Experiment 4(B)

Student Name: Tanmaya Kumar Pani **UID: 22BCS12986**
Branch: BE-CSE **Section/Group: IOT-613B**
Semester: 5 **Date of Performance: 13/8/24**
Subject Name: Advanced Programming Lab-1 **Subject Code: 22CSP-314**

1. Title: Quick Sort 1 - Partition

2. Objective:

The previous challenges covered Insertion Sort, which is a simple and intuitive sorting algorithm with a running time of . In these next few challenges, we're covering a *divide-and-conquer* algorithm called Quick Sort (also known as *Partition Sort*). This challenge is a modified version of the algorithm that only addresses partitioning.

3. Algorithm:

a) **Input**: An unsorted vector arr of integers.
b) **Choose Pivot**:
- Select the first element of the array as the pivot.

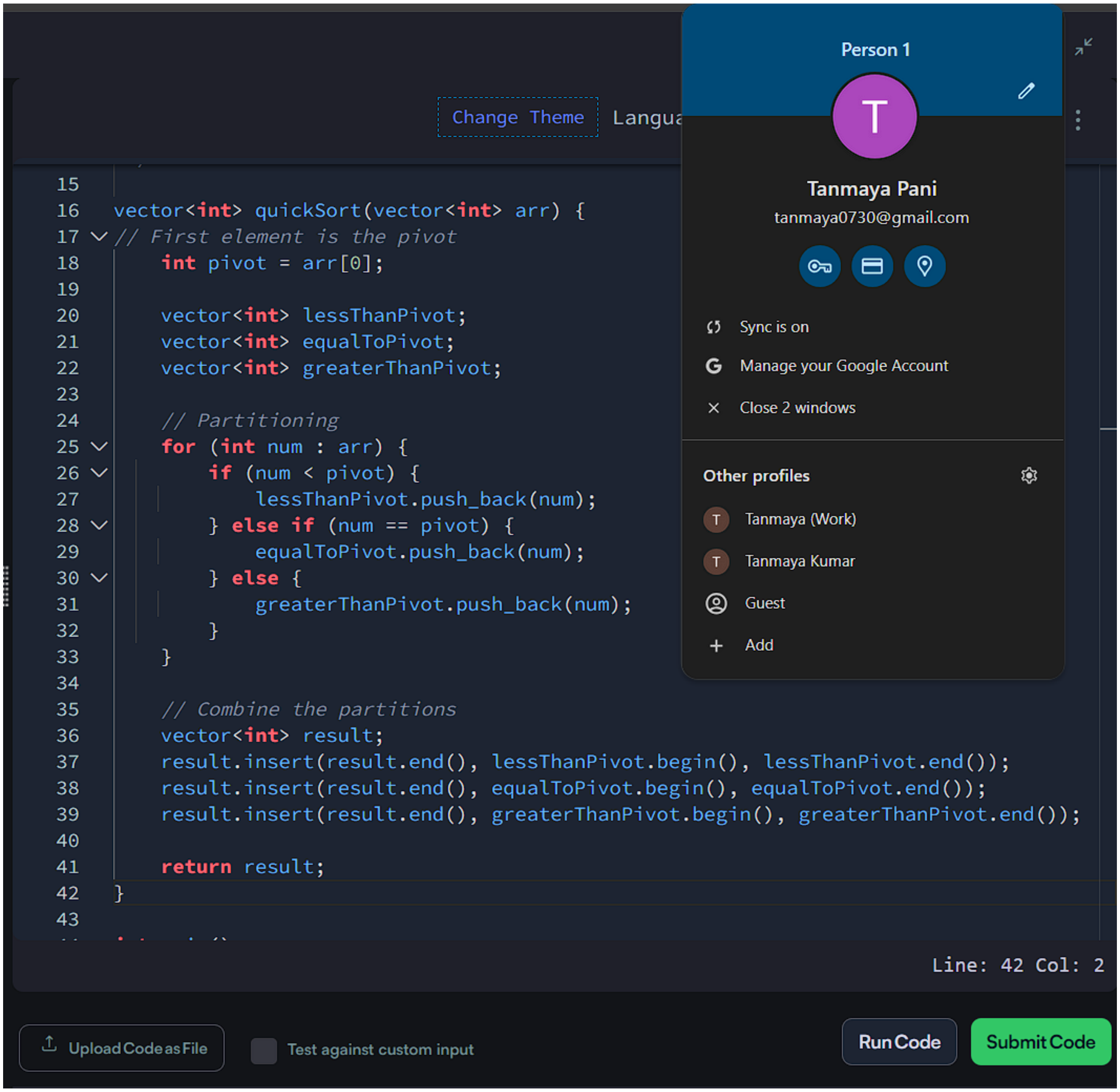
c) **Partitioning**:
- Create three empty vectors: lessThanPivot, equalToPivot, and greaterThanPivot.
- Iterate through each element in the array:
  - If the element is less than the pivot, add it to lessThanPivot.
  - If the element equals the pivot, add it to equalToPivot.
  - If the element is greater than the pivot, add it to greaterThanPivot.

d) **Recursive Sorting**:
- Recursively apply the same procedure to lessThanPivot and greaterThanPivot (not shown in your code).
- **Base Case**: When the vector has 1 or 0 elements, it is already sorted.

e) **Combine**:
- Concatenate lessThanPivot, equalToPivot, and greaterThanPivot into a single vector.

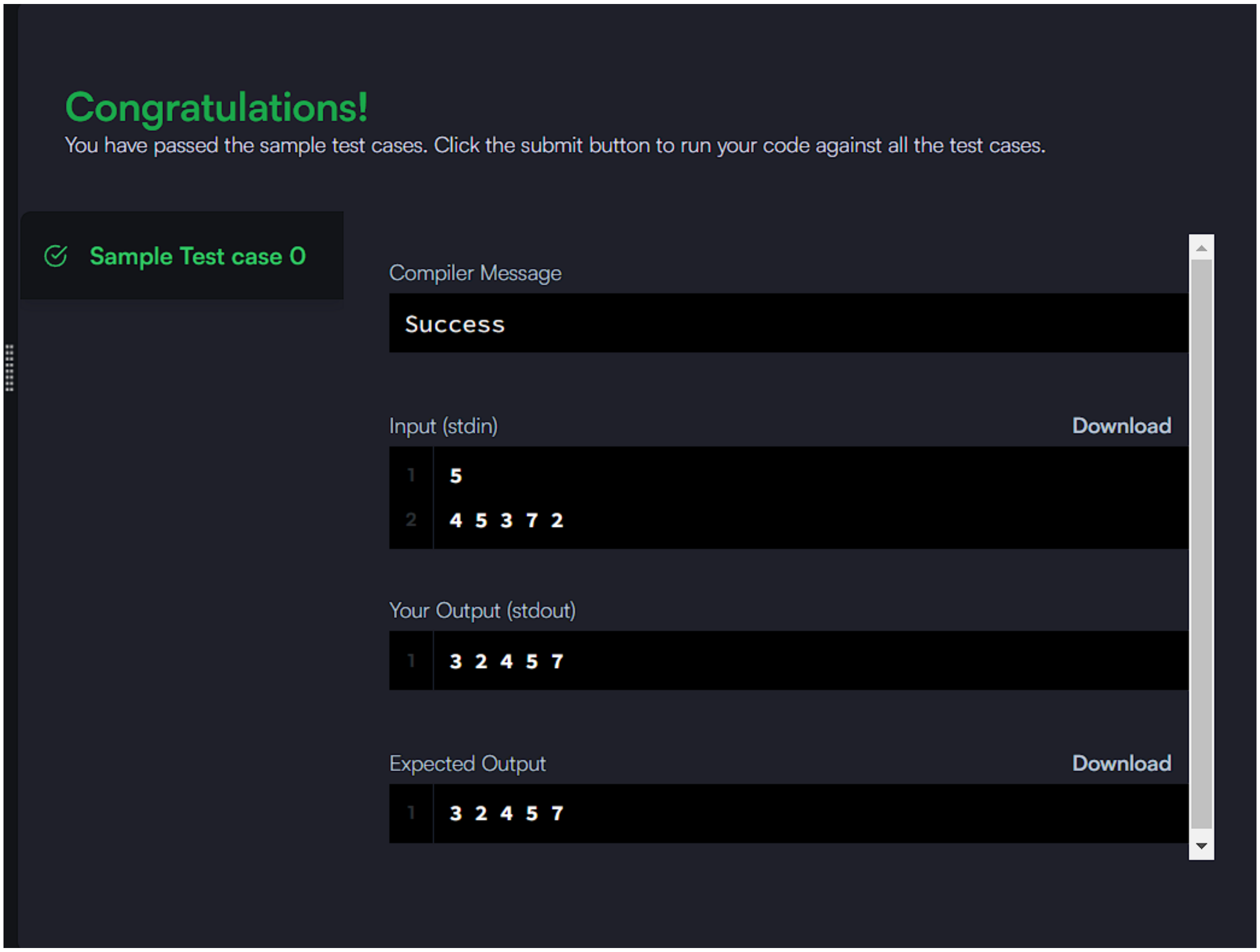
f) **Return**:
- Return the sorted vector.

4. Implementation/Code:



```
vector<int> quickSort(vector<int> arr) {
// First element is the pivot
    int pivot = arr[0];

    vector<int> lessThanPivot;
    vector<int> equalToPivot;
    vector<int> greaterThanPivot;

    // Partitioning
    for (int num : arr) {
        if (num < pivot) {
            lessThanPivot.push_back(num);
        } else if (num == pivot) {
            equalToPivot.push_back(num);
        } else {
            greaterThanPivot.push_back(num);
        }
    }

    // Combine the partitions
    vector<int> result;
    result.insert(result.end(), lessThanPivot.begin(), lessThanPivot.end());
    result.insert(result.end(), equalToPivot.begin(), equalToPivot.end());
    result.insert(result.end(), greaterThanPivot.begin(), greaterThanPivot.end());

    return result;
}
```

5. Output:



Congratulations!

You have passed the sample test cases. Click the submit button to run your code against all the test cases.

Compiler Message

```
Success
```

Input (stdin)

```
5
4 5 3 7 2
```

Your Output (stdout)

```
3 2 4 5 7
```

Expected Output

```
3 2 4 5 7
```

6. Learning Outcomes:

• **Understanding Divide and Conquer**: Learn how QuickSort uses the divide-and-conquer strategy to recursively break down the problem of sorting into smaller subproblems.
• **Efficient Partitioning**: Gain insights into partitioning techniques that separate elements based on their relationship to a pivot, facilitating faster sorting.
• **Handling Worst-Case Scenarios**: Recognize how improper pivot selection can affect performance and explore ways to mitigate this issue (e.g., using random pivot selection or median-of-three).

7. Time Complexity: O (n logn)

8. Space Complexity: O(n)