CSE222

HomeWork #7

Report

1901042676

Mustafa Mercan

Part a

Merge Sort

Best, average, and worst case: Merge Sort consistently performs at the same level and has a time complexity of O(n log n) based on the size of the array. This is due to the algorithm continuously dividing the array in half and merging it back together in a sorted manner using a recursive approach.

Selection Sort:

Best case: O(n^2)

Average case: O(n^2)

Worst case: O(n^2)

The Selection Sort algorithm exhibits the same performance in the best, average, and worst cases. In all cases, it uses loops to compare elements and find the minimum or maximum value to place it in the correct position.

Insertion Sort:

Best case: If the array is already sorted, Insertion Sort completes in a single pass. In this case, the time complexity is O(n).

Average case: In the average case, Insertion Sort uses nested loops to place each element in its proper position, resulting in a time complexity of O(n^2).

Worst case: If the array is sorted in reverse order, Insertion Sort completes by comparing and swapping all elements. In this case, the time complexity is also O(n^2).

Bubble Sort

Best case: If the array is already sorted, Bubble Sort completes in a single pass. In this case, the time complexity is O(n).

Average case: In the average case, Bubble Sort compares every two elements, resulting in a time complexity of O(n^2) depending on the size of the array.

Worst case: If the array is sorted in reverse order, Bubble Sort completes by comparing and swapping all elements. In this case, the time complexity is again O(n^2).

Quick Sort

Best case: In the best case, Quick Sort partitions the array exactly in the middle and divides it in half each time. In this case, the time complexity is O(n log n).

Average case: In the average case, Quick Sort selects a random element as the pivot and divides the array for sorting. The average time complexity is also considered to be O(n log n).

Worst case: In the worst case, Quick Sort occurs when the pivot is consistently chosen as the largest or smallest element, resulting in partitioning with only one element less each time. In this case, the time complexity is O(n^2).

Part b

Merge Sort

In Merge Sort algorithm, the best, average, and worst case scenarios are the same. The time complexity of Merge Sort is always O(n log n).

input = “a bb ccc dddd eeee” (ordered list)

```
Merge Sort execution time: 44725 nanoseconds.
```

input = “a bb ccc d eee ff g” (balanced list)

```
Merge Sort execution time: 38394 nanoseconds.
```

input = “a ab abc abcd abcde” (reverse ordered list)

```
Merge Sort execution time: 28841 nanoseconds.
```

Selection Sort:

In Selection Sort algorithm, the best, average, and worst case scenarios are the same. The time complexity of Selection Sort is always O(N ^ 2).

input = “a bb ccc dddd eeee” (ordered list)

```
Selection Sort execution time: 70941 nanoseconds.
```

input = “a bb ccc d eee ff g” (balanced list)

```
Selection Sort execution time: 67559 nanoseconds.
```

input = “a ab abc abcd abcde” (reverse ordered list)

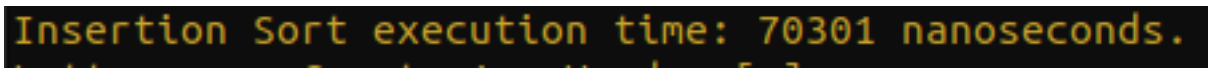
```
Selection Sort execution time: 43323 nanoseconds.
```

Insertion Sort:

In the Insertion Sort algorithm, the best case time complexity is O(n), while in other cases, it is O(n^2)

input = “a bb ccc dddd eeee” (ordered list)

input = “a bb ccc d eee ff g” (balanced list)

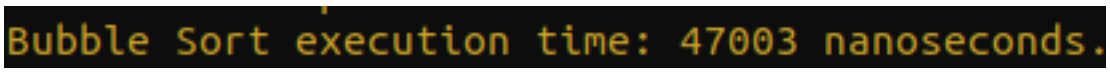


input = “a ab abc abcd abcde” (reverse ordered list)

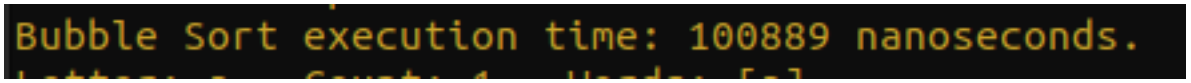
Bubble Sort

In the Bubble Sort algorithm, the best case time complexity is O(n), while in other cases, it is O(n^2)

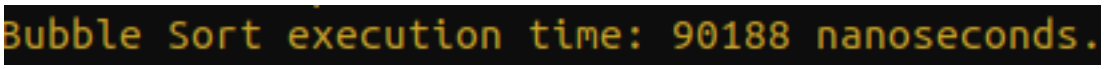
input = “a bb ccc dddd eeee” (ordered list)



input = “a bb ccc d eee ff g” (balanced list)



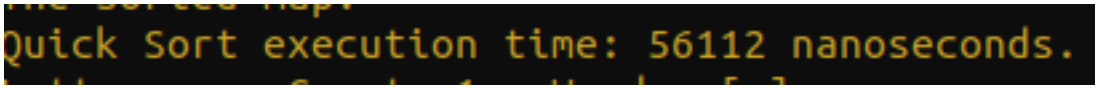
input = “a ab abc abcd abcde” (reverse ordered list)



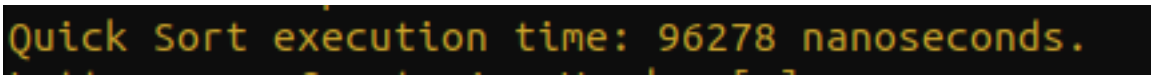
Quick Sort

In the Quick Sort algorithm, the time complexity is O(N log N) for the best case and average case, while it is O(N^2) for the worst case.

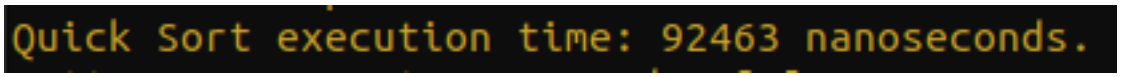
input = “a bb ccc dddd eeee” (ordered list)



input = “a bb ccc d eee ff g” (balanced list)



input = “a ab abc abcd abcde” (reverse ordered list)



Part C

First Input: The fastest performing algorithm for first input was the Insertion Sort algorithm with a runtime of 34486 nanoseconds.

Second Input: The fastest performing algorithm for second input was the Merge Sort algorithm with a runtime of 38394 nanoseconds.

Third Input: The fastest performing algorithm for third input was the Insertion Sort algorithm with a runtime of 18789 nanoseconds

Part D

The reason for this difference is that the sorting algorithms used have different working principles and operations. Each algorithm uses a different approach to sort the elements. Therefore, the input order of the elements can be preserved in some cases, while in other cases, it cannot be preserved.

Merge Sort: Merge Sort is a stable sorting algorithm. It preserves the original order of elements with equal values while sorting. It performs sorting using the steps of dividing the array into halves and merging them. During the merging step, it places elements with equal values in a sorted manner. For example, when we sort an array like [5, 5, 4 ,3, 2, 1] using Merge Sort, the result is [1, 2, 3, 4,5, 5]. Even though the first 5 and the second 5 have the same value, the first occurrence of the number 5 retains its initial position after sorting.

```
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        aux.set(k, Left[i]);
        i++;
    }

    else {
        aux.set(k, Right[j]);
        j++;
    }

    k++;
}
```

In this part, L[i] is compared with R[j], and if they are equal, L[i] is placed first. This ensures that elements with equal values preserve their original order of appearance.

Selection Sort: Selection Sort is an unstable sorting algorithm. This is due to the swap (rearrangement) operations performed by Selection Sort while sorting the elements. Selection Sort proceeds step by step and in each step, it selects the smallest (or largest) element and places it in its correct position. However, when performing the swap operation among elements with the same value, the original order of these elements can be changed.

```
}

private void selectionSort()
{
    int n = aux.size();
    for (int i = 0; i < n - 1; i++)
    {
        int minIndex = i;
        for (int j = i + 1; j < n; j++)
        {
            int count1 = originalMap.map.get(aux.get(j)).count;
            int count2 = originalMap.map.get(aux.get(minIndex)).count;
            if (count1 < count2 || (count1 == count2 && aux.get(j).compareTo(aux.get(minIndex)) < 0))
            {
                minIndex = j;
            }
        }
        swap(i, minIndex);
    }
}
```

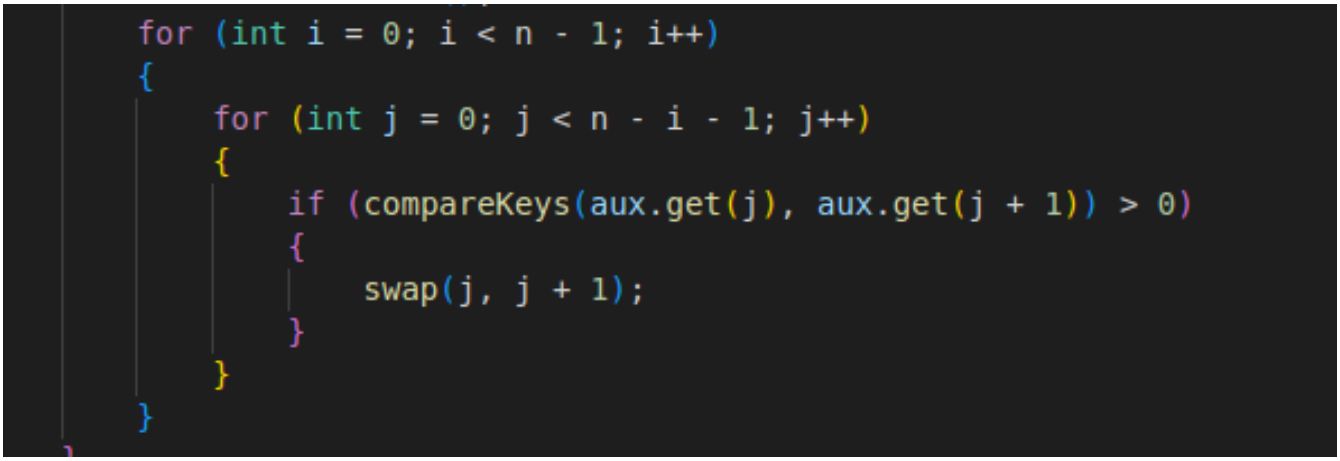
The swap function used in the above selection sort algorithm can change the original order of elements and lead to instability.

Insertion Sort: Insertion Sort is a stable sorting algorithm.The Insertion Sort algorithm iterates through the array and places each element in its appropriate position by iterating through it. During these iterations, when sorting the elements, no swapping is performed between elements with equal values. In other words, there is no priority or change in order among elements with the same value.

```
while (j >= 0 && compareKeys(aux.get(j), key) > 0)
{
    aux.set(j + 1, aux.get(j));
    j--;
}
```

In the compareKeys method, a false value is returned for values that are not equal. Therefore, this ensures that the sorting order is preserved.

Bubble Sort: Bubble Sort is a stable sorting algorithm. It maintains the original order of elements with equal values while comparing and swapping them. Due to this feature, Bubble Sort is considered a stable sorting algorithm.



```
for (int i = 0; i < n - 1; i++)
{
    for (int j = 0; j < n - i - 1; j++)
    {
        if (compareKeys(aux.get(j), aux.get(j + 1)) > 0)
        {
            swap(j, j + 1);
        }
    }
}
```

In the compareKeys method, a false value is returned for values that are not equal. Therefore, this ensures that the sorting order is preserved.

Quick Sort: Quick Sort is an unstable sorting algorithm. It divides the array using a pivot element and during this process, it does not provide a guarantee of maintaining the relative order of elements with equal values.