**DAA Project-1 (Sorting Algorithms)**

Name: Nakka Tulasi

UTA ID: 1001928971

**Introduction:**

The Goal of this project is to implement and compare the 7 given algorithms and also if possible to develop a GUI.

Algorithms Implemented:

Merger Sort

Heap Sort

Quick Sort

Quick Sort Using 3 Medians

Insertion Sort

Selection Sort

Bubble Sort.

**Environmental Setup:**

I have used **Python** to implement the Algorithms and for developing GUI I used the **Tkinter GUI Interface** that is provided for python also for plotting graph matplotlib is used.

**Running Program:**

* Redirect to project folder “Sorting\_Algorithms”
* Run in command prompt -> “python AlgorithmsUI.py”
* Install matplotlib libraries if not installed before.

The Main Data structure used is **arrays** to sort all algorithms.

**Algorithms:**

**1. Merger Sort:** It is a divide and conquer algorithm which divides the array into sub-arrays until every single element is sorted and then merges those sub-arrays to compute the final sorted array.

The Time Complexity of Merge Sort is O(N log N) for best, average, and worst case.

**2. Heap Sort:** Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list. Heapsort is the in-place sorting algorithm.

The Time complexity of Heap Sort is also O(N log N) for all the best, average, and worst case.

**3. Quick Sort:** Quick sort is also a divide-and-conquer algorithm that works based on the pivot element and partitions the array in such a way that all the elements less than the pivot will be on the left and elements greater than the pivot are to the right of pivot.

The Time complexity for quick sort is O(N log N) for the best and average case but O(N ^ 2) for the worst case.

\*\* I used the pivot as the start element while implementing the algorithm.

**4. Quick Sort using 3 Medians:** It works similarly to Quick sort but here the pivot item is selected as the median between the first element, the last element, and the middle element. In the cases of already sorted lists, this should take the middle element as the pivot and hence reducing the inefficiency found in normal quicksort.

**5. Insertion Sort:** Insertion sort is based on finding the correct position of the element and placing it. It means that it is virtually split into a sorted and unsorted part where values from the unsorted part are picked and placed at the correct position in sorted part. It is an in-place algorithm.

Time Complexity for the best case is O(N) whereas for the average and worst case it is O(N ^2).

**6. Selection Sort:** This algorithm works based on repeatedly sorting array by finding the minimum element from an unsorted array and placing it at the beginning. It is an in-place comparison algorithm.

The time complexity for all three cases the best, average, and worst cases is O( N^ 2).

**7.Bubble Sort:** Bubble sort is the basic sorting algorithm that works based on repeatedly swapping the adjacent elements if they are not placed in correct position. It doesn’t work well for large arrays.

The Time complexity for best case is O(N) and for average and worst case it is O(N^2).

**GUI Interface:**

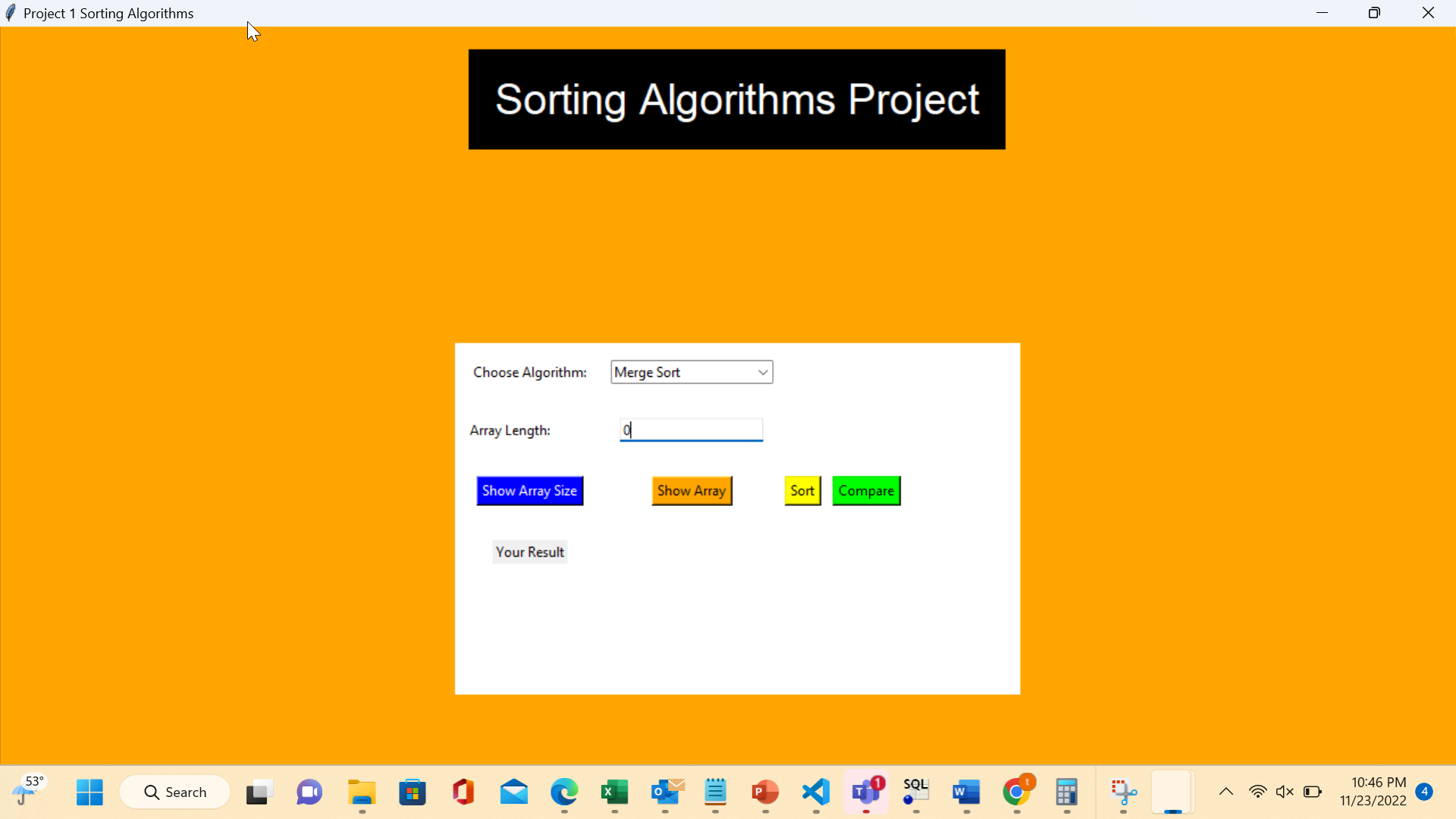
I have used the Tkinter framework to develop a basic UI to run algorithms and generate the run time complexity for each algorithm based on array length.

Functionalities of UI:

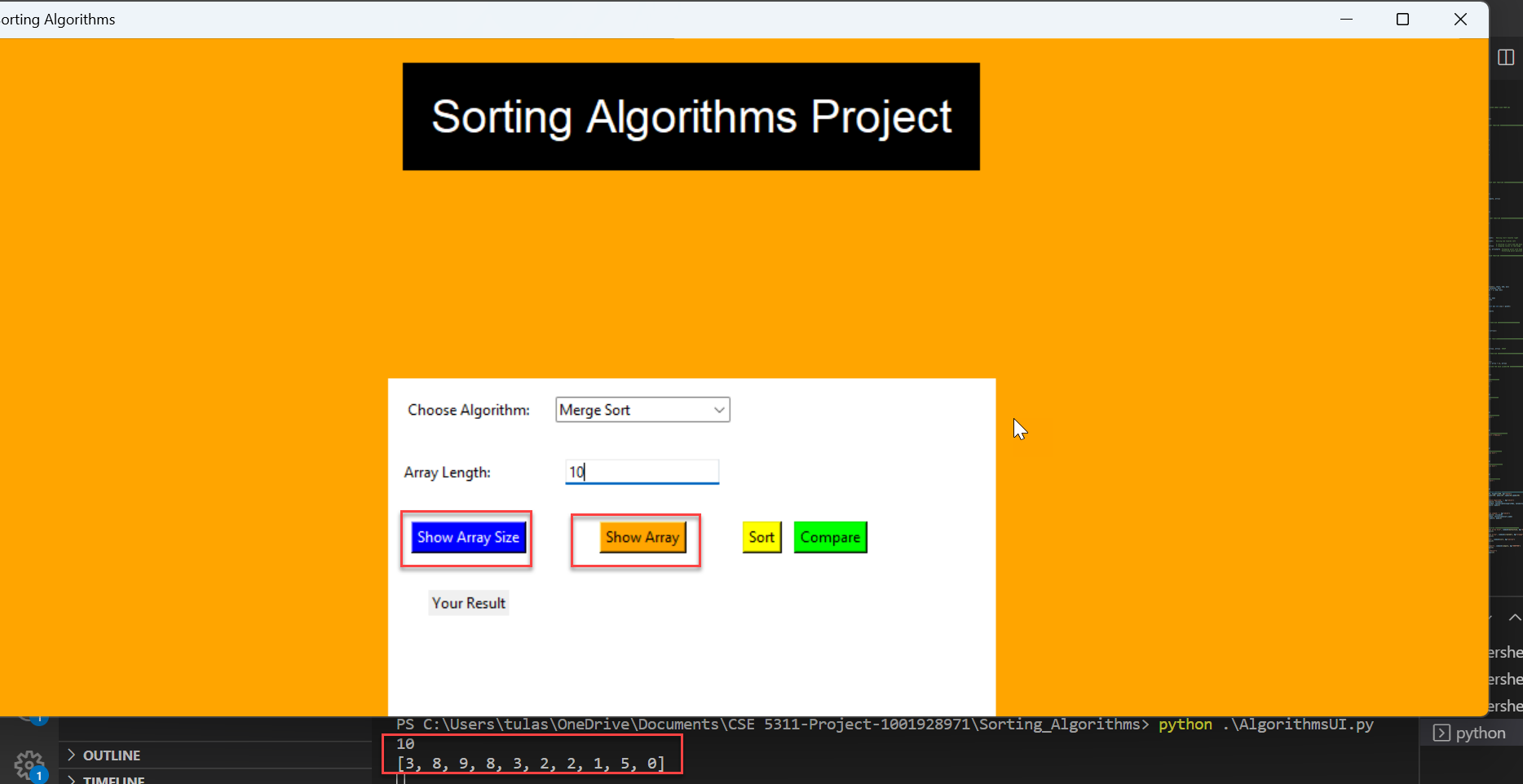
* In the developed widget first box asks for the choice of algorithm
* The second box is for giving the array size to generate the random array of that specified length.
* The button “Show Array Size” will display the given array length on the Command prompt (This I kept only for reference to check whether the array length input is correctly functioning or not).
* As it is not better option to display the generated random array on UI screen so I kept a button “Show Array” which On Click shows the array on the Command prompt.
* There is another button “Sort” which will sort the particular selected algorithm and display the runtime on the UI screen.
* The button “Compare” will generate the graph by comparing all the algorithms for array size ranging from 1000 to 9000. I have changed the input array size and generated the graphs for various lengths which I am attaching below.

**Screen Shots:**

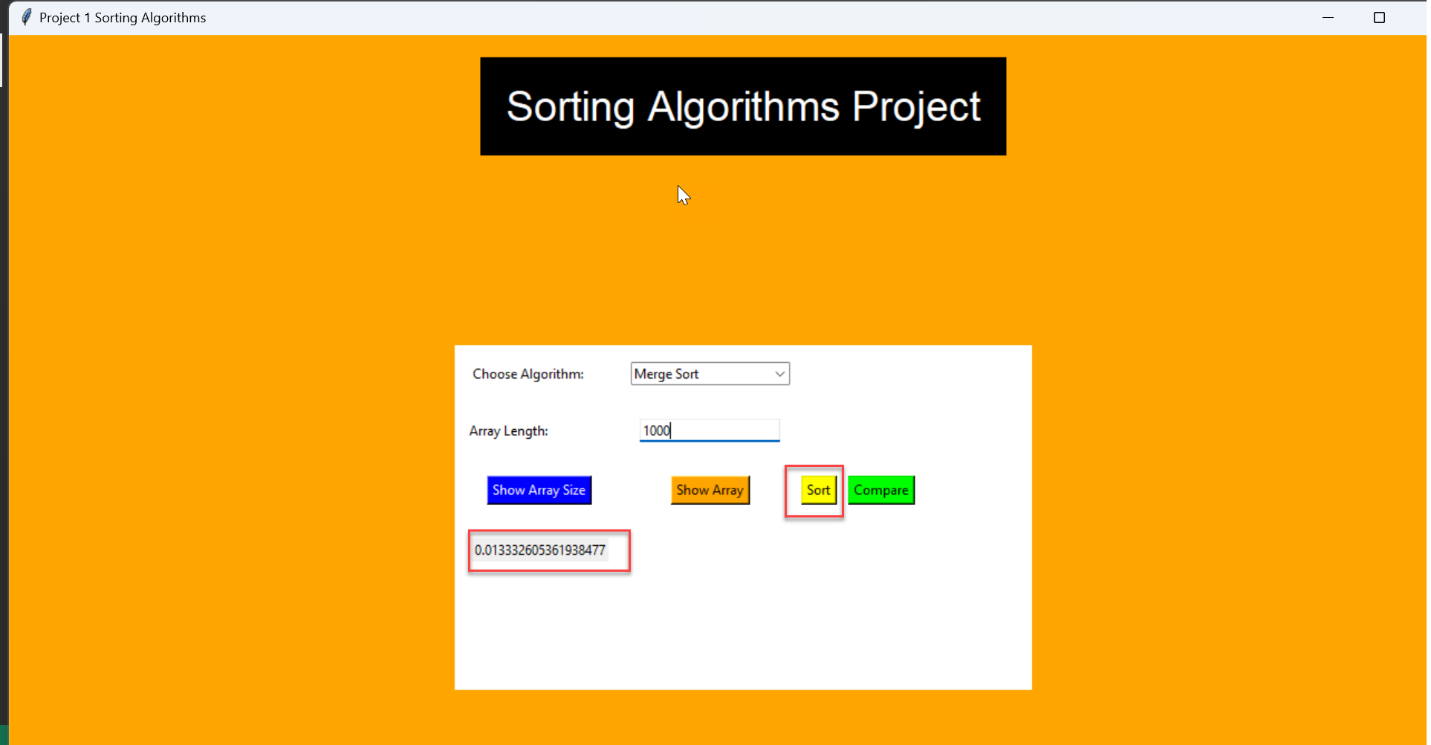
**UI Screen:**



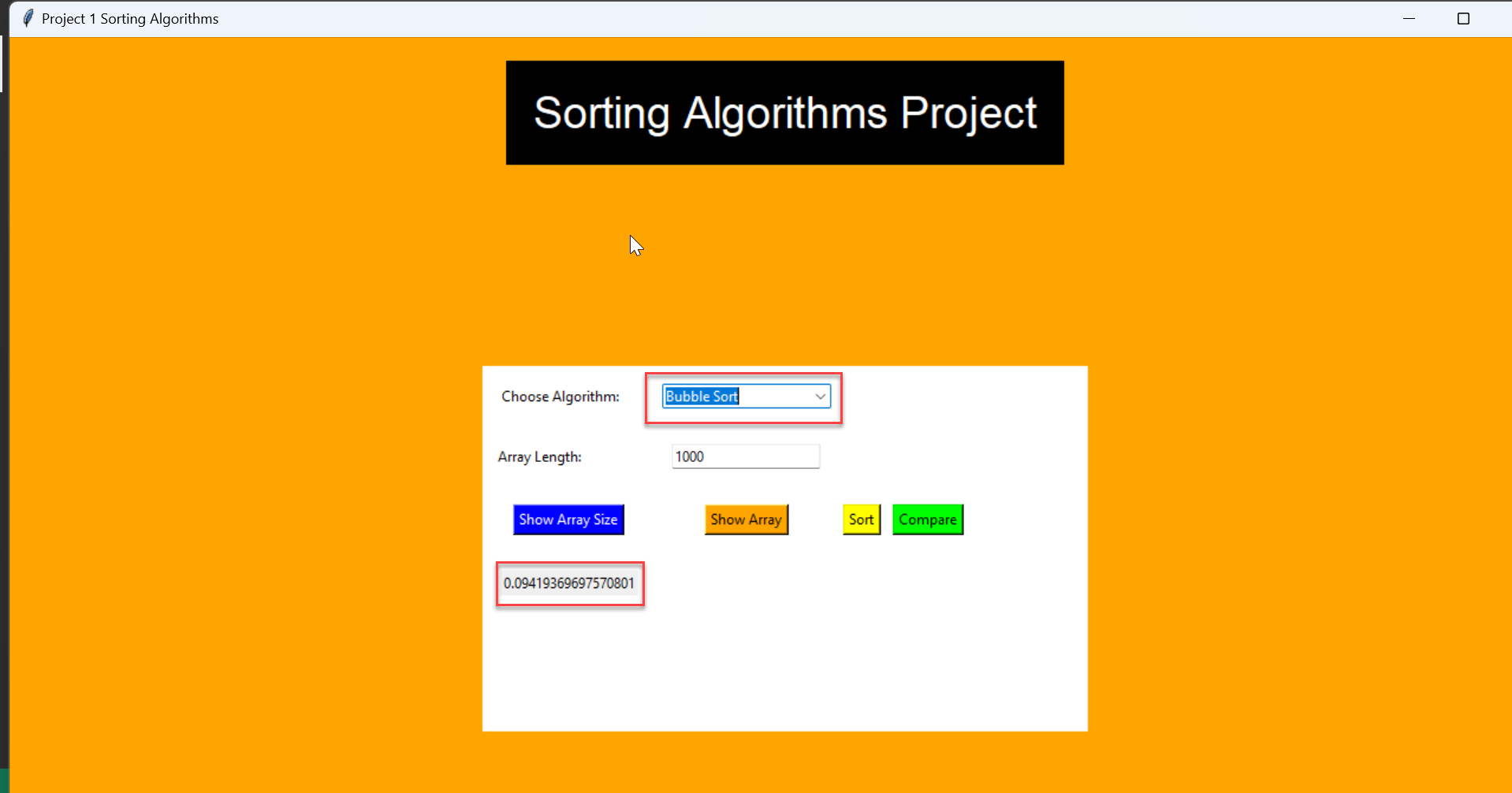
1. Showing Functionalities of Buttons 1 and 2 ( Show Array size and Show Array)



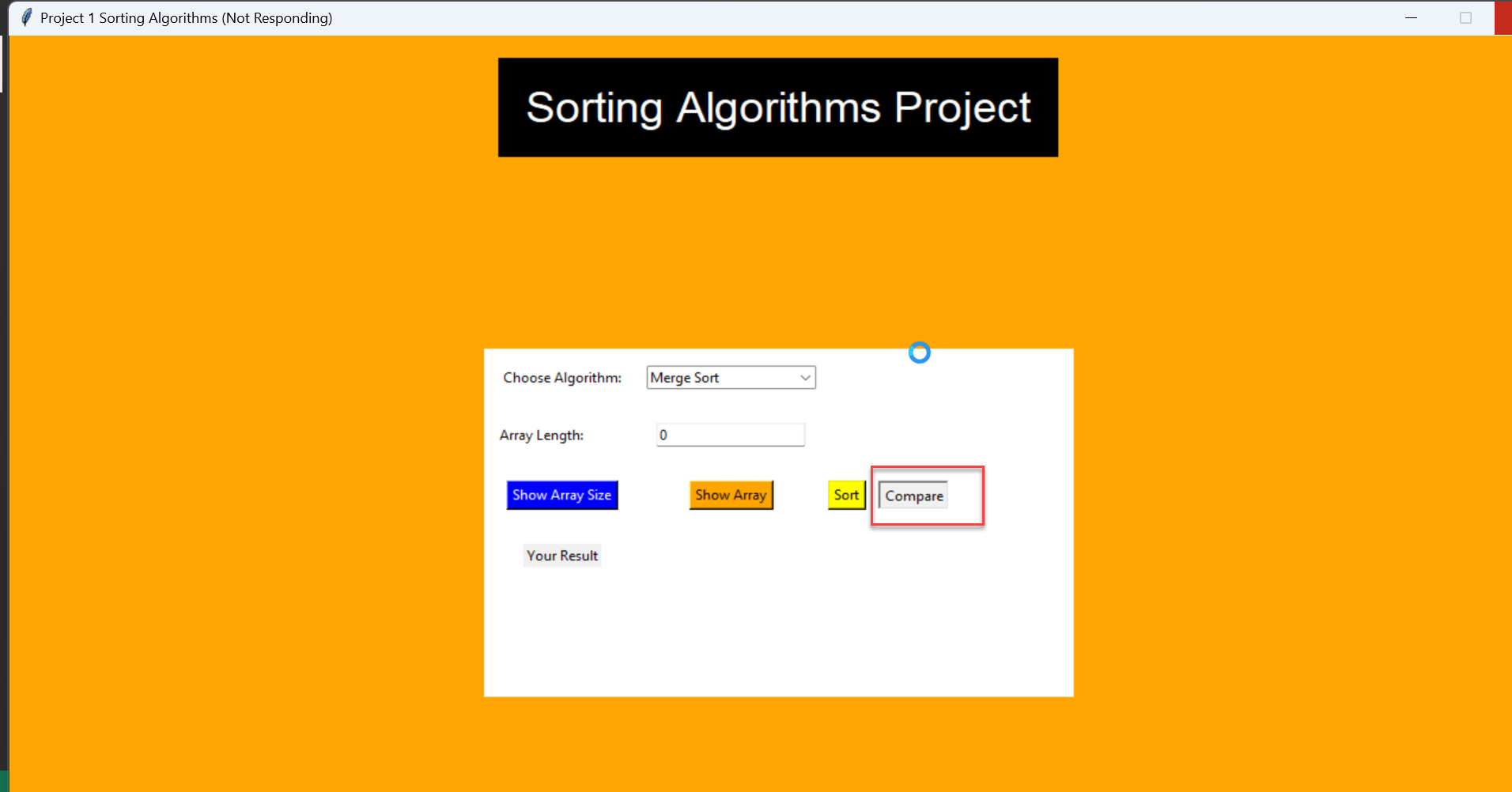
1. Showing the functionality of sort button (Eg: Merge sort for Array size 1000)



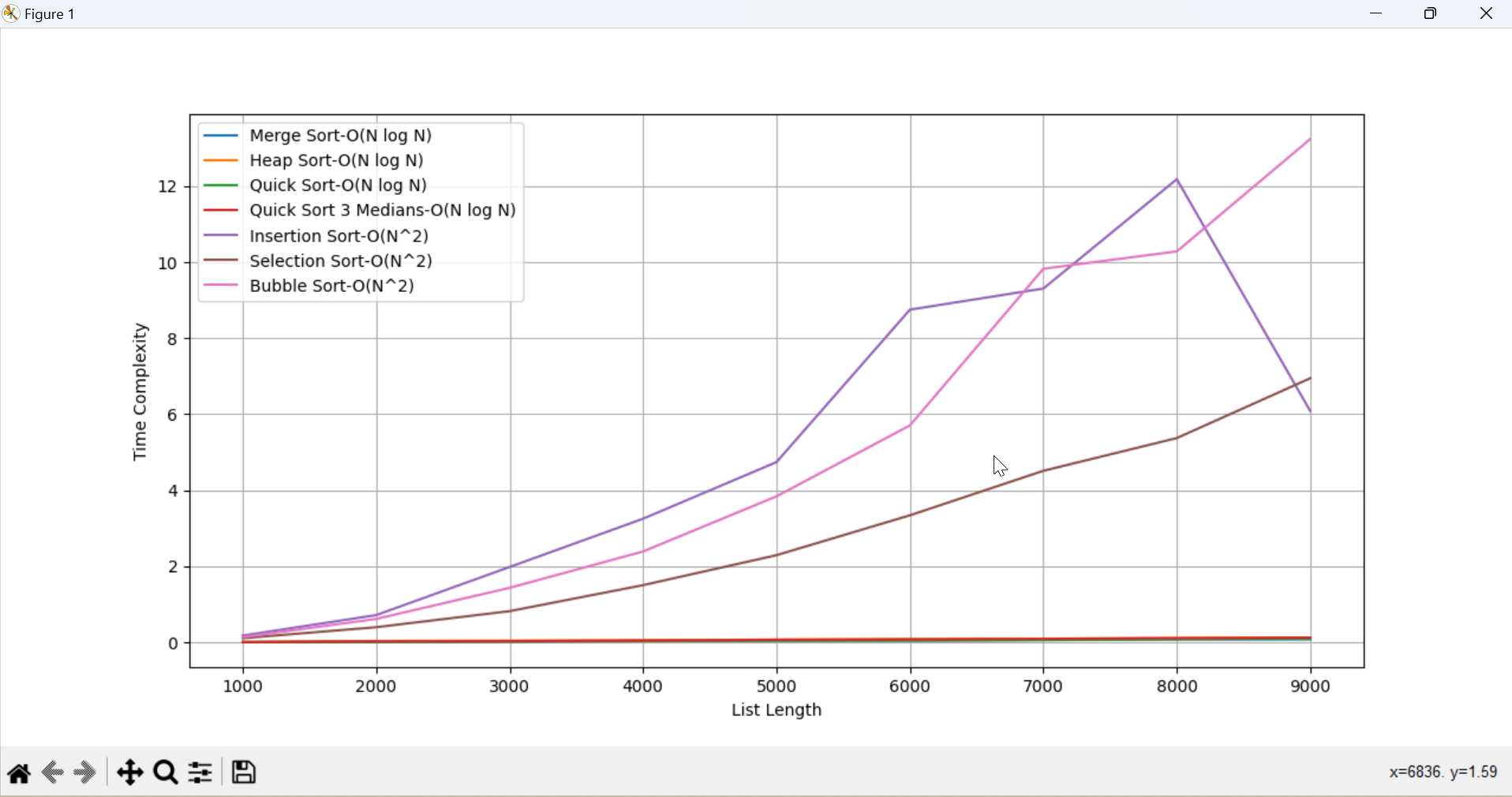
Similarly another example for Bubble sort



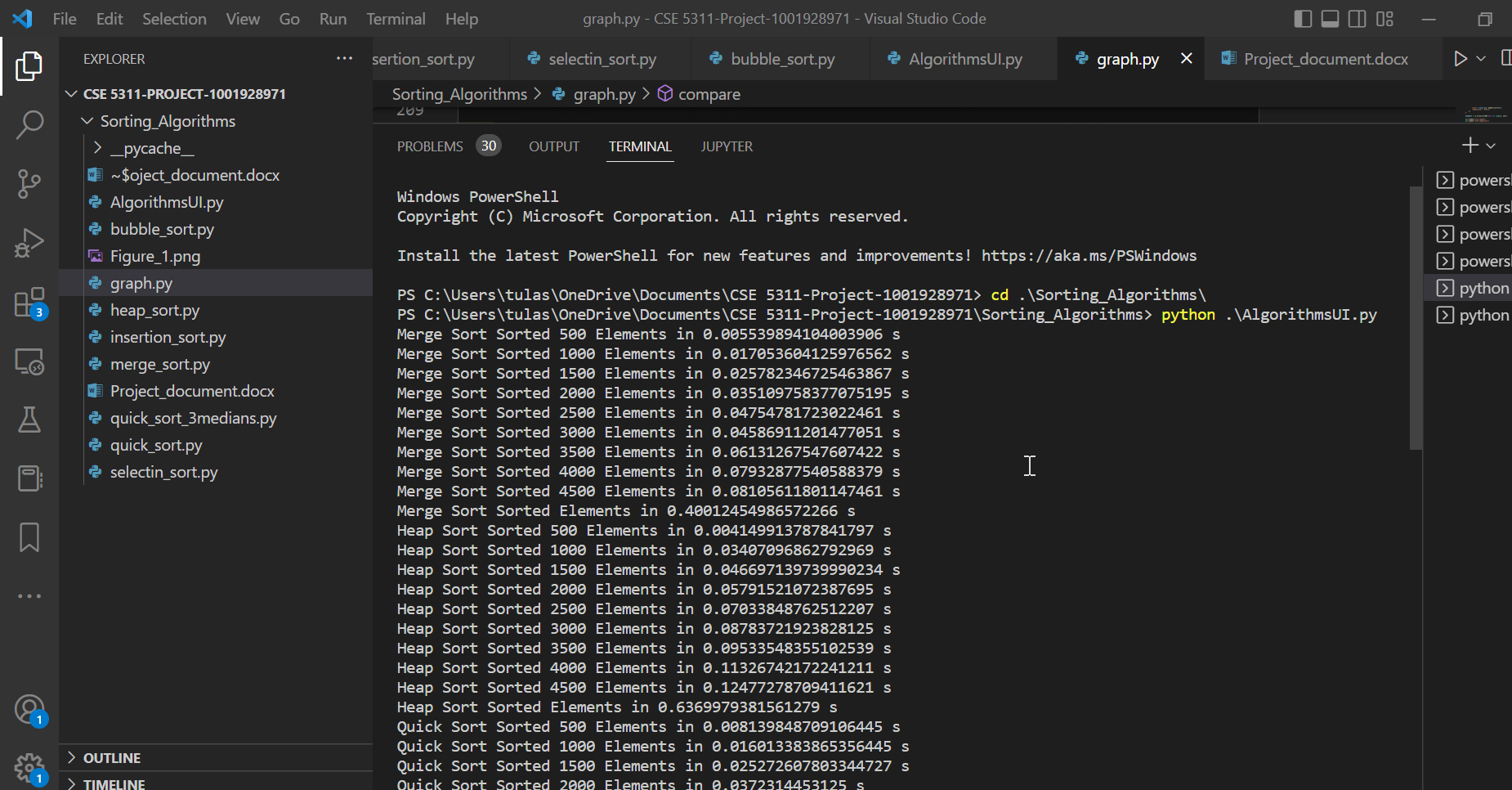
1. Showing Compare button functionality

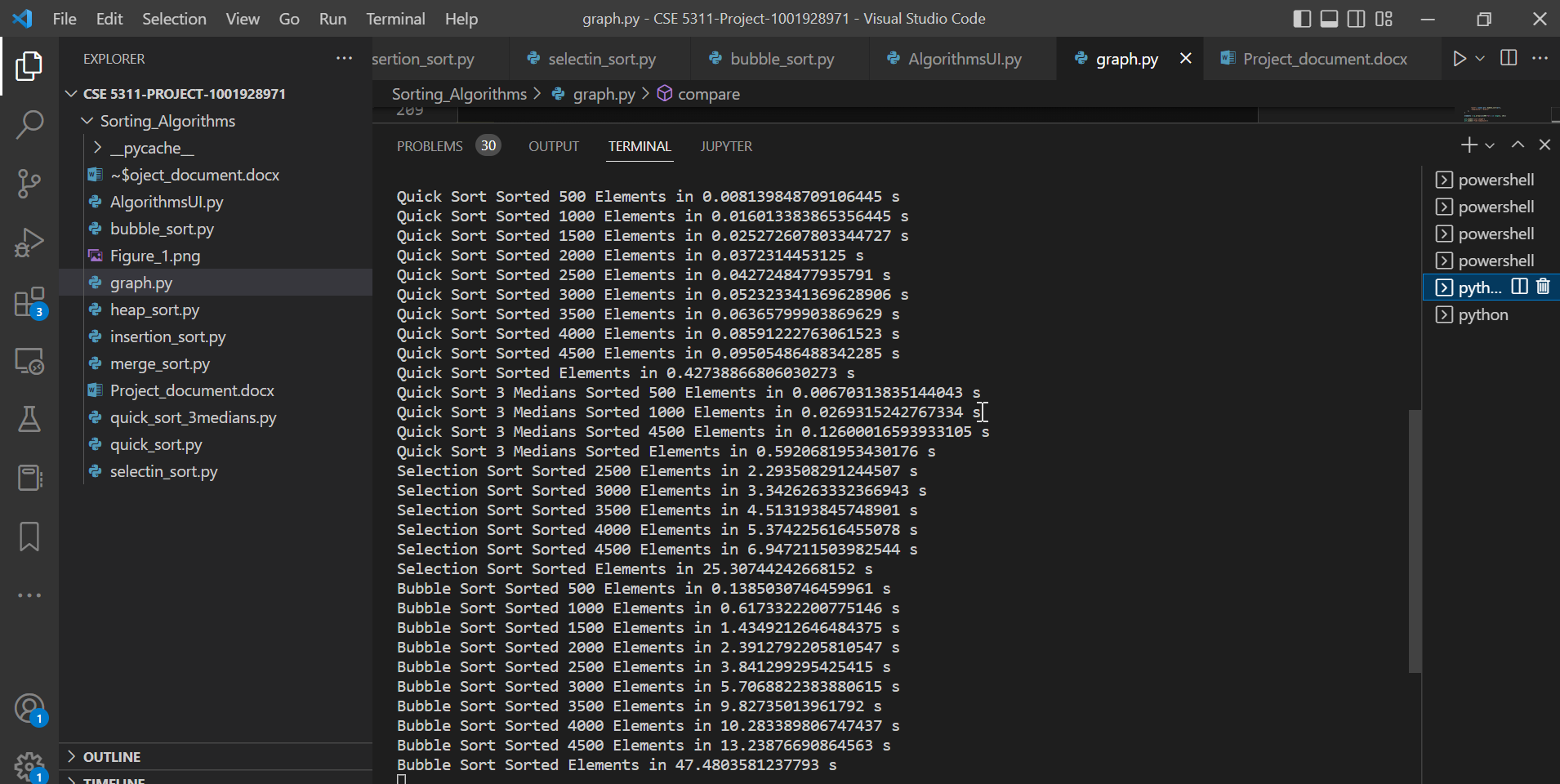


**Comparison of Data Range from 500 to 4500 elements :**



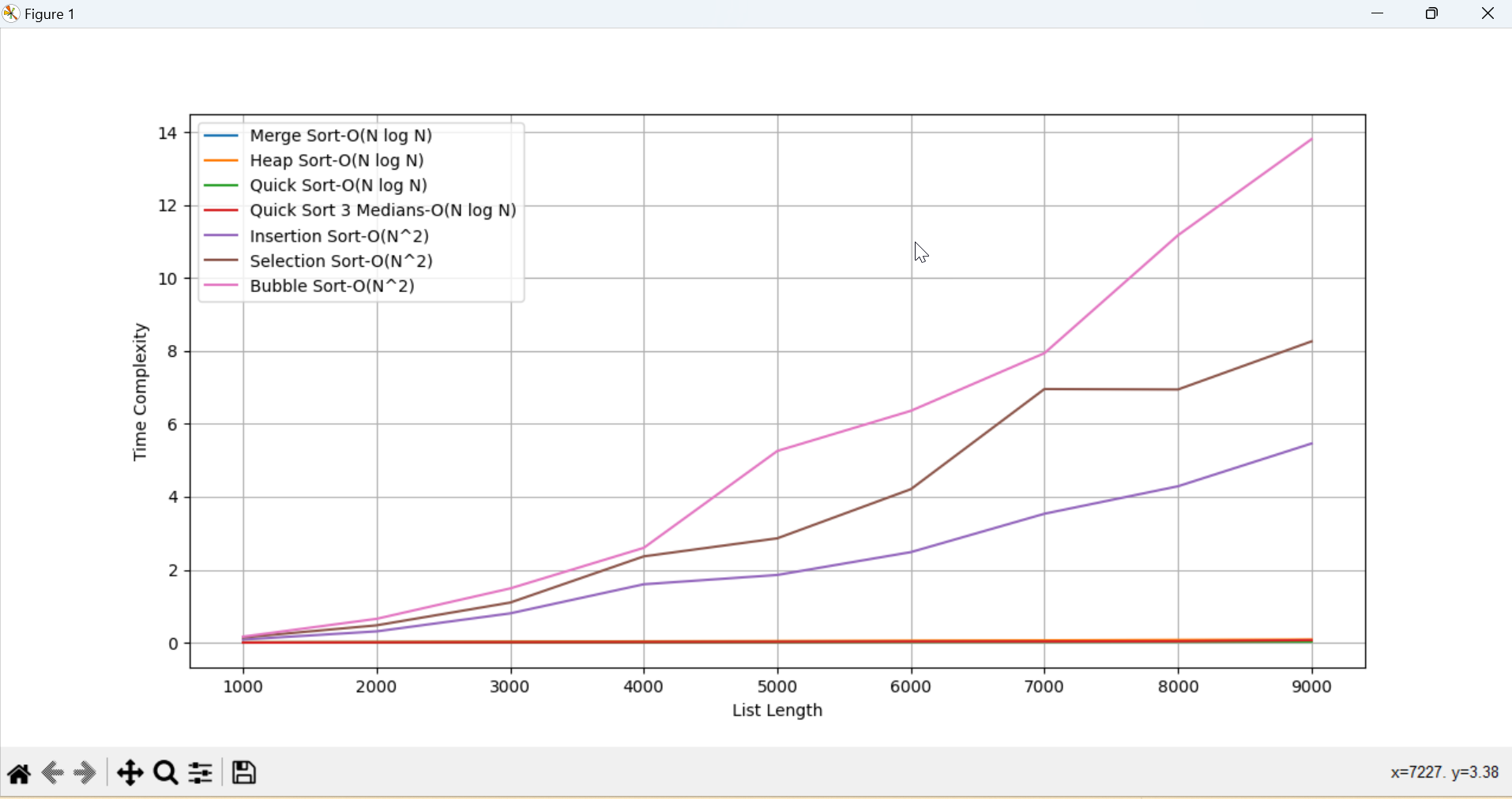
**Output for the graph above:**



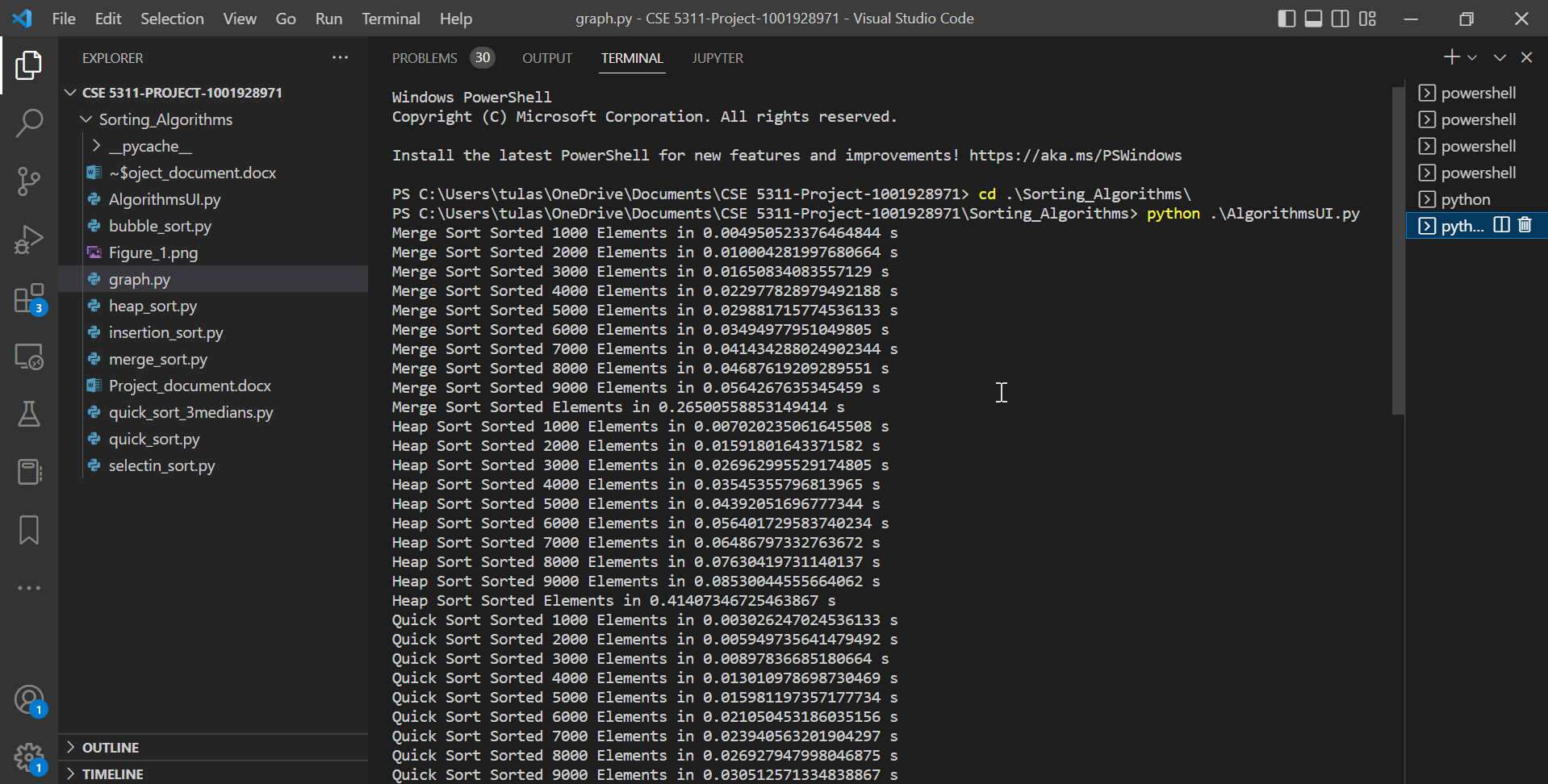


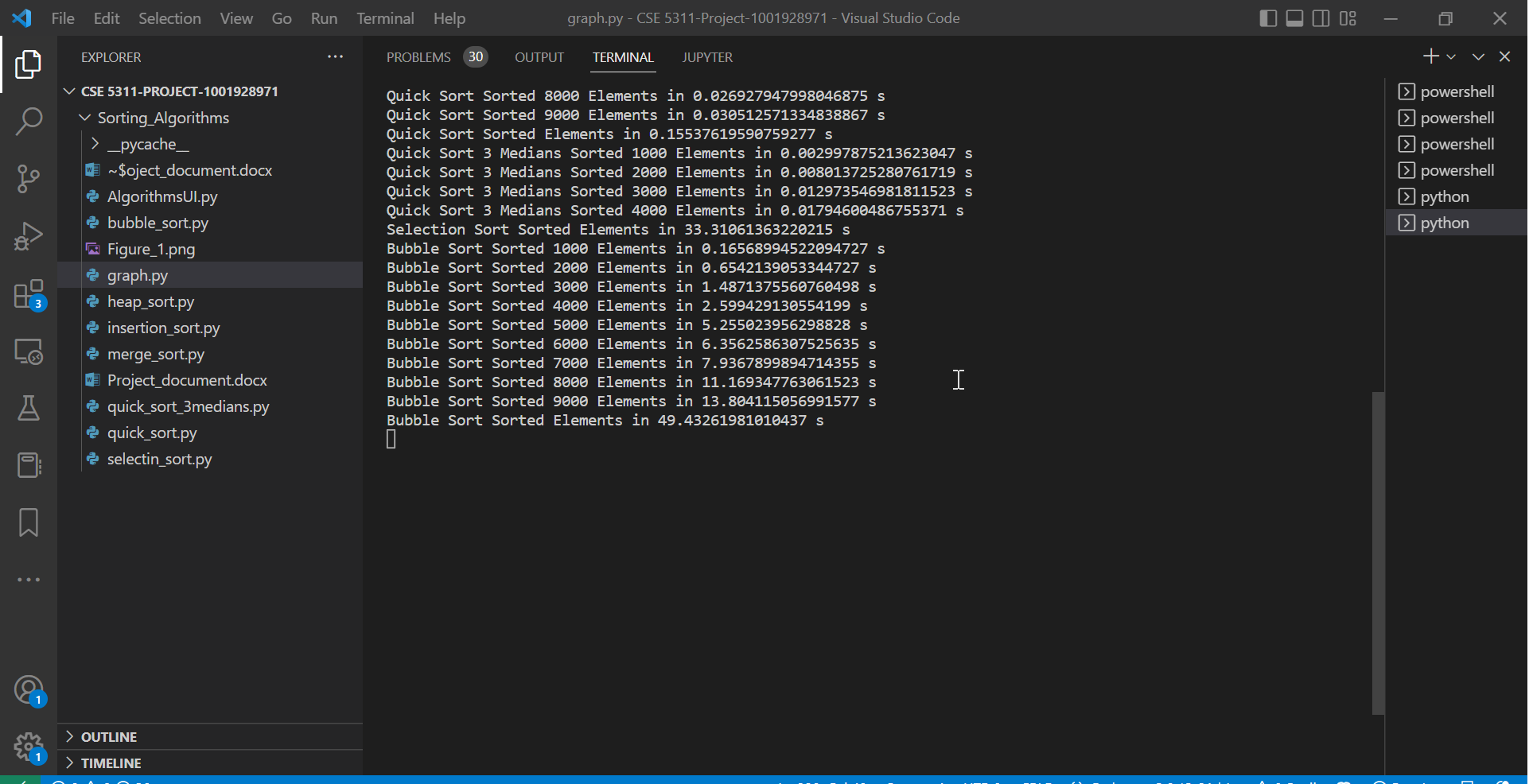
* The output describes that the **run time complexities** of different algorithms for each iteration increasing elements with 500 range i.e for array size of **500, 1000, 1500, ……………, 4500** (10 iterations).
* From above graph we can observe that among all the 7 algorithms the time taken by **bubble sort** to sort array length of 4500 elements has taken **more time than other algorithms** (13.23876690864563) and **Merge sort gave the best results** which took very less time than other algorithms to sort 4500 elements.

**Comparison of Data Range from 1000 to 9000 elements:**



**Output for above graph:**





* The above output describes the **run time complexities of algorithms ranging from 1000 to 9000 elements**.
* The above graph plots statistics that again **bubble sort performance is poor** compared to other algorithms as complexity is ~13.80411… for 9000 elements and the **best algorithm** for 9000 elements was **Quick sort** with complexity ~ 0.03051…