GTU Department of Computer Engineering
CSE 222/505 - Spring 2023
Homework #7 Report

REŞİT AYDIN
200104004019

1. Problem Definition

In this homework, we were supposed to sort a given map using various algorithms such as merge, selection, insertion, bubble, quick sorts. We sort the map according to its value which is the occurrence count of characters in it.

2. Problem Solution Approach

It is always easy to sort a given list or array that contains basic data types but, in our case, it is not one of them but a map of character and info class. We were asked to sort maps according to its values which exist in another class which named info. I modified classic sorting algorithms so that the algorithms compare the values of map then sort them according to the value.

3. Examining the Time Complexities of the Algorithms and Examining the Distinctions Among them

3.a - Best, average, and worst-case time complexities analysis of each sorting algorithm

3.a.1 – Merge Sort

Best: O(n * logn)

Average: O(n * logn)

Worst: O(n * logn)

Merge sort has the same time complexity for best, average and worst cases. This is because, regardless of the initial order of the elements, Merge sort will always divide the array into equal halves at each recursive step until the base case is reached.

Moreover, in merge sort algorithm, the merging process takes linear time, proportional to the size of the input array. Therefore, the dominant factor in the time complexity of Merge sort is the recursive splitting of the array. This makes it a reliable and efficient sorting algorithm, especially for large input sizes, where its performance is generally better than algorithms such as bubble sort or insertion sort.

3.a.2 – Selection Sort

Best: O(n^2)

Average: O(n^2)

Worst: O(n^2)

Selection sort has the same time complexity for best, average and worst cases. The crucial point is that the number of comparisons and swaps is determined by the algorithm's design, which remains consistent regardless of the input characteristics or order. The algorithm scans the entire unsorted portion of the array in each pass to find the minimum or maximum element, resulting in a fixed number of operations. The reason why the time complexity of selection sort remains unchanged regardless of the input characteristics is because the algorithm's design guarantees a fixed number of comparisons and swaps in each case.

3.a.3 – Insertion Sort

Best: O(n)

Average: O(n^2)

Worst: O(n^2)

The best case occurs when the input array is already sorted in ascending order. In this scenario, insertion sort can determine that each element is in its correct position with just one comparison, and no swaps are required. As a result, the algorithm performs a linear number of operations, resulting in a time complexity of O(n). In the average case, insertion sort has a time complexity of O(n^2).

The average case assumes that the input array is randomly ordered. In insertion sort, for each element, the algorithm iterates backward through the sorted portion of the array, comparing and shifting elements until it finds the correct position to insert the current element. Therefore, the number of operations grows quadratically with the input size, resulting in a time complexity of O(n^2).

The worst case of insertion sort occurs when the input array is sorted in descending order. In this case, the algorithm needs to shift every element to the right to insert each subsequent element, resulting in the maximum number of comparisons and swaps. For each element, it compares and shifts all the sorted elements until it finds the correct position. The number of operations in the worst case is proportional to (n^2 - n)/2, leading to a time complexity of O(n^2).

Overall, the time complexity of insertion sort makes it suitable for small-scale or nearly sorted inputs, but it may not be the most efficient choice for larger or unsorted arrays.

3.a.4 – Bubble Sort

Best: O(n)

Average: O(n^2)

Worst: O(n^2)

The best case occurs when the input array is already sorted. In this scenario, bubble sort can detect that no swaps are needed after a single pass through the array. With the input already sorted, no swaps are required, resulting in a time complexity of O(n).

In the average case, bubble sort has a time complexity of O(n^2). The average case assumes that the input array is randomly ordered. Bubble sort compares adjacent elements and swaps them if they are in the wrong order, repeatedly traversing the array until it is sorted. On average, the algorithm needs to make approximately n/2 passes through the array for each element, resulting in a quadratic time complexity of O(n^2).

The worst case of bubble sort occurs when the input array is sorted in descending order. In this case, bubble sort requires the maximum number of swaps for each element. It needs to traverse the array multiple times, repeatedly comparing and swapping adjacent elements until the largest element "bubbles" to the end of the array. The number of comparisons and swaps is proportional to (n^2 - n)/2, resulting in a time complexity of O(n^2).

In conclusion, bubble sort has a quadratic time complexity in both average and worst cases, making it inefficient for larger input sizes. Other sorting algorithms, such as merge sort or quicksort, have better time complexities (O(n log n)) and are typically preferred for larger or unsorted arrays.

3.a.5 – Quick Sort

Best: O(n * logn)

Average: O(n * logn)

Worst: O(n^2)

The best case occurs when the pivot chosen in each partitioning step divides the array into two nearly equal-sized subarrays. In this scenario, the recursive calls are evenly balanced, leading to efficient sorting. The best case time complexity of quicksort is O(n log n).

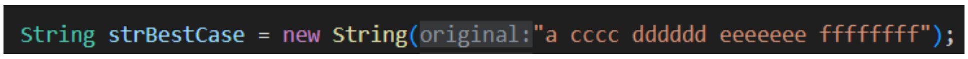
The average case time complexity of quicksort is also O(n log n). The average case assumes that the input array is randomly ordered. Quicksort partitions the array based on a chosen pivot element and recursively operates on the resulting subarrays.

The worst case occurs when the pivot selection or partitioning process consistently results in highly imbalanced subarrays. For example, this can happen if the pivot is chosen as the smallest or largest element in each partition, resulting in only one element being placed in one subarray and the rest in the other. In the worst case, the time complexity of quicksort is O(n^2).

Overall, merge sort's stable sorting property, consistent time complexity, and potential for parallelization make it an excellent choice for sorting applications where stability and predictable performance are important. However, its additional space requirements and non-in-place nature can be limiting factors in memory-constrained or resource-limited environments.

3.b - Running time of each sorting algorithm for each input

3.b.1 - Best Case Inputs



```
String strBestCase = new String(original:"a cccc dddddd eeeeeee ffffffff");
```

Merge Sort: 11100 ns

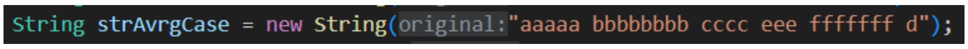
Selection Sort: 10900 ns

Insertion Sort: 6700 ns

Bubble Sort: 5700 ns

Quick Sort: 28500 ns

3.b.2 - Average Case Inputs



```
String strAvrgCase = new String(original:"aaaaa bbbbbbbb cccc eee fffffff d");
```

Merge Sort: 8500 ns

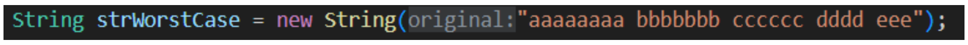
Selection Sort: 99000 ns

Insertion Sort: 46800 ns

Bubble Sort: 91500 ns

Quick Sort: 22500 ns

3.b.3 - Worst Case Inputs



```
String strWorstCase = new String(original:"aaaaaaaa bbbbbbb cccccc dddd eee");
```

Merge Sort: 6600 ns

Selection Sort: 4200 ns

Insertion Sort: 4000 ns

Bubble Sort: 244400 ns

Quick Sort: 9700 ns

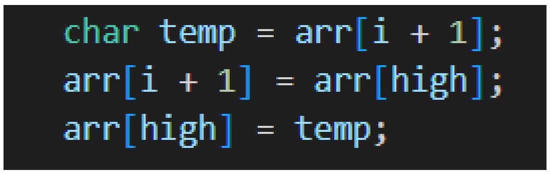
3.c - Comparison of the sorting algorithms

Based on the information in part 3.a, Merge Sort is the most efficient algorithm in terms of time complexity with a worst, average and best-case time complexity of O(n log n). The least efficient algorithm in terms of time complexity is Selection Sort with a worst, average and best-case time complexity of O(n^2).

Based on the information in part 3.b, in best case, Bubble sort is the fastest algorithm. In average case, Merge Sort is the most efficient algorithm. In worst case, both Insertion and Selection are the fastest algorithms.

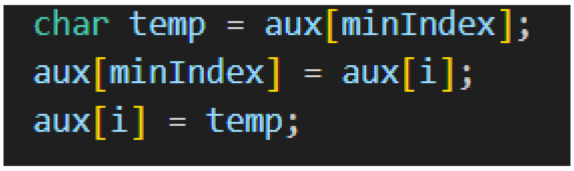
3.d - Analysing the order of the outputs of algorithms

Bubble Sort, Insertion Sort, Merge Sort keeps the ordering of the letters while others don’t.



```
    char temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
```

In Quick Sort, these lines cause it to not keep the ordering.



```
  char temp = aux[minIndex];
  aux[minIndex] = aux[i];
  aux[i] = temp;
```

In Selection Sort, these lines cause it to not keep the ordering.