Assignment 8 (Advance Programming)

Submitted By: Ankit Kharb **Class:** IOT_614(B)

UID: 22BCS16964

Q- Max Units on a Truck

You are assigned to put some amount of boxes onto **one truck**. You are given a 2D array boxTypes, where boxTypes[i] = [numberOfBoxes_i, $\text{numberOfUnitsPerBox}_i$]:

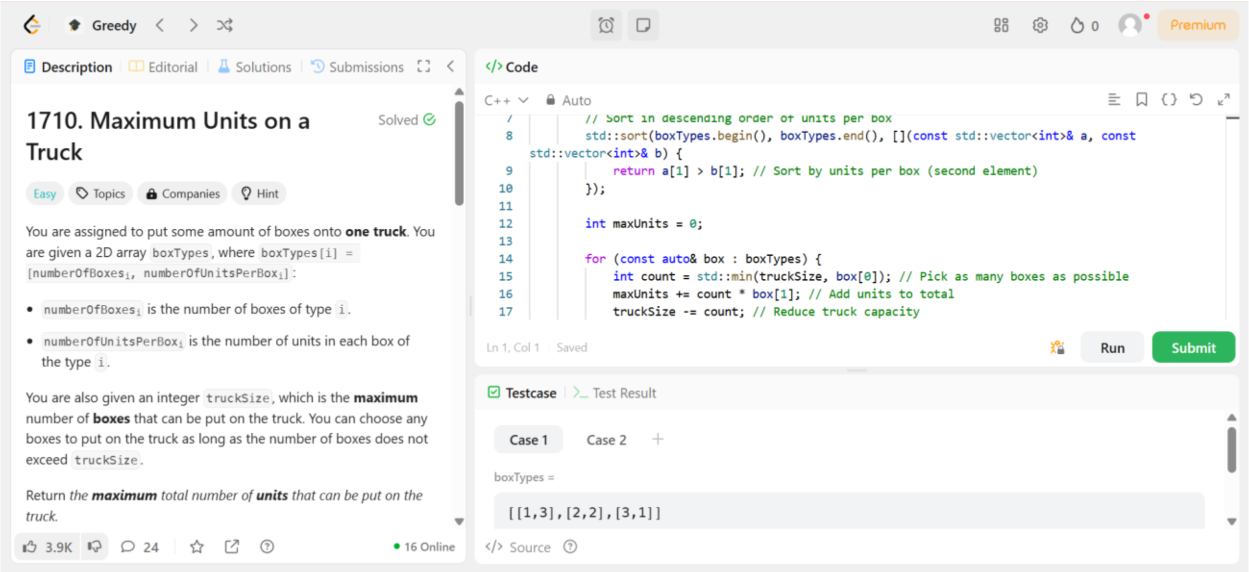
- numberOfBoxes_i is the number of boxes of type i.
- $\text{numberOfUnitsPerBox}_i$ is the number of units in each box of the type i.

You are also given an integer truckSize, which is the **maximum** number of **boxes** that can be put on the truck. You can choose any boxes to put on the truck as long as the number of boxes does not exceed truckSize.

Return *the **maximum** total number of **units** that can be put on the truck.*

Solution:

```
#include <vector>
#include <algorithm>

class Solution {
public:
    int maximumUnits(std::vector<std::vector<int>>& boxTypes, int truckSize) {
        // Sort in descending order of units per box
        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 (second element)
        });

        int maxUnits = 0;
    for (const auto& box : boxTypes) {
      int count = std::min(truckSize, box[0]); // Pick as many boxes as possible
      maxUnits += count * box[1]; // Add units to total
      truckSize -= count; // Reduce truck capacity

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

    return maxUnits;
  }
};
```



1710. Maximum Units on a Truck

You are assigned to put some amount of boxes onto **one truck**. You are given a 2D array `boxTypes`, where `boxTypes[i] = [numberOfBoxes_i, numberOfUnitsPerBox_i]`:

- `numberOfBoxes_i` is the number of boxes of type `i`.
- `numberOfUnitsPerBox_i` is the number of units in each box of the type `i`.

You are also given an integer `truckSize`, which is the **maximum** number of **boxes** that can be put on the truck. You can choose any boxes to put on the truck as long as the number of boxes does not exceed `truckSize`.

Return *the* ***maximum*** *total number of* ***units*** *that can be put on the truck.*

```
        // Sort in descending order of units per box
        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 (second element)
        });

        int maxUnits = 0;

        for (const auto& box : boxTypes) {
            int count = std::min(truckSize, box[0]); // Pick as many boxes as possible
            maxUnits += count * box[1]; // Add units to total
            truckSize -= count; // Reduce truck capacity
```

Ln 1, Col 1 | Saved

boxTypes =

```
[[1,3],[2,2],[3,1]]
```

Q- Min Operations to Make Array Increasing

You are given an integer array nums (**0-indexed**). In one operation, you can choose an element of the array and increment it by 1.

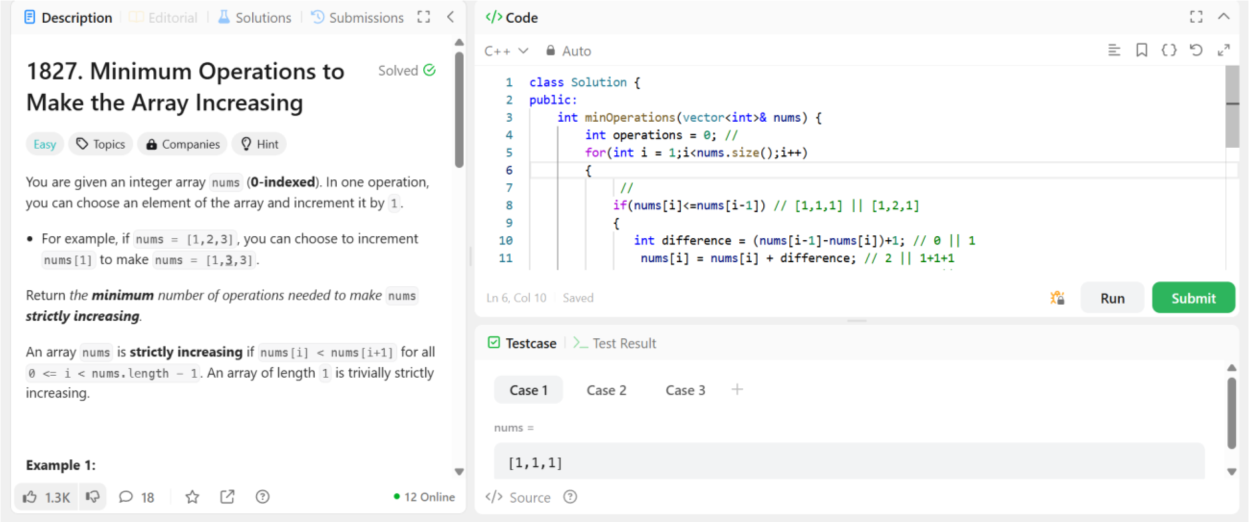
- For example, if nums = [1,2,3], you can choose to increment nums[1] to make nums = [1,**3**,3].

Return *the **minimum** number of operations needed to make* nums ***strictly increasing***.

An array nums is **strictly increasing** if nums[i] < nums[i+1] for all 0 <= i < nums.length - 1. An array of length 1 is trivially strictly increasing.

Solution:

```
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]) // [1,1,1] || [1,2,1]

            {

               int difference = (nums[i-1]-nums[i])+1; // 0 || 1

                nums[i] = nums[i] + difference; // 2 || 1+1+1

                operations = difference + operations; // 1 ||1+1


            }

            else

            {


            }


        }

        return operations;
    }
};
```



1827. Minimum Operations to Make the Array Increasing

You are given an integer array `nums` (**0-indexed**). In one operation, you can choose an element of the array and increment it by `1`.

- For example, if `nums = [1,2,3]`, you can choose to increment `nums[1]` to make `nums = [1,3,3]`.

Return *the **minimum** number of operations needed to make* `nums` ***strictly increasing***.

An array `nums` is **strictly increasing** if `nums[i] < nums[i+1]` for all `0 <= i < nums.length - 1`. An array of length `1` is trivially strictly increasing.

Example 1:

```
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]) // [1,1,1] || [1,2,1]
          {
            int difference = (nums[i-1]-nums[i])+1; // 0 || 1
             nums[i] = nums[i] + difference; // 2 || 1+1+1
```

nums =

```
[1,1,1]
```

Q- Max Score from Removing Substrings

You are given a string s and two integers x and y. You can perform two types of operations any number of times.

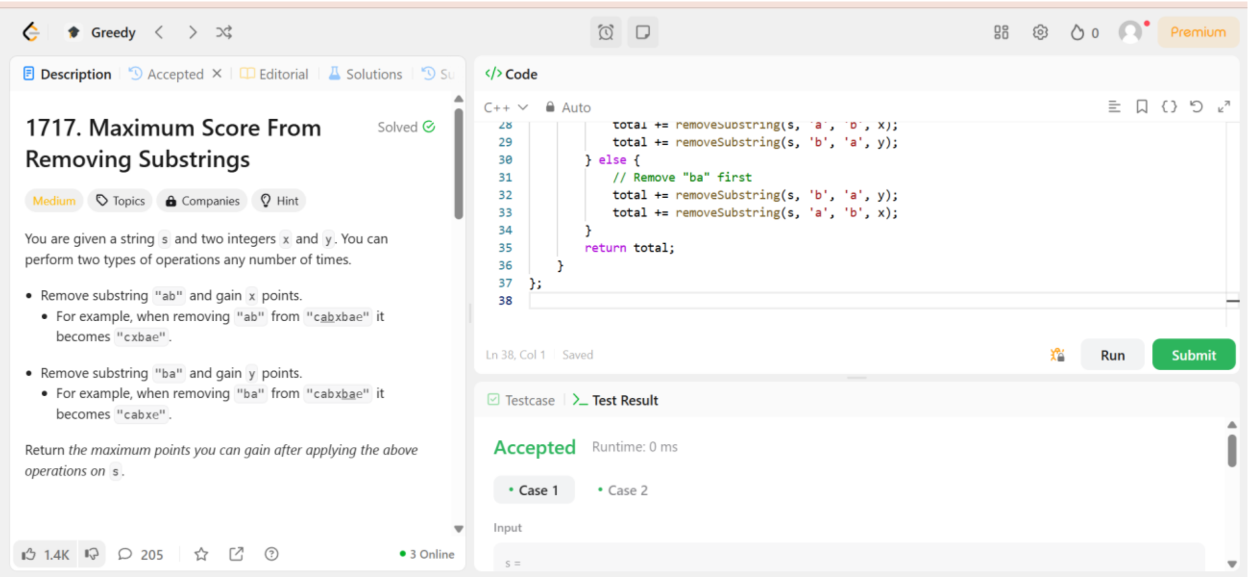
- Remove substring "ab" and gain x points.
    - For example, when removing "ab" from "cabxbae" it becomes "cxbae".
- Remove substring "ba" and gain y points.
    - For example, when removing "ba" from "cabxbae" it becomes "cabxe".

Return *the maximum points you can gain after applying the above operations on* s

Solution:

```
class Solution {
public:
    int removeSubstring(string &s, char a, char b, int points) {
        stack<char> stk;
        int score = 0;
        for (char ch : s) {
            if (!stk.empty() && stk.top() == a && ch == b) {
                stk.pop();
                score += points;
            } else {
                stk.push(ch);
            }
        }
        // Reconstruct the string from the stack
        s.clear();
        while (!stk.empty()) {
            s += stk.top();
            stk.pop();
        }
        reverse(s.begin(), s.end());
        return score;
    }

    int maximumGain(string s, int x, int y) {
        int total = 0;
        if (x > y) {
            // Remove "ab" first
            total += removeSubstring(s, 'a', 'b', x);
            total += removeSubstring(s, 'b', 'a', y);
        } else {
            // Remove "ba" first
```

```
            total += removeSubstring(s, 'b', 'a', y);
            total += removeSubstring(s, 'a', 'b', x);
        }
        return total;
    }
};
```



1717. Maximum Score From Removing Substrings

You are given a string s and two integers x and y. You can perform two types of operations any number of times.

- Remove substring "ab" and gain x points.
  - For example, when removing "ab" from "cabxbae" it becomes "cxbae".
- Remove substring "ba" and gain y points.
  - For example, when removing "ba" from "cabxbae" it becomes "cabxe".

Return *the maximum points you can gain after applying the above operations on* s.

```
            total += removeSubstring(s, 'a', 'b', x);
            total += removeSubstring(s, 'b', 'a', y);
        } else {
            // Remove "ba" first
            total += removeSubstring(s, 'b', 'a', y);
            total += removeSubstring(s, 'a', 'b', x);
        }
        return total;
    }
};
```

Accepted Runtime: 0 ms

Q- Min Operations to Make a Subsequence

You are given an array target that consists of **distinct** integers and another integer array arr that **can** have duplicates.

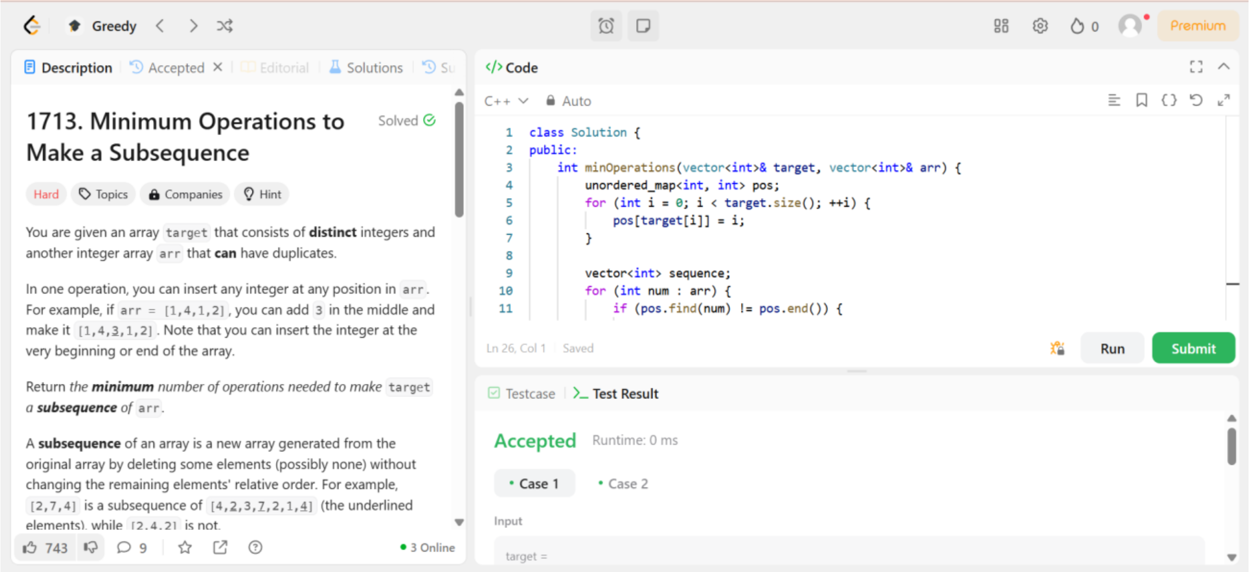
In one operation, you can insert any integer at any position in arr. For example, if arr = [1,4,1,2], you can add 3 in the middle and make it [1,4,3,1,2]. Note that you can insert the integer at the very beginning or end of the array.

Return *the **minimum** number of operations needed to make* target *a **subsequence** of* arr.

A **subsequence** of an array is a new array generated from the original array by deleting some elements (possibly none) without changing the remaining elements' relative order. For example, [2,7,4] is a subsequence of [4,2,3,7,2,1,4] (the underlined elements), while [2,4,2] is not.

Solution:

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

        vector<int> sequence;
        for (int num : arr) {
            if (pos.find(num) != pos.end()) {
                int idx = pos[num];
                // Find insertion position for LIS
                auto it = lower_bound(sequence.begin(), sequence.end(), idx);
                if (it == sequence.end()) {
                    sequence.push_back(idx);
                } else {
                    *it = idx;
                }
            }
        }

        return target.size() - sequence.size();
    }
};
```



1713. Minimum Operations to Make a Subsequence

Solved ✓

Hard · Topics · Companies · Hint

You are given an array `target` that consists of **distinct** integers and another integer array `arr` that **can** have duplicates.

In one operation, you can insert any integer at any position in `arr`. For example, if `arr = [1,4,1,2]`, you can add `3` in the middle and make it `[1,4,3,1,2]`. Note that you can insert the integer at the very beginning or end of the array.

Return *the **minimum** number of operations needed to make* `target` *a **subsequence** of* `arr`.

A **subsequence** of an array is a new array generated from the original array by deleting some elements (possibly none) without changing the remaining elements' relative order. For example, `[2,7,4]` is a subsequence of `[4,2,3,7,2,1,4]` (the underlined elements), while `[2,4,2]` is not.

743 · 9 · ● 3 Online

C++ · Auto

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

        vector<int> sequence;
        for (int num : arr) {
            if (pos.find(num) != pos.end()) {
```

Ln 26, Col 1 | Saved

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

target =