 – 1 – 1 – 1 – 1 - 1 – 1 – 1 – 1 – 1 - 1 – 1 – 1 – 1 – 1 - 1

As seen above, mergeSort was called 4 times that equals to log16.

PART 2 : time complexity of merge

In the merge method, left and right arrays are merged to created a sorted array . During this process, each element is compared and placed its accurate position. Placing and comparing operation is done in a constant time. So the time complexity of merge method is O(n).

* So no case will cause merge sort to show different behavior. Therefore, there is no need to perform separate analyses for the best-case, avarage case and worst-case time complexities.

b. Quick Sort

a. In the quick sort algorithm, we have two part : quickSort and partition parts. quickSort method calls partition method, so total complexity must be :

time complexity of mergeSort() * time complexity of merge()

PART 1 : time complexity of partition()

Partition method is linear no matter what. It just checks all the elements and places them in descending or ascending order.

PART 2 : time complexity of quickSort()

Time complexity of quick sort totally depends on pivot. If the pivot is succeeded in choosing from the middle rather than the beginning or the end, which makes the quickSort a quick sort, which is more likely in arrays with large elements, the time complexity will be logn since the array will always be split in half. But, if choosen pivot element consistently results in unbalanced partitons, it makes quickSort() method's time complexity O(n^2). Because in this case, we cannot split the array into two, in which case we are sorting each element individually.

i. Worst Case Time Complexity

The worst-case time complexity of quick sort occurs when the array is completely reverse sorted or sorted or when the selected pivot element is always the smallest or largest element.

ii. Avarage Case Time Complexity

The avarage-case time complexity of quick sort occurs when pivot can be choosen near the middle.

iii. Best Case Time Complexity

The best-case time complexity of quick sort occurs when pivot can be choosen in the middle every time.

c. Bubble Sort

Bubble sort algorithm has two loop in it, which makes its complexity is O(n^2). Outer loop starts from 0 to n, and inner loop start from 0 again to n – 1 – i(first loop counter). So it can be calcuated as follows :

n*(n-1) + (n-1)*(n-2) + (n-2)*(n-3) + (n-3)*(n-4) + ….. + 1*0

Since highest degree term is n^2, time complexity of bubble sort is O(n^2) in every case.

d. Selection Sort

Selection sort algorithm has two loop in it, which makes its complexity is O(n^2). Outer loop starts from 0 to n, and in every loop findMin method is called, findMin method finds min starting from the given index (i : outer loop counter), and at the end swap method swaps two element with the time complexity of O(1). So whole method's time complexity can be calculated as follows:

n*n + (n-1)*(n-1) + (n-2)*(n-3) + … + 1*1

Since highest degree term is n^2, time complexity of selection sort is O(n^2) in every case.

e. Insertion Sort

Insertion sort algorithm has two loop in it, which makes its complexity is O(n^2). Outer loop starts from 1 to n, the aim of the inner loop is to compare the current element with the previous element, and if the current element is smaller, repeatedly compare and shift it to the left by one position, that means inner loop is n-1 to 0. So it can be calcuated as follows :

(n-1)*(n-2) + (n-2)*(n-3) + (n-3)*(n-4) + … + 1*0

Since highest degree term is n^2, time complexity of selection sort is O(n^2) in every case.

5- Test Cases – Running Times

Input1: Best Case

"abbcccddddeeeeeffffffggggggghhhhhhhhiiiiiiiiijjjjjjjjjkkkkkkkkkkkllllllll lllllmmmmmmmmmmmmmmnnnnnnnnnnnnnnnoooooooooooooooppp pppppppppppppqqqqqqqqqqqqqqqqrrrrrrrrrrrrrrrrrrssssssssssssssssss sstttttttttttttttttttuuuuuuuuuuuuuuuuuuuuuz"

Input2 : Worst Case

“uuuuuuuuuuuuuuuuuuuuuttttttttttttttttttttsssssssssssssssssssrrrrrrr rrrrrrrrrrrqqqqqqqqqqqqqqqqqpppppppppppppppppppooooooooooooo oonnnnnnnnnnnnnnnnnmmmmmmmmmmmmmmlllllllllllllkkkkkkkkkkkkjjjj jjjjjjjiiiiiiiiiiihhhhhhhhhgggggggggfffffffeeeeeeeedddcccbbba”

Input3 : Avarage Case

“qqqqqqqqqqqqqqqqqaccctttttttttttttttttttsssssssssssssssssshhhhhh hhfffffffiiiiiiiiiiinnnnnnnnnnnnnnnnbbrrrrrrrrrrrrrrrrrrddddjjjjjjjjjjjeeeee eelllllllllllllkkkkkkkkkkkkmmmmmmmmmmmmmmmpppppppppppppppppppo ooooooooooooooouuuuuuuuuuuuuuuuuuuuu”

```
TIME ANALYSIS FOR DIFFERENT INPUTS
-------------------------------------------------
INPUT 1
-------------------------------------------------
original string: abbcccddddeeeeeffffffgggggggghhhh
modified string: abbcccddddeeeeeffffffgggggggghhhh

time for bubbleSort = 69700 ns
time for selectionSort = 48500 ns
time for insertionSort = 7500 ns
time for mergeSort = 67200 ns
time for quickSort = 31900 ns
-------------------------------------------------
INPUT 2
-------------------------------------------------
original string: uuuuuuuuuuuuuuuuuuuuuttttttttttt
modified string: uuuuuuuuuuuuuuuuuuuuuttttttttttt

time for bubbleSort = 54500 ns
time for selectionSort = 59300 ns
time for insertionSort = 23000 ns
time for mergeSort = 51000 ns
time for quickSort = 31700 ns
-------------------------------------------------
INPUT 3
-------------------------------------------------
original string: qqqqqqqqqqqqqqqqqaccctttttttttt
modified string: qqqqqqqqqqqqqqqqqaccctttttttttt

time for bubbleSort = 39700 ns
time for selectionSort = 52700 ns
time for insertionSort = 15300 ns
time for mergeSort = 56300 ns
time for quickSort = 23100 ns

Process finished with exit code 0
```

6-Comparison of Running Times

As can be seen in the results above, **insertion sort** is the best among the sorting algorithms with n^2 complexity.In best case the difference between **bubble sort and selection sort** is that selection sort's findMin method finds minimum in first element every time, while bubble sort performs many more checking. In worst case the slight difference **between bubble sort and selection sort** is that of the swap and findMin calls made in the selection sort. In avarage case the difference between **bubble sort and selection sort** is that while selection sort swaps once, bubble sorts swaps much more time. The reason why mergeSort works so slowly in all cases is that our dataset is not big enough. The reason why **quick sort** only performs the fastest in the average case section is because the pivot is unbalanced in the best and worst cases, that is, the pivot is always the largest or the smallest. In this case, it cannot divide the array into two.

7-Stability Control

```
STABILITY TEST
Input : Hello World
-------------------------------------------------
Press any key to continue...

original string: Hello World
modified string: hello world

------------------Original Map---------------------
Letter: h - Count: 1 - Words: [hello]
Letter: e - Count: 1 - Words: [hello]
Letter: l - Count: 3 - Words: [hello hello world]
Letter: o - Count: 2 - Words: [hello world]
Letter: w - Count: 1 - Words: [world]
Letter: r - Count: 1 - Words: [world]
Letter: d - Count: 1 - Words: [world]
Map Size: 7
```

RESULTS:

```
------------------Sorted Map (BUBBLE SORT)--------------------
Letter: h - Count: 1 - Words: [hello]
Letter: e - Count: 1 - Words: [hello]
Letter: w - Count: 1 - Words: [world]
Letter: r - Count: 1 - Words: [world]
Letter: d - Count: 1 - Words: [world]
Letter: o - Count: 2 - Words: [hello world]
Letter: l - Count: 3 - Words: [hello hello world]
```

```
------------------Sorted Map (SELECTION SORT)--------------------
Letter: h - Count: 1 - Words: [hello]
Letter: e - Count: 1 - Words: [hello]
Letter: w - Count: 1 - Words: [world]
Letter: r - Count: 1 - Words: [world]
Letter: d - Count: 1 - Words: [world]
Letter: o - Count: 2 - Words: [hello world]
Letter: l - Count: 3 - Words: [hello hello world]
```

```
------------------Sorted Map (INSERTION SORT)--------------------
Letter: h - Count: 1 - Words: [hello]
Letter: e - Count: 1 - Words: [hello]
Letter: w - Count: 1 - Words: [world]
Letter: r - Count: 1 - Words: [world]
Letter: d - Count: 1 - Words: [world]
Letter: o - Count: 2 - Words: [hello world]
Letter: l - Count: 3 - Words: [hello hello world]
```

```
------------------Sorted Map (MERGE SORT)--------------------
Letter: h - Count: 1 - Words: [hello]
Letter: e - Count: 1 - Words: [hello]
Letter: w - Count: 1 - Words: [world]
Letter: r - Count: 1 - Words: [world]
Letter: d - Count: 1 - Words: [world]
Letter: o - Count: 2 - Words: [hello world]
Letter: l - Count: 3 - Words: [hello hello world]
```

```
------------------Sorted Map (QUICK SORT)--------------------
Letter: d - Count: 1 - Words: [world]
Letter: h - Count: 1 - Words: [hello]
Letter: e - Count: 1 - Words: [hello]
Letter: r - Count: 1 - Words: [world]
Letter: w - Count: 1 - Words: [world]
Letter: o - Count: 2 - Words: [hello world]
Letter: l - Count: 3 - Words: [hello hello world]
```

INFERENCES OF STABILITY CONTROL

As seen in the outputs, the letter h has a count of 1 but its order has changed, indicating that the quickSort is not stable.

All algorithms except quick sort are stable, while quick sort is not.