Data Structures 2

Assignment 1

Implementing Sorting Techniques

Name	ID
Ranime Ahmed Elsayed Shehata.	21010531
Youssef Tarek Hussein Yousry.	21011595
Osama Hosny Hashem Elsayed.	21010238
Youssef Alaa Ahmed Haridy.	21011603
Mohamed Mohamed Mohamed Abdelmeneim Eldesouky.	21011213

Time & Space Complexity Analysis:

1. Simple Sort: “Bubble Sort”

Time Complexity:

- Best Case: **O(n)**

The array is already sorted and no swaps are performed.

- Average Case: **$O(n^2)$**

The array is random or partially sorted.

- Worst Case: **$O(n^2)$**

The array is sorted in the reverse order.

Space Complexity:

In terms of space complexity, since we only swapped the elements with one another and never stored anything, we don’t need any extra space to run the algorithm. This is amazing because it means the space complexity comes out as constant or **O(1)**. This makes it an in-place algorithm that works by modifying the input directly.

Advantages:

- It doesn’t require any additional memory space.
- It’s a stable sorting algorithm. In other words, the elements with the same key value maintain their relative order in the sorted output.

Disadvantages:

- Very slow and inefficient for large data sets.
- It’s a comparison based sorting algorithm. It requires a comparison operator to determine the relative order of elements in the input data set.

2. Efficient Sort: “Merge Sort”

Time Complexity:

It’s **O(n log n)** in all the 3 cases (worst, average and best) as merge sort always divides the array in two halves and takes linear time to merge two halves regardless of the input distribution.

- Best Case: **O(n log n)**
- Average Case: **O(n log n)**
- Worst Case: **O(n log n)**

Space Complexity:

O(n)

N auxiliary space is required for merge sort, since all elements are copied into an auxiliary array.

Advantages:

- It guarantees worst-case performance since it has a worst case time complexity of O(n log n), which means it performs well even on large datasets.
- It’s a stable sorting algorithm. In other words, the elements with the same key value maintain their relative order in the sorted output.
- Merge sort is parallelizable algorithm and can easily take advantage of multiple processors or threads.

Disadvantages:

- Its space complexity as it requires additional memory to store the merged sub-array during the sorting process.
- It’s not in-place sorting algorithm. This can be a disadvantage in applications where memory usage is a concern.
- Not always optimal for small data sets.

3. Non-Comparison Based Sort: “Counting Sort”

Time Complexity:

It’s **O(n+m)** in all the 3 cases (worst, average and best), where:

N: is the size of the input array[].

M: is the size of the count array[] / max possible input.

- Best Case: **O(n+m)**
- Average Case: **O(n+m)**
- Worst Case: **O(n+m)**

Space Complexity:

O(n+m)

N: The output array[].

M: Thecount array[].

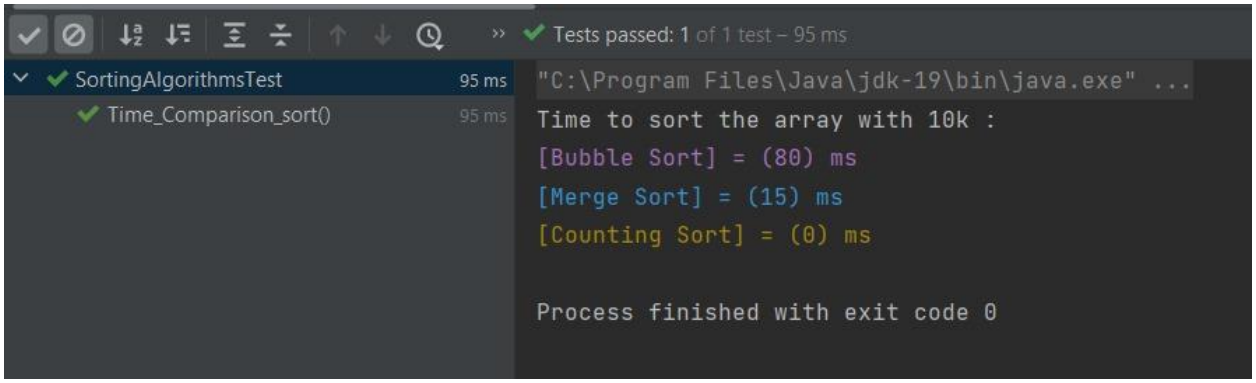
Advantages:

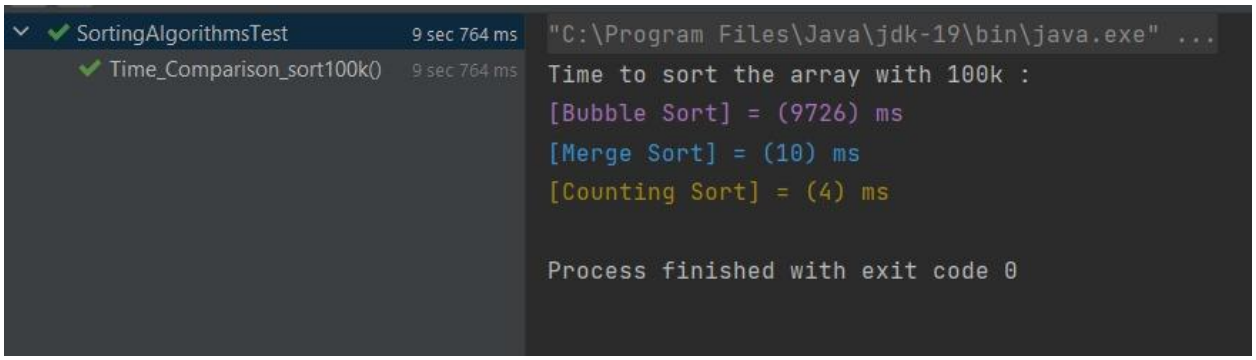
- It performs faster than all comparison-based sorting algorithms if the range of input is of the order of the size of input.
- It’s a stable sorting algorithm. In other words, the elements with the same key value maintain their relative order in the sorted output.

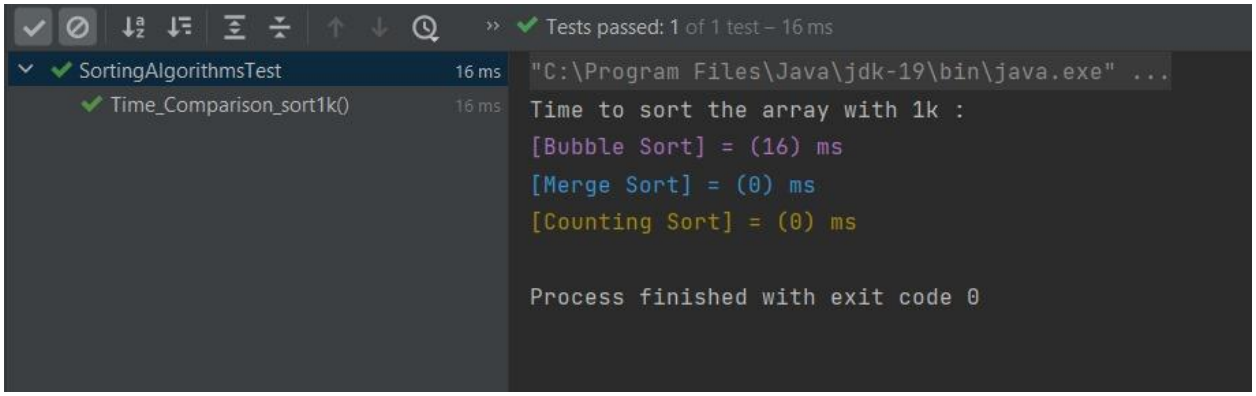
Disadvantages:

- It’s inefficient if the range of values to be sorted is very large.
- It’s not in-place sorting algorithm as it uses extra space for sorting the array elements. This can be a disadvantage in applications where memory usage is a concern.
- It doesn’t work on decimal values.

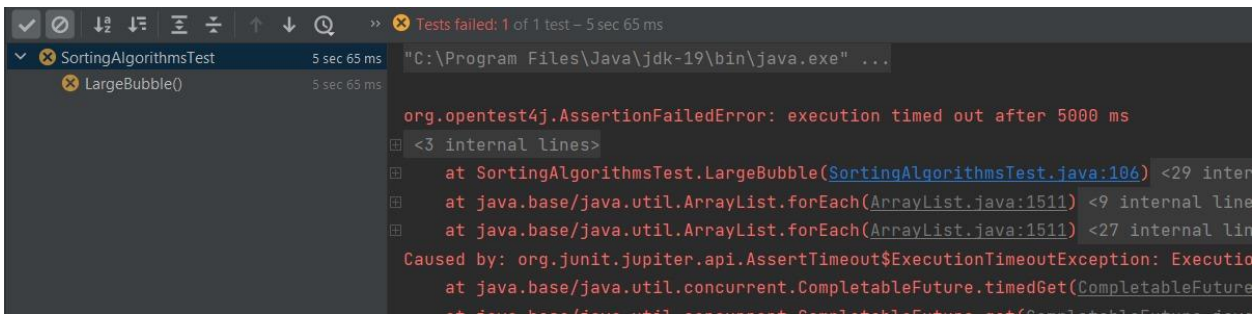
Comparison between them at different array sizes w.r.t the meantime to get the sort of the entire array:



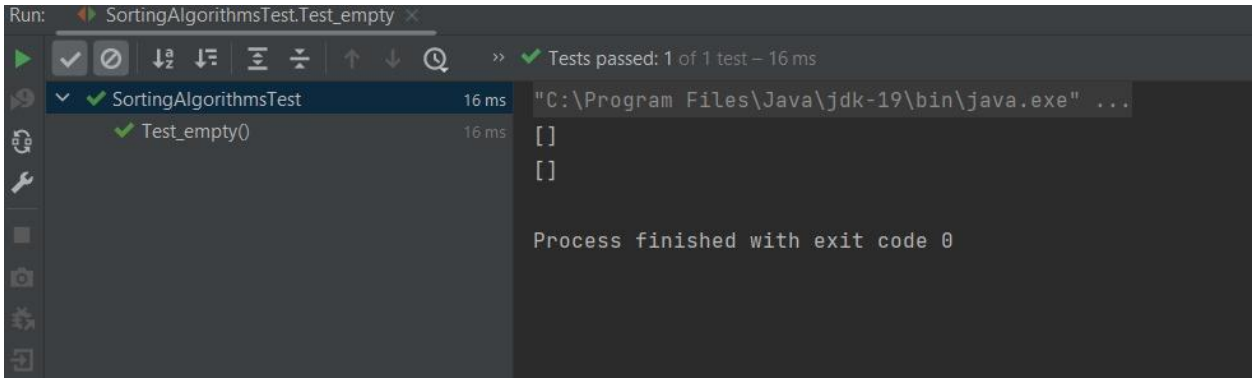




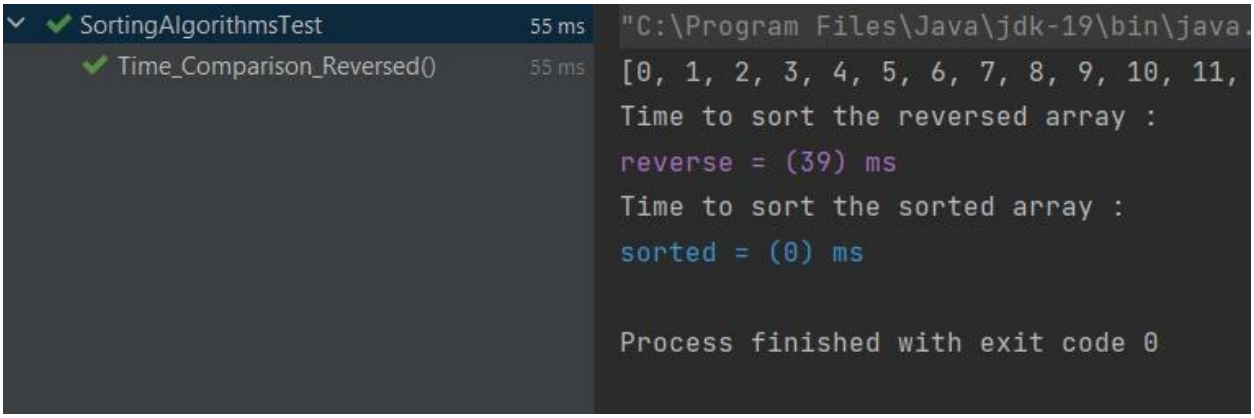
- In case of very large input array size using Bubble Sort:



- In case of empty array:

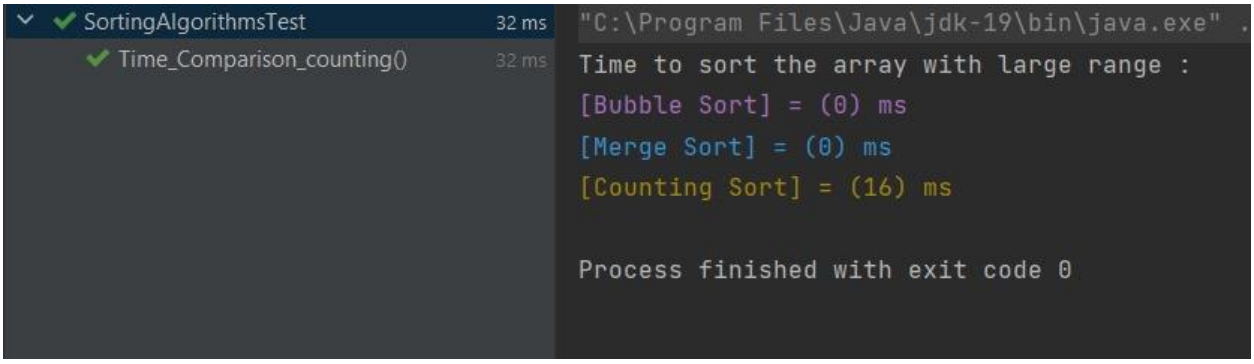


- In case of a reversed array in Bubble Sort:

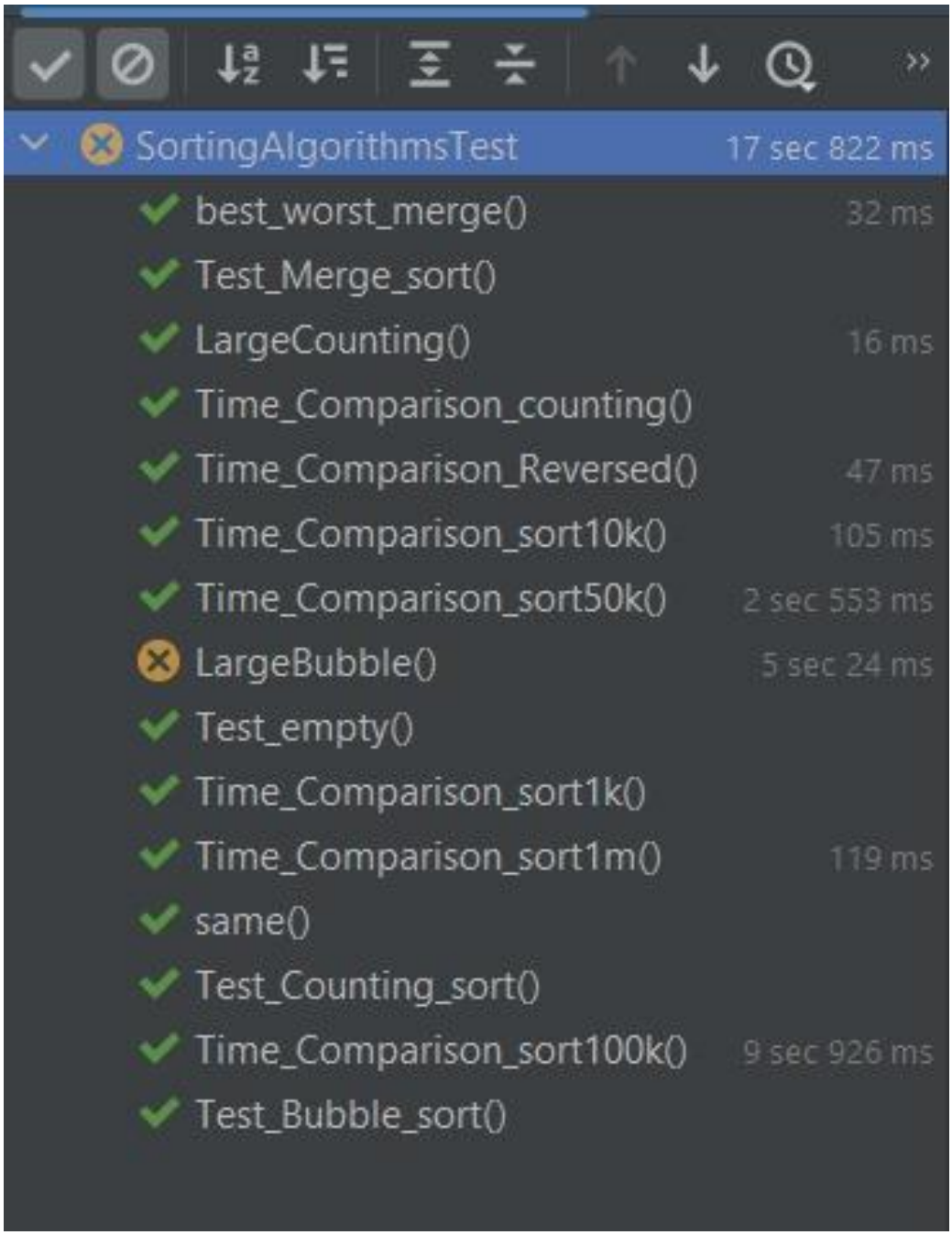


- For an array with large range:

[999999,150000,12000,-9,250,470,1928,999,12,-18,45000,6000,-999999]



JUnit Test:


SortingAlgorithmsTest 17 sec 822 ms
best_worst_merge() 32 ms
Test_Merge_sort()
LargeCounting() 16 ms
Time_Comparison_counting()
Time_Comparison_Reversed() 47 ms
Time_Comparison_sort10k() 105 ms
Time_Comparison_sort50k() 2 sec 553 ms
LargeBubble() 5 sec 24 ms
Test_empty()
Time_Comparison_sort1k()
Time_Comparison_sort1m() 119 ms
same()
Test_Counting_sort()
Time_Comparison_sort100k() 9 sec 926 ms
Test_Bubble_sort()