Experiment 8

Student Name: Shivansh Singh **UID: 22BET10105**
Branch: IT **Section/Group: 22BET_IOT-701/A**
Semester: 6th **Date of Performance:28.03.25**
Subject Name: AP Lab - 2 **Subject Code: 22ITP-351**

1. Aim: To Optimize operations to maximize efficiency or minimize cost in different problem scenarios.

i.) Max Units on a Truck
ii.) Min Operations to make array increasing
iii.) Remove stones to Maximize total
iv.) Max Score from removing substrings
v.) Min operations to make a subsequence
vi.) Max number of tasks you can assign

2. Objective:

- Apply Dynamic Programming (DP) to solve optimization and sequence-based problems efficiently.
- Optimize the selection and allocation of resources (e.g., boxes, workers, or operations) to maximize output or minimize cost.
- Reduce the number of modifications required to achieve a desired structure, such as making an array strictly increasing or forming a subsequence.
- Apply operations intelligently to minimize remaining values or maximize scores, such as removing stones or substrings for optimal results.
- Assign tasks, load items, or utilize workers in a way that maximizes efficiency while considering given constraints.
- Make decisions that balance available resources, limitations (e.g., truck size, worker strength), and potential rewards to achieve the best possible outcome.

3. Code:

<u>Problem 1: Max Units on a Truck</u>

```
class Solution {
public:
    int maximumUnits(vector<vector<int>>& boxTypes, int truckSize) {
        std::sort(boxTypes.begin(), boxTypes.end(), [](const std::vector<int>& a, const std::vector<int>& b)
        {
            return a[1] > b[1]; // Sort by units per box in decreasing order
        });
        int maxUnits = 0;
```

```
        for (const auto& box : boxTypes) {
            int boxCount = box[0];  // Number of boxes
            int unitsPerBox = box[1]; // Units per box
            // Take as many boxes as possible without exceeding truckSize
            int boxesToTake = std::min(truckSize, boxCount);
            maxUnits += boxesToTake * unitsPerBox;
            truckSize -= boxesToTake;
            if (truckSize == 0) break; // Stop when truck is full
        }
        return maxUnits;
    }
};
```

Problem 2: Min Operations to make array incresing

```
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] - nums[i] + 1;
                nums[i] += increment;  // Make nums[i] strictly greater than nums[i-1]
                operations += increment;
            }
        }
        return operations;
    }
};
```

Problem 3: Remove stones to Maximize total

```
class Solution {
public:
    int minStoneSum(vector<int>& piles, int k) {
        std::priority_queue<int> maxHeap(piles.begin(), piles.end());
        while (k-- > 0) {
            int largestPile = maxHeap.top();
            maxHeap.pop();
            largestPile -= largestPile / 2;
            maxHeap.push(largestPile);
        }
        int totalStones = 0;
        while (!maxHeap.empty()) {
            totalStones += maxHeap.top();
            maxHeap.pop();
        }
        return totalStones;
    }
};
```

Problem 4: Max Score from removing substrings

```
class Solution {
public:
    int maximumGain(string s, int x, int y) {
        if (y > x) {
            std::swap(x, y);
            for (char &ch : s) {
                if (ch == 'a') ch = 'b';
                else if (ch == 'b') ch = 'a';
            }
        }
        int maxPoints = 0;
        std::stack<char> st;
        std::string remaining;
        for (char c : s) {
            if (!st.empty() && st.top() == 'a' && c == 'b') {
                st.pop();
                maxPoints += x;
            } else {
                st.push(c);
            }
        }
        while (!st.empty()) {
            remaining += st.top();
            st.pop();
        }
        std::reverse(remaining.begin(), remaining.end());
        for (char c : remaining) {
            if (!st.empty() && st.top() == 'b' && c == 'a') {
                st.pop(); // Remove 'b'
                maxPoints += y; // Earn y points
            } else {
                st.push(c);
            }
        }
        return maxPoints;
    }
};
```

Problem 5: Min operations to make a subsequence

```
class Solution {
public:
    int minOperations(vector<int>& target, vector<int>& arr) {
        std::unordered_map<int, int> indexMap;
        for (int i = 0; i < target.size(); i++) {
            indexMap[target[i]] = i;
        }
        std::vector<int> sequence;
        for (int num : arr) {
            if (indexMap.find(num) != indexMap.end()) {
```

```
            sequence.push_back(indexMap[num]);
        }
    }
    std::vector<int> lis;
    for (int idx : sequence) {
        auto it = std::lower_bound(lis.begin(), lis.end(), idx);
        if (it == lis.end()) {
            lis.push_back(idx);
        } else {
            *it = idx;
        }
    }
    return target.size() - lis.size();
  }
};
```

6: Max number of tasks you can assign

```
class Solution {
public:
  bool canComplete(int taskCount, std::vector<int>& tasks, std::vector<int>& workers, int pills, int
     strength) {
    std::multiset<int> availableWorkers;
    int m = workers.size();
    for (int i = m - taskCount; i < m; i++) {
        availableWorkers.insert(workers[i]);
    }
    int pillsUsed = 0;
    for (int i = taskCount - 1; i >= 0; i--) {
        int task = tasks[i];
        auto it = availableWorkers.lower_bound(task);
        if (it != availableWorkers.end()) {
            availableWorkers.erase(it);
        } else {
            if (pillsUsed < pills) {
                it = availableWorkers.lower_bound(task - strength);
                if (it != availableWorkers.end()) {
                    availableWorkers.erase(it);
                    pillsUsed++;
                } else {
                    return false;
                }
            } else {
                return false;
            }
        }
    }
    return true;
  }
  int maxTaskAssign(std::vector<int>& tasks, std::vector<int>& workers, int pills, int strength) {
    std::sort(tasks.begin(), tasks.end());
    std::sort(workers.begin(), workers.end());
```

```
        int left = 0, right = std::min(tasks.size(), workers.size()), ans = 0;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (canComplete(mid, tasks, workers, pills, strength)) {
                ans = mid;
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return ans;
    }
};
```

4. Output:

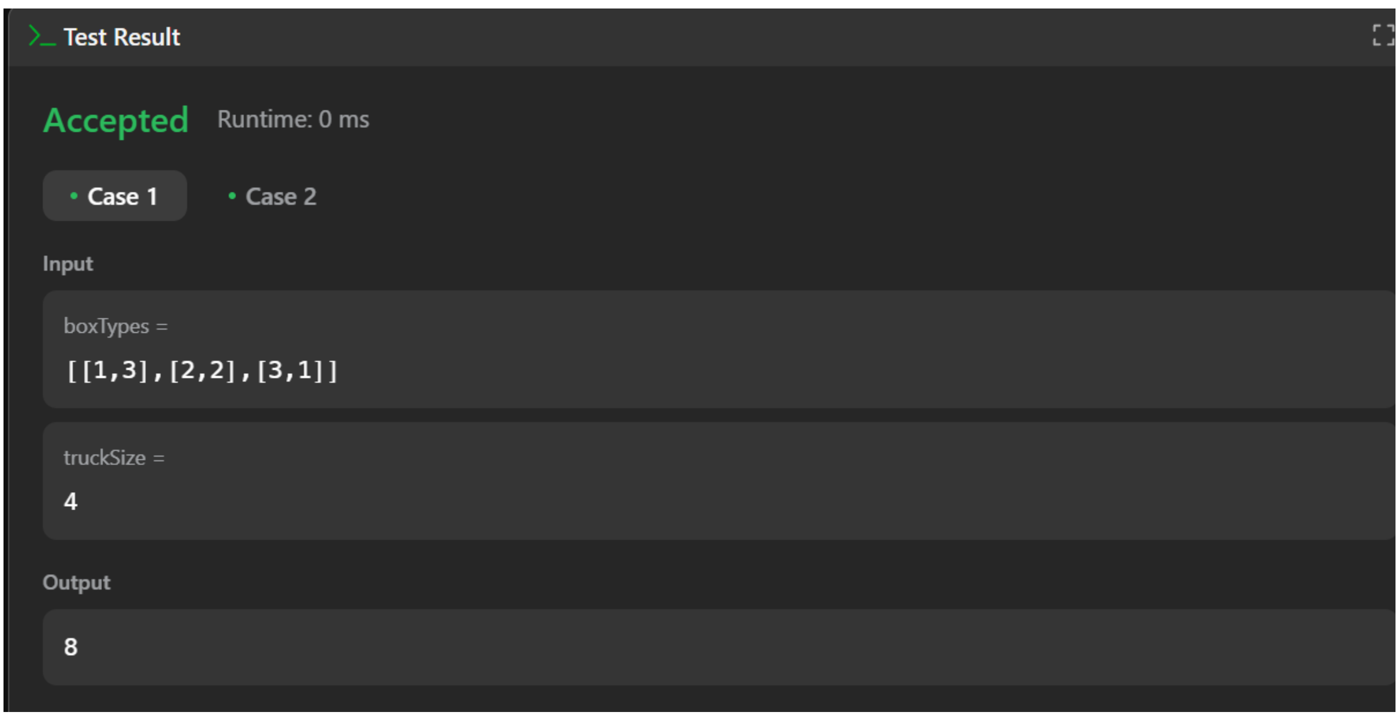

Fig 1. Max Units on a Truck

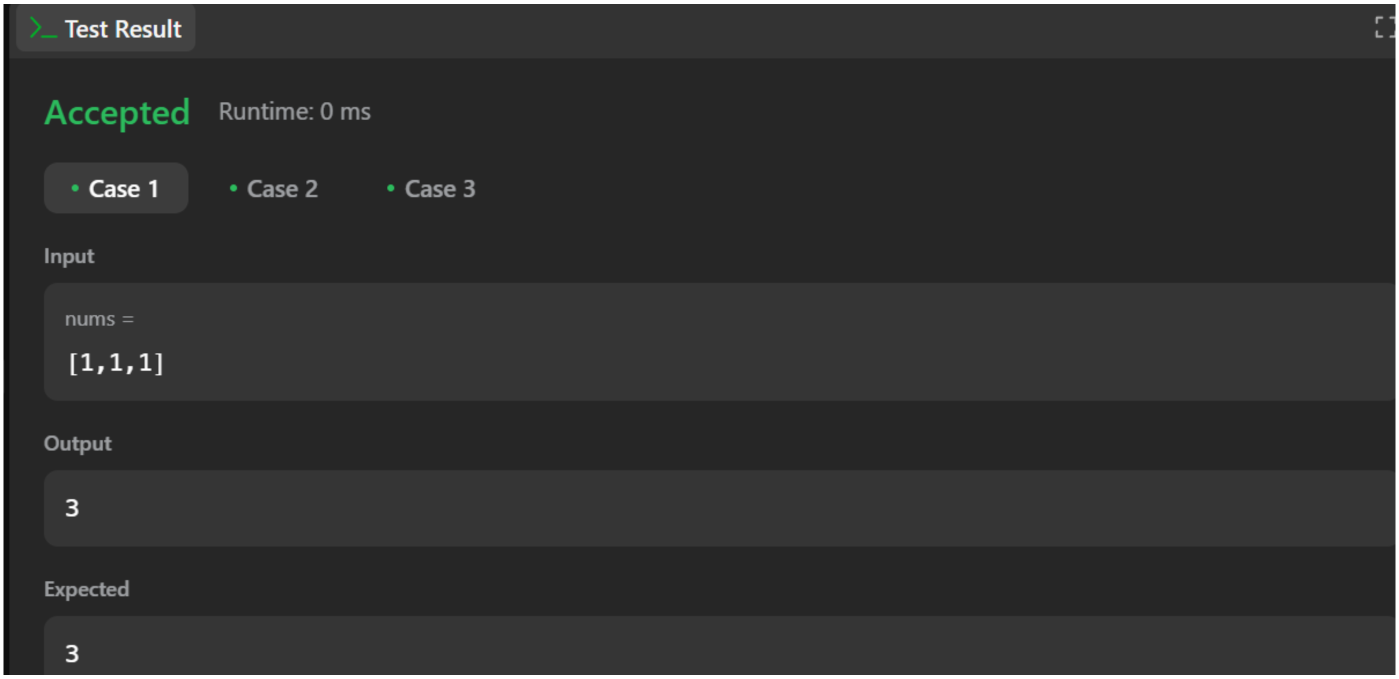

Fig 2. Min Operations to make array increasing

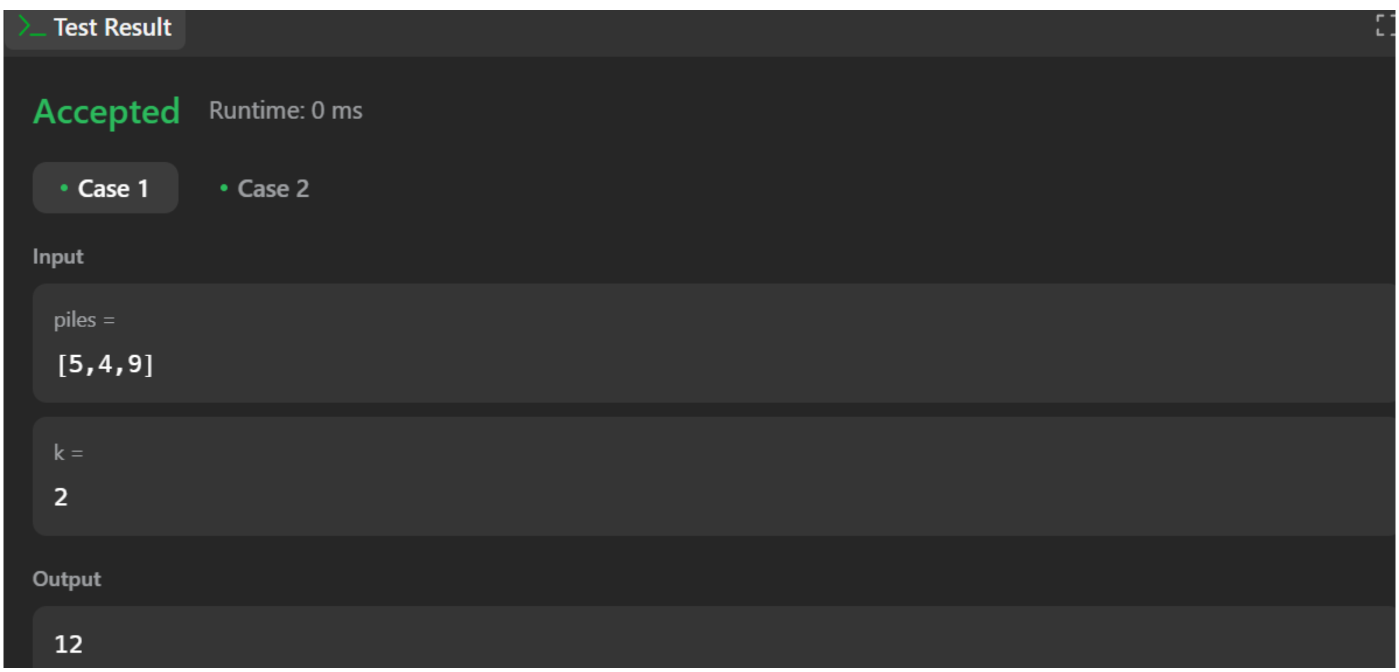

Fig 3. Remove stones to Maximize total

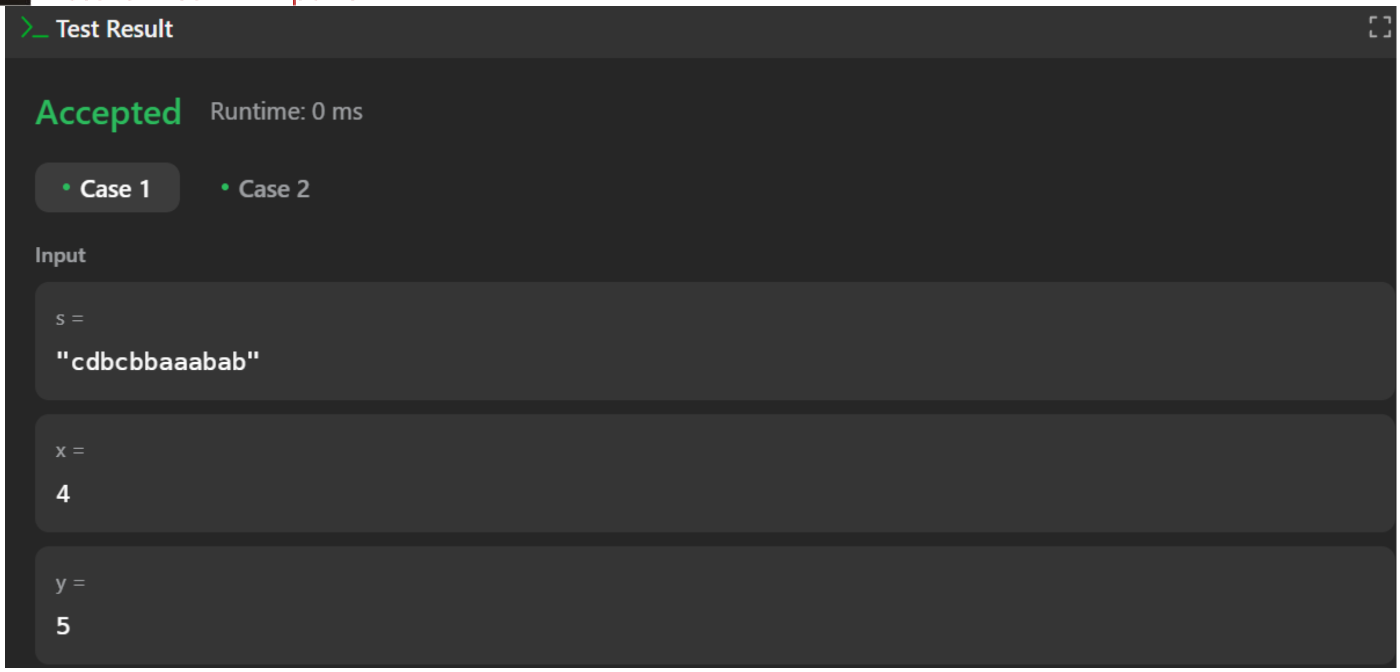

Fig 4. Max Score from removing substrings

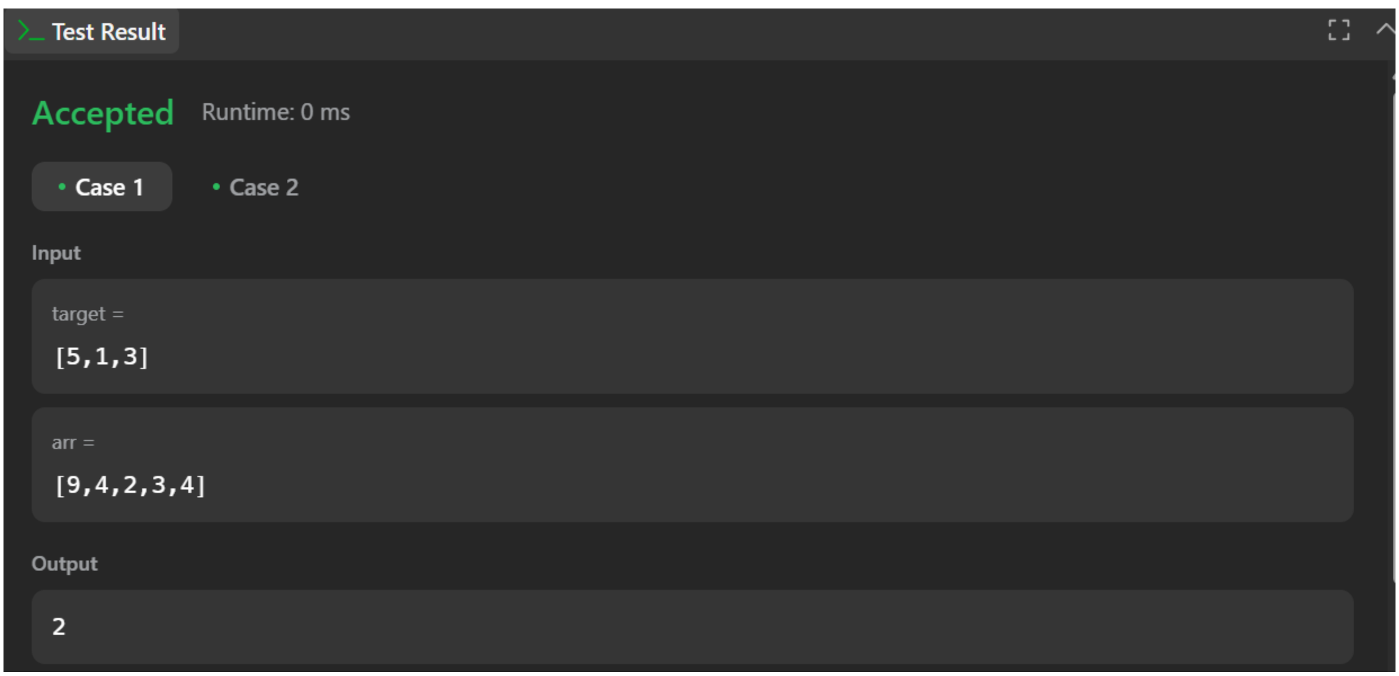

Fig 5. Min operations to make a subsequence

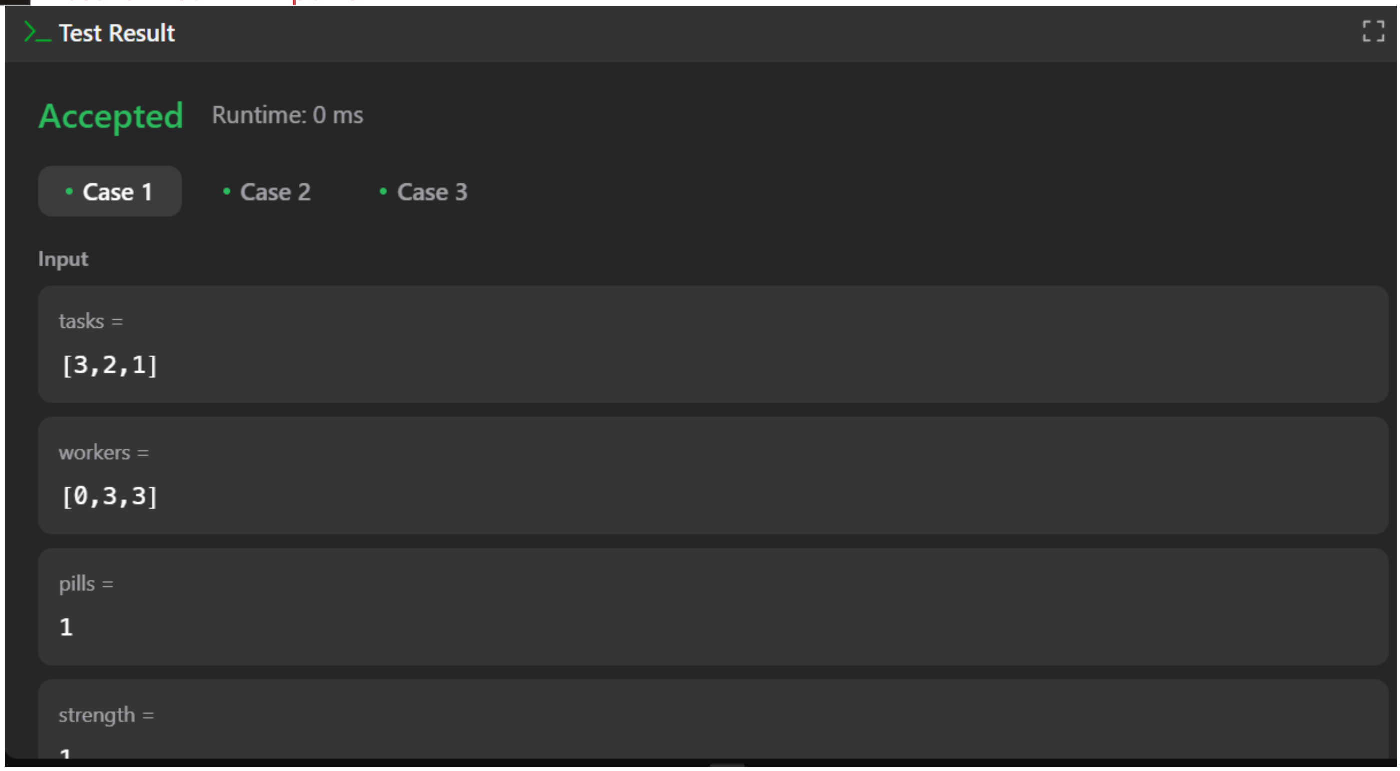

Fig 6. Max number of tasks you can assign

5. Learning Outcomes:

- Understanding Greedy and Optimized Approaches – Learn how to apply greedy algorithms and optimization techniques to maximize efficiency or minimize costs in various problem scenarios.
- Enhancing Problem-Solving Skills – Develop the ability to analyze constraints and devise optimal strategies for resource allocation, transformation, and removal tasks.
- Implementing Efficient Data Structures – Gain hands-on experience with sorting, heaps, binary search, and other data structures to solve problems efficiently.
- Balancing Trade-offs in Algorithmic Decisions – Learn how to make decisions that balance constraints (e.g., limited capacity, operations, or resources) with the goal of achieving the best outcome.
- Applying Algorithmic Thinking in Real-world Scenarios – Understand how these optimization strategies can be applied to real-world problems like logistics, scheduling, and cost minimization.