Experiment 8

Student Name: Deepak Agrawal   **UID:22BCS12192**

Branch: CSE   **Section/Group: NTPP-IOT627/A**

Semester: 6th   **Date of Performance:28-3-25**

Subject Name :AP Lab-2   **Subject Code:** 22CSP-351

Problem-1

Aim: Max Units on a Truck

Objective: The objective of the code is to determine the maximum total number of units that can be loaded onto a truck, given the constraints of truck capacity and different box types, each with a specific number of units. It uses a greedy algorithm to prioritize boxes with the highest units per box to optimize the total units loaded.

Implementation/Code:
```
class Solution {
public:
    int maximumUnits(vector<vector<int>>& boxTypes, int truckSize) {
        sort(boxTypes.begin(), boxTypes.end(), [](vector<int>& a, vector<int>& b) {
            return a[1] > b[1];
        });

        int totalUnits = 0;

        for (auto& box : boxTypes) {
            int numBoxes = box[0];
            int unitsPerBox = box[1];

            if (truckSize >= numBoxes) {
                totalUnits += numBoxes * unitsPerBox;
                truckSize -= numBoxes;
            } else {
                totalUnits += truckSize * unitsPerBox;
                break;
            }
        }
        return totalUnits;
    }
};
```
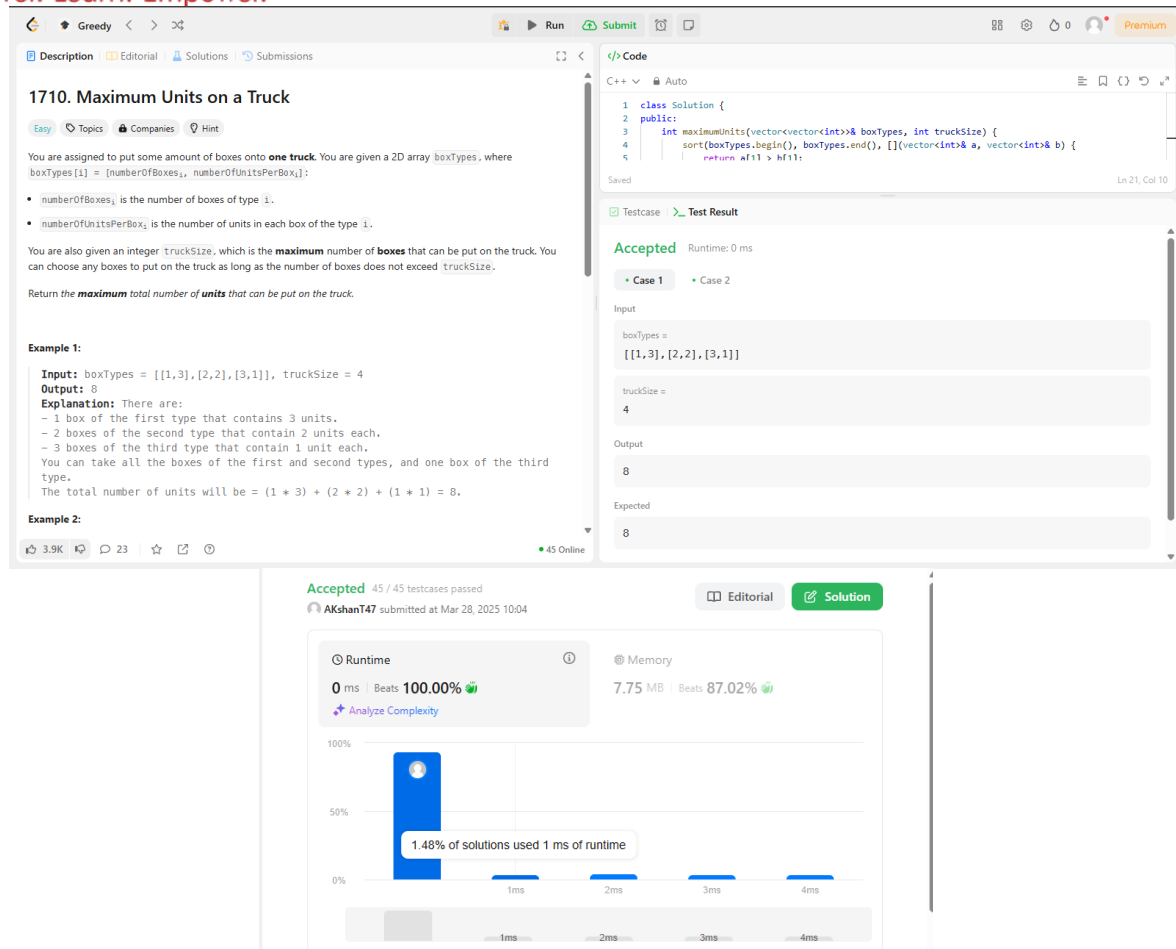
Output:

Discover. Learn. Empower.

Figure1

Learning Outcome:

- Understanding how to use a **greedy algorithm** to optimize outcomes by prioritizing items based on specific criteria (units per box).

- Learning how to efficiently handle constraints, such as truck capacity, while iterating through and selecting items.

- Improving the ability to sort data and apply conditional logic to make decisions during traversal.

- Enhancing problem-solving skills by developing efficient solutions with $O(n \log n)$ time complexity and $O(1)$ space complexity for real-world optimization problems.

Problem-2

Aim: Min Operations to Make Array Increasing

Objective: The objective of this code is to calculate the minimum number of operations required to make a given integer array strictly increasing. This is achieved by incrementing the necessary elements of the array while ensuring that each element is strictly greater than the previous one, following the given constraints.

Implementation/Code:
```cpp
class Solution {
public:
    int minOperations(vector<int>& nums) {
        int operations = 0;

        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] <= nums[i - 1]) {
                int increment = nums[i - 1] + 1 - nums[i];
                operations += increment;
                nums[i] += increment;
            }
        }
        return operations;
    }
};
```
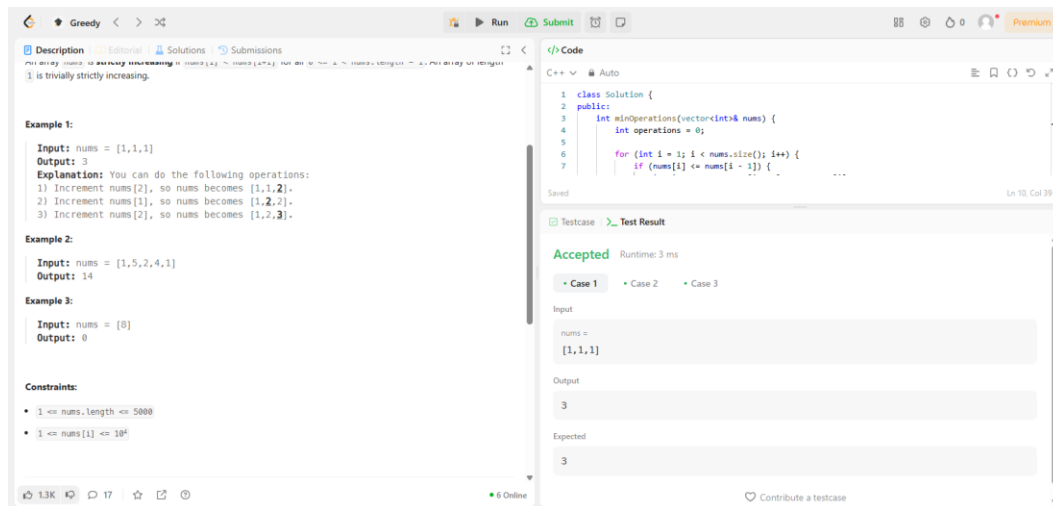
Output:

Figure2

Learning Outcomes:
- Understanding how to use a **greedy approach** by making optimal decisions step-by-step to solve problems effectively.
- Enhancing skills in **array traversal** and applying conditional logic to modify elements based on specific

constraints.

- Learning to **calculate minimum operations** required to meet a condition by analyzing and adjusting elements iteratively.
- Developing efficient algorithms with **linear time complexity** $O(n)O(n)$ and minimal space usage, ensuring scalability and performance.

<u>**Problem–3**</u>

Aim: Min Operations to Make a Subsequence

Objective: The objective of this code is to determine the minimum number of operations required to make the given array target a subsequence of another array arr. By utilizing the concept of the Longest Increasing Subsequence (LIS) on the indices of target elements in arr, the code efficiently calculates the elements that need to be inserted to achieve the desired subsequence.

Implementation/Code:

```cpp
class Solution {
public:
    int minOperations(vector<int>& target, vector<int>& arr) {
        unordered_map<int, int> targetIndex;
        for (int i = 0; i < target.size(); i++) {
            targetIndex[target[i]] = i;
        }

        vector<int> mappedIndices;
        for (int num : arr) {
            if (targetIndex.find(num) != targetIndex.end()) {
                mappedIndices.push_back(targetIndex[num]);
            }
        }
        vector<int> lis;
        for (int index : mappedIndices) {
            auto it = lower_bound(lis.begin(), lis.end(), index);
            if (it == lis.end()) {
                lis.push_back(index);
            } else {
                *it = index;
            }
        }
        return target.size() - lis.size();
    }
};
```
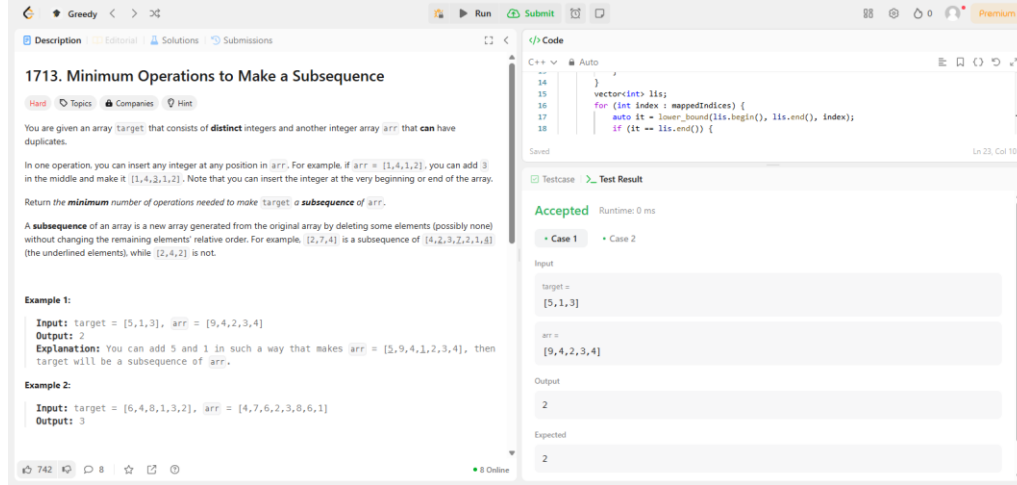
Output:

Figure3

Learning Outcomes:

- Developing the ability to use the **Longest Increasing Subsequence (LIS)** approach to optimize problems involving subsequences efficiently.

- Learning how to leverage **hash maps** for mapping elements to indices, enabling quick lookups and processing large datasets.

- Understanding how to filter and transform arrays based on constraints while maintaining their **relative order**. Enhancing algorithmic skills by creating solutions that achieve $O(n\log n)$ O(n \log n) time complexity and are scalable for large inputs, aligning with real-world scenarios.