Experiment 8

Student Name: Nisha Kumari **UID: 22BET10118**

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 develop a deep understanding of algorithmic problem-solving by implementing efficient strategies such as greedy algorithms, dynamic programming, and optimization techniques.

i.) Max Units on a Truck
ii.) Min Operations to make array incresing
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:

- Max Units on a Truck:
  - Implement a greedy approach by sorting box types based on units per box in descending order and loading the most valuable boxes first to maximize the total units on the truck.
- Min Operations to Make Array Increasing:
  - Ensure a strictly increasing sequence by iterating through the array and increasing elements where necessary, while minimizing the number of operations required.
- Remove Stones to Maximize Total:
  - Use a greedy strategy to remove stones from the highest piles first, ensuring that the total number of stones removed is maximized within the given constraints.
- Max Score from Removing Substrings:
  - Implement string manipulation techniques to remove high-scoring substrings in an optimal order, ensuring the maximum possible score is obtained efficiently.
- Min Operations to Make a Subsequence:
  - Utilize binary search or dynamic programming to determine the minimum number of operations required to transform an input sequence into a given target subsequence.
- Max Number of Tasks You Can Assign:
  - Optimize task allocation using sorting, binary search, or greedy methods to maximize the number of tasks assigned while maintaining balance and efficiency.

3. Code:

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

```
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 maxUnits = 0;

        for (auto& box : boxTypes) {
            int boxCount = min(truckSize, box[0]);  // Take as many boxes as possible
            maxUnits += boxCount * box[1];  // Add units to total
            truckSize -= boxCount;  // Reduce available truck space

            if (truckSize == 0) break;  // Stop when truck is full
        }

        return maxUnits;
    }
};
```

Problem 2: Min Operations to make the Array Increasing

```
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 diff = (nums[i - 1] - nums[i]) + 1;
                nums[i] += diff;  // Increase current element
                operations += diff;  // Count operations
            }
        }

        return operations;
    }
};
```

Problem 3: Remove stone to minimize the total

```
class Solution {
public:
    int minStoneSum(vector<int>& piles, int k) {
        priority_queue<int> maxHeap(piles.begin(), piles.end()); // Max heap
        while (k-- > 0) {
            int top = maxHeap.top(); // Get largest pile
            maxHeap.pop();
            top -= top / 2; // Remove floor(top/2) stones
            maxHeap.push(top); // Push back updated pile
        }

        int totalStones = 0;
        while (!maxHeap.empty()) {
            totalStones += maxHeap.top();
            maxHeap.pop();
```

```
        }

        return totalStones;
    }
};
```

Problem 4: Maximum Score from Removing Substrings

```
class Solution {
public:
    int maximumGain(string s, int x, int y) {
         char firstChar = 'a', secondChar = 'b';
        if (x < y) {
            swap(x, y);
            swap(firstChar, secondChar);  // Process "ba" before "ab" if y is greater
        }
        int score = 0;
        stack<char> st1;
        // First pass: Remove the higher-scoring substring
        string remaining = "";
        for (char c : s) {
            if (!st1.empty() && st1.top() == firstChar && c == secondChar) {
                st1.pop();
                score += x;
            } else {
                st1.push(c);
            }
        }
        // Collect remaining characters after first pass
        while (!st1.empty()) {
            remaining += st1.top();
            st1.pop();
        }
        reverse(remaining.begin(), remaining.end());  // Since we used a stack
        // Second pass: Remove the lower-scoring substring
        for (char c : remaining) {
            if (!st1.empty() && st1.top() == secondChar && c == firstChar) {
                st1.pop();
                score += y;
            } else {
                st1.push(c);
            }
        }
        return score;
    }};
```

Problem 5: Minimum Operations to make a Subsequence

```
class Solution {
public:
    int minOperations(vector<int>& target, vector<int>& arr) {
        unordered_map<int, int> indexMap;
        for (int i = 0; i < target.size(); i++) {
            indexMap[target[i]] = i;  // Store index of each element in target
        }

        vector<int> sequence;
        for (int num : arr) {
            if (indexMap.count(num)) {
                sequence.push_back(indexMap[num]);  // Convert arr elements to indices
            }
        }

        return target.size() - lengthOfLIS(sequence);
    }
private:
    int lengthOfLIS(vector<int>& nums) {
        vector<int> lis;
        for (int num : nums) {
            auto it = lower_bound(lis.begin(), lis.end(), num);
            if (it == lis.end()) {
                lis.push_back(num);
            } else {
                *it = num;
            }
        }
        return lis.size();
    }
};
```

Problem 6: Maximum number of tasks you can assign

```
class Solution {
public:
    bool canAssign(int mid, vector<int>& tasks, vector<int>& workers, int pills, int strength) {
        multiset<int> availableWorkers(workers.end() - mid, workers.end());
        int usedPills = 0;

        for (int i = mid - 1; i >= 0; --i) {
            auto it = availableWorkers.lower_bound(tasks[i]);
            if (it != availableWorkers.end()) {
                availableWorkers.erase(it);  // Assign task without pill
            } else {
                if (usedPills >= pills) return false;  // No more pills available
                it = availableWorkers.lower_bound(tasks[i] - strength);
                if (it == availableWorkers.end()) return false;  // No valid worker even with pill
                availableWorkers.erase(it);
                usedPills++;
            }
        }
        return true;
    }
```

```
    int maxTaskAssign(vector<int>& tasks, vector<int>& workers, int pills, int strength) {
        sort(tasks.begin(), tasks.end());
        sort(workers.begin(), workers.end());

        int left = 0, right = min(tasks.size(), workers.size()), result = 0;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (canAssign(mid, tasks, workers, pills, strength)) {
                result = mid;  // Try to assign more tasks
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return result;
    }
};
```

4. Output:

Fig 1. Max Units on a Truck

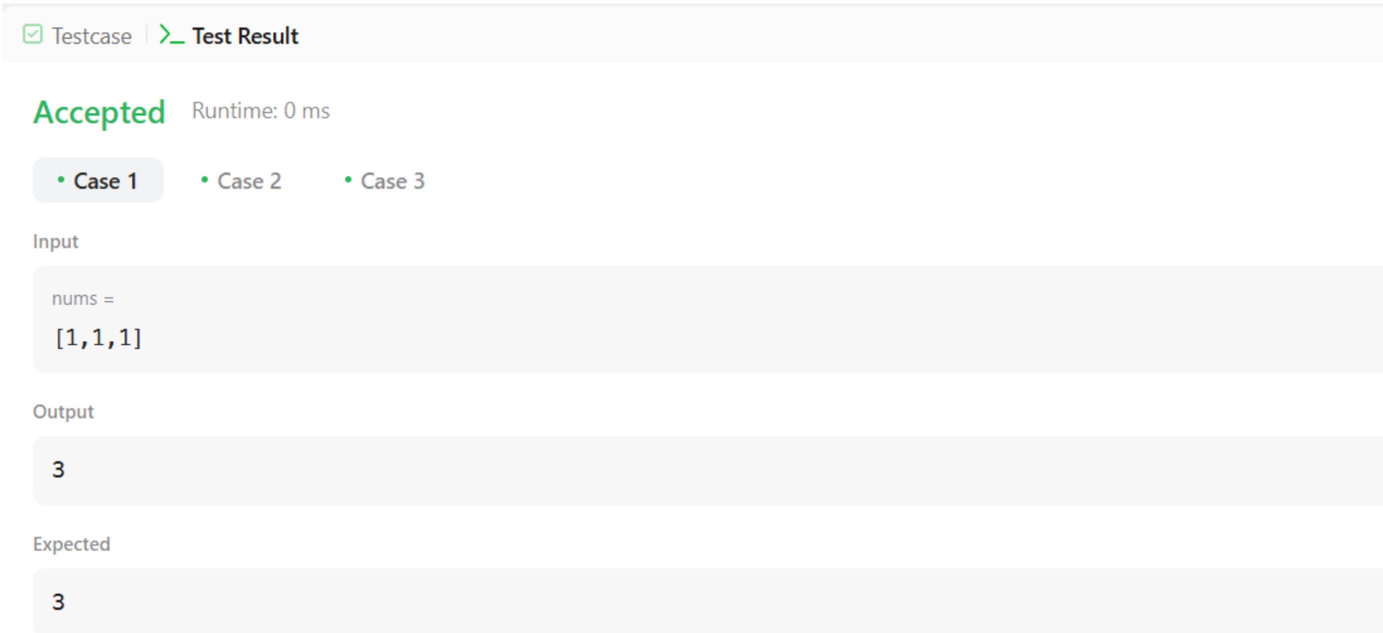

Fig 2. Min Operations to make array incresing

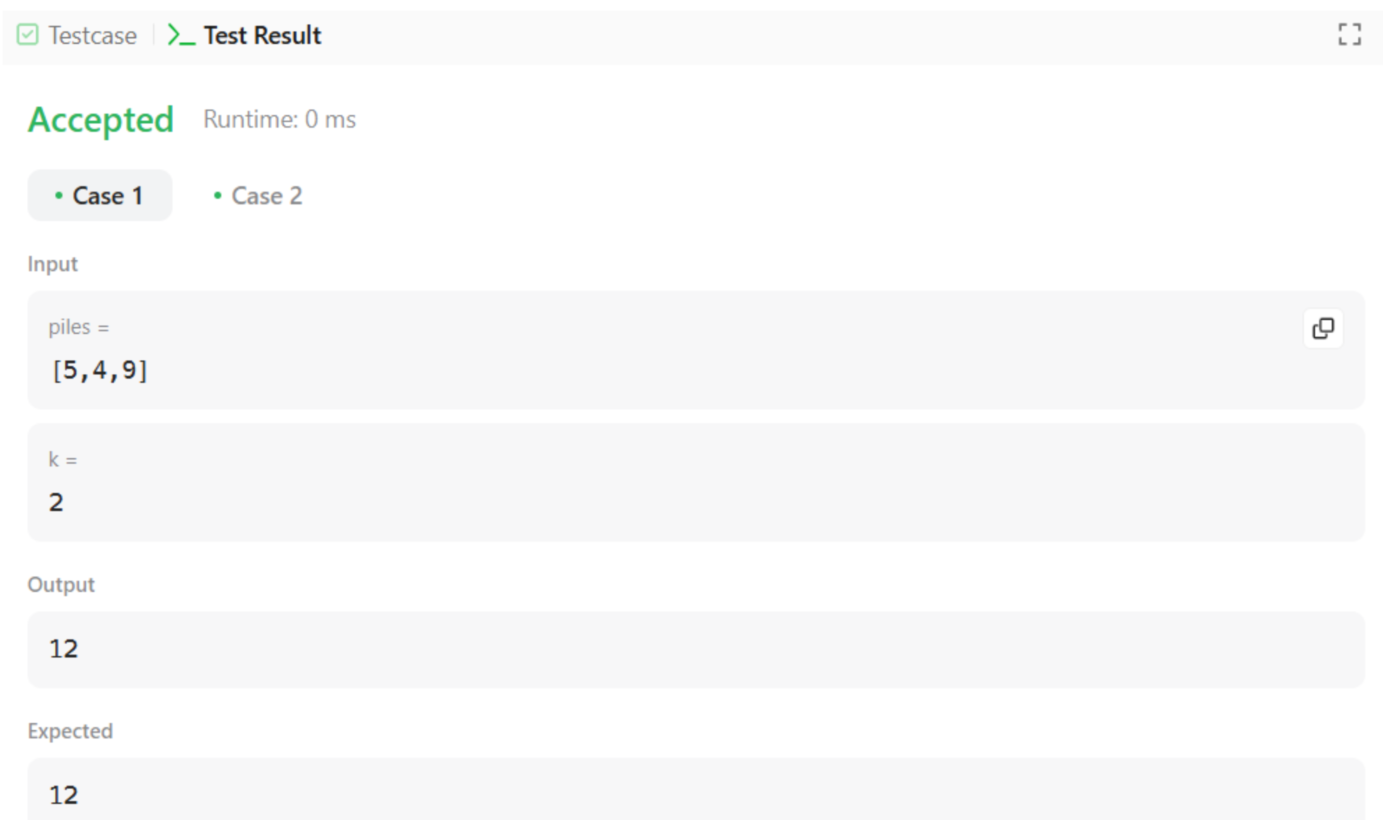

Fig 3. Remove stones to Maximize total

Fig 4. Maximum Score from Removing Substrings

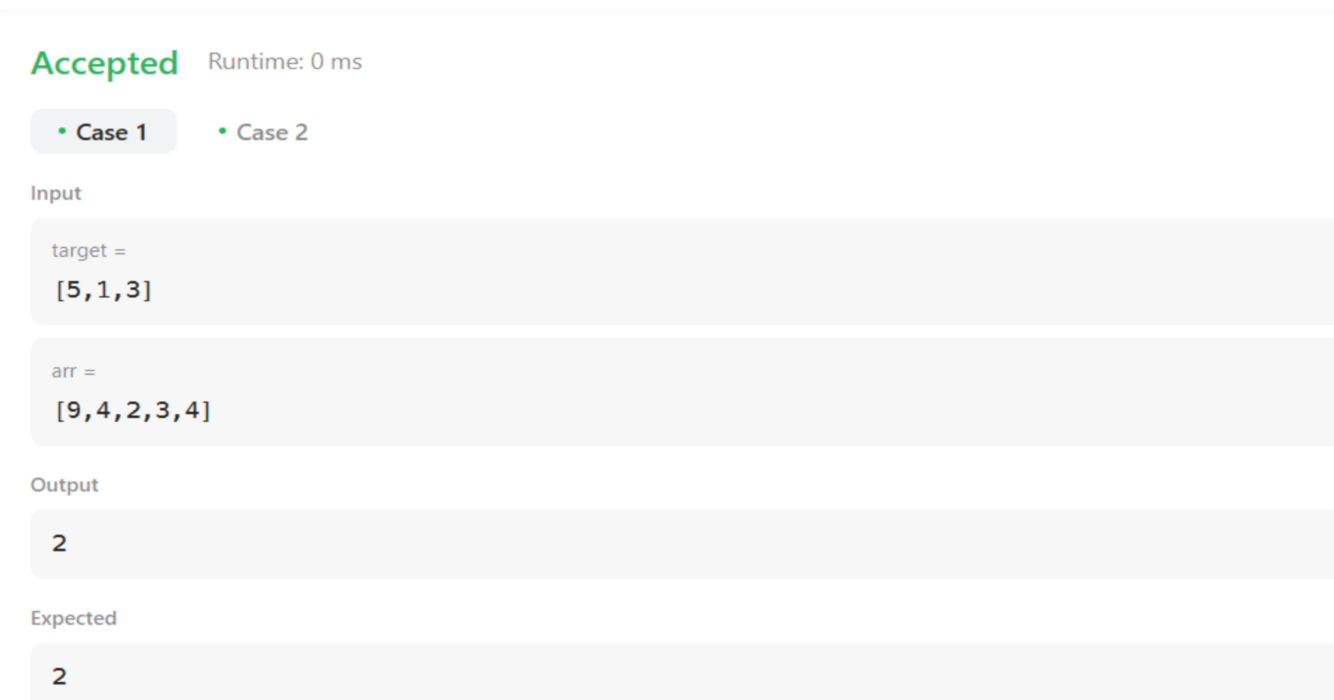

Fig. Minimum Operations to make a Subsequence

Fig 6. Maximum number of tasks you can assign

5. Learning Outcomes:

- Optimize Problem-Solving with Greedy and Binary Search – Learn how to make optimal choices step-by-step and apply binary search to efficiently find the best solution.
- Gain hands-on experience with heaps, hash maps, sets, and multisets for efficient data management and retrieval.
- Develop skills in modifying arrays in-place, handling substring removals, and working with sorted data to minimize operations.
- Understand how to distribute limited resources like truck space, pills, and workers effectively using priority-based selection strategies.
- Learn to optimize solutions by choosing the right approach (sorting, heaps, two-pointers, dynamic programming) while ensuring optimal performance.