Experiment 1

Student Name: Divyanshu **UID:22BCS14850**
Branch: CSE **Section/Group:614 B**
Semester: 6 **Date :09/04/2025**
Subject Name:AP Lab 2 **Subject Code: 22CSH-351**

1. Aim:Maximum Units on a Truck

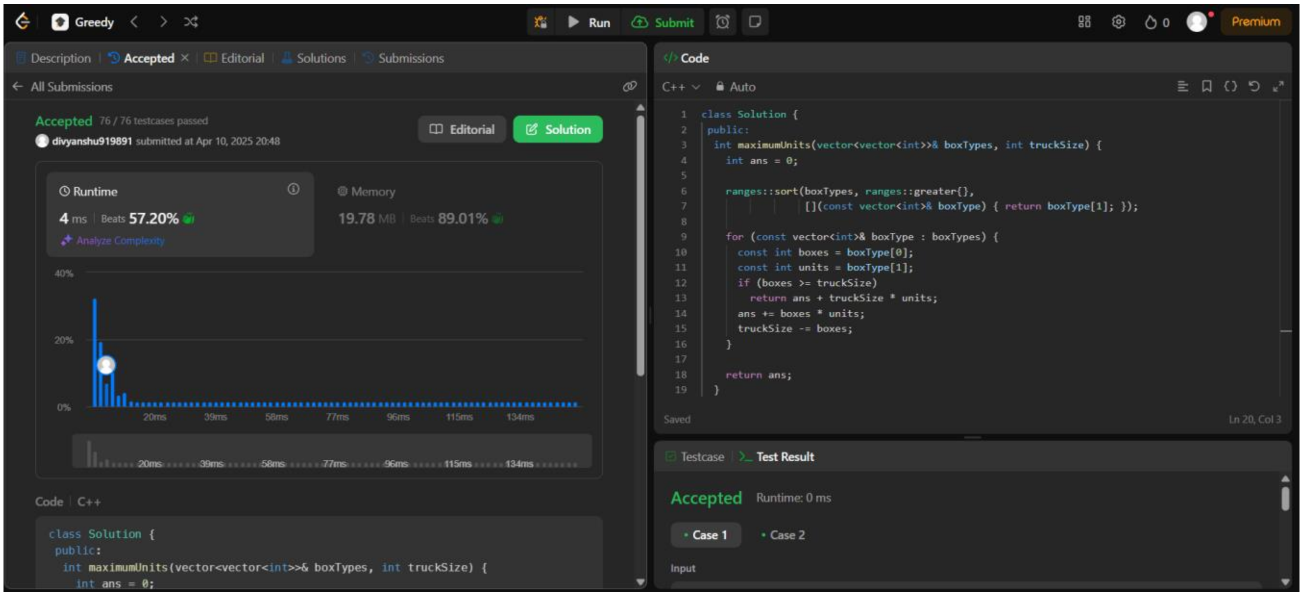
Problem Statement : You are assigned to put some amount of boxes onto **one truck**. You are given a 2D array `boxTypes`, where `boxTypes[i] = [numberOfBoxes`$_i$`, numberOfUnitsPerBox`$_i$`]`:

- `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.*

2. Implementation/Code and Output:



Accepted 76 / 76 testcases passed
divyanshu919891 submitted at Apr 10, 2025 20:48

Runtime
4 ms | Beats 57.20%

Memory
19.78 MB | Beats 89.01%

```cpp
class Solution {
 public:
  int maximumUnits(vector<vector<int>>& boxTypes, int truckSize) {
    int ans = 0;

    ranges::sort(boxTypes, ranges::greater{},
                 [](const vector<int>& boxType) { return boxType[1]; });

    for (const vector<int>& boxType : boxTypes) {
      const int boxes = boxType[0];
      const int units = boxType[1];
      if (boxes >= truckSize)
        return ans + truckSize * units;
      ans += boxes * units;
      truckSize -= boxes;
    }

    return ans;
  }
```

Accepted Runtime: 0 ms

1.Aim: Minimum Operations to Make the Array Increasing

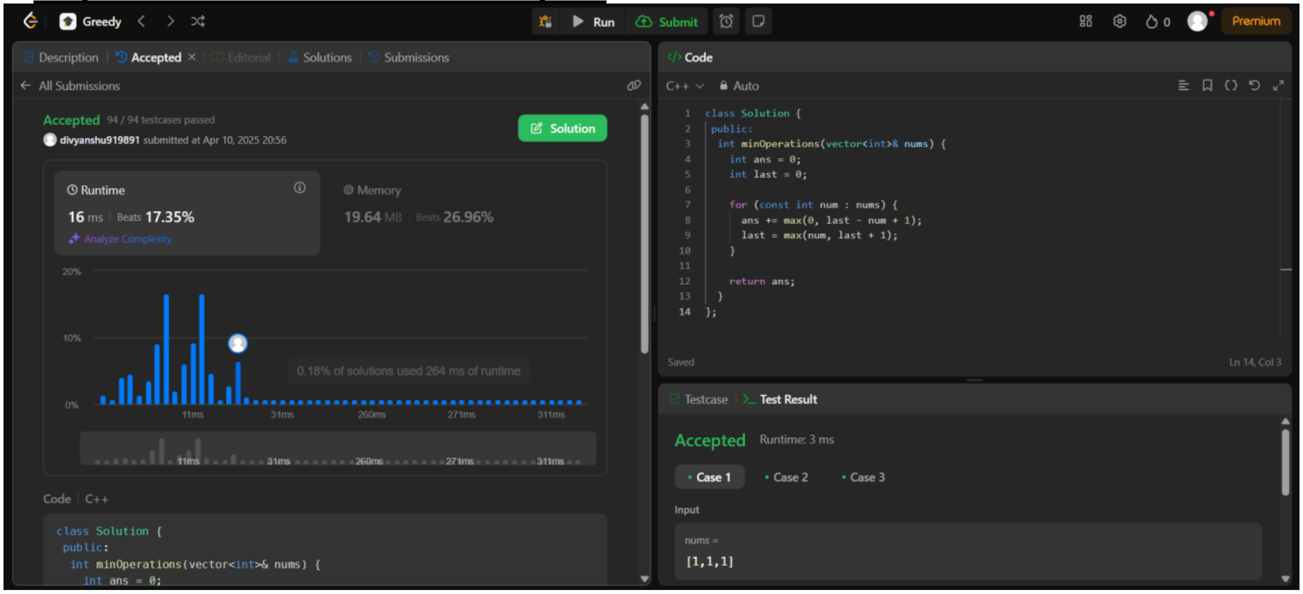
Problem Statement: You are given an integer array `nums` (**0-indexed**). In one operation, you can choose an element of the array and increment it by `1`.

- 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.

2.Implementation/Code and Output:



Accepted 94 / 94 testcases passed

divyanshu919891 submitted at Apr 10, 2025 20:56

Runtime 16 ms | Beats 17.35%

Memory 19.64 MB | Beats 26.96%

0.18% of solutions used 264 ms of runtime

Code | C++

```
class Solution {
public:
  int minOperations(vector<int>& nums) {
    int ans = 0;
```

```
class Solution {
 public:
  int minOperations(vector<int>& nums) {
    int ans = 0;
    int last = 0;

    for (const int num : nums) {
      ans += max(0, last - num + 1);
      last = max(num, last + 1);
    }

    return ans;
  }
};
```

Saved

Ln 14, Col 3

Accepted Runtime: 3 ms

Input

nums =

[1,1,1]

1.Aim: Remove Stones to Minimize the Total

Problem Statement: You are given a **0-indexed** integer array `piles`, where `piles[i]` represents the number of stones in the i^{th} pile, and an integer `k`. You should apply the following operation **exactly** `k` times:

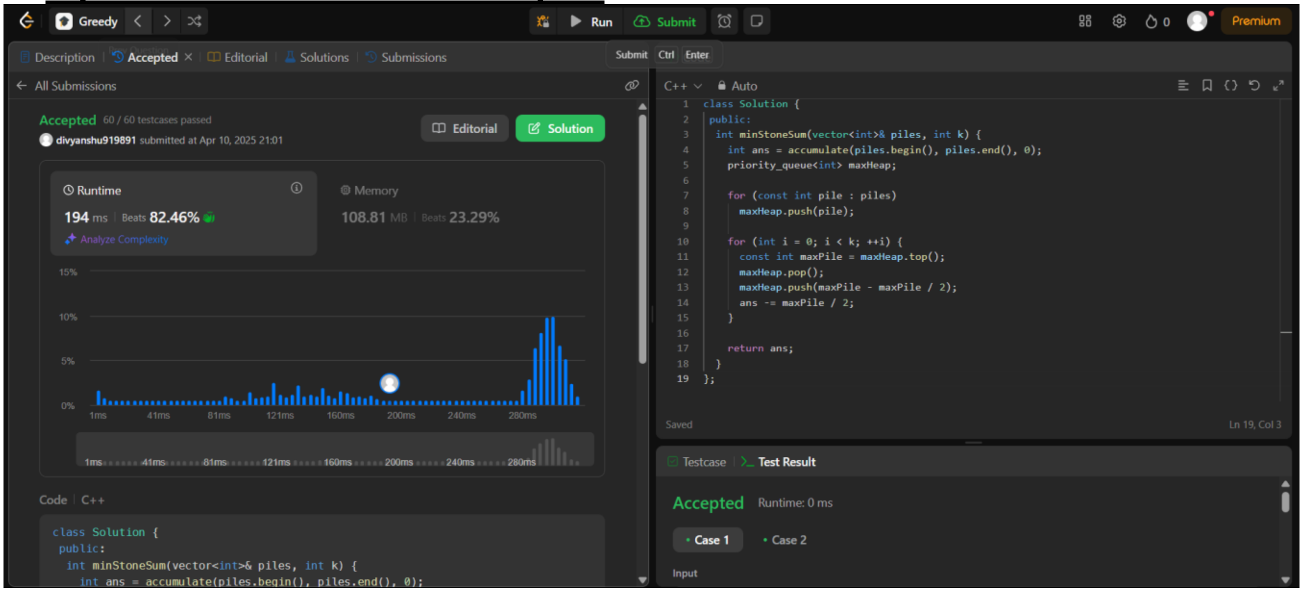
- Choose any `piles[i]` and **remove** `floor(piles[i] / 2)` stones from it.

Notice that you can apply the operation on the **same** pile more than once.

Return *the **minimum** possible total number of stones remaining after applying the* `k` *operations*.

`floor(x)` is the **greatest** integer that is **smaller** than or **equal** to `x` (i.e., rounds `x` down).

2.Implementation/Code and Output:



Accepted 60 / 60 testcases passed
divyanshu919891 submitted at Apr 10, 2025 21:01

Runtime
194 ms | Beats 82.46%

Memory
108.81 MB | Beats 23.29%

Code | C++

```
class Solution {
 public:
  int minStoneSum(vector<int>& piles, int k) {
    int ans = accumulate(piles.begin(), piles.end(), 0);
```

```
class Solution {
 public:
  int minStoneSum(vector<int>& piles, int k) {
    int ans = accumulate(piles.begin(), piles.end(), 0);
    priority_queue<int> maxHeap;

    for (const int pile : piles)
      maxHeap.push(pile);

    for (int i = 0; i < k; ++i) {
      const int maxPile = maxHeap.top();
      maxHeap.pop();
      maxHeap.push(maxPile - maxPile / 2);
      ans -= maxPile / 2;
    }

    return ans;
  }
};
```

Saved
Ln 19, Col 3

Accepted Runtime: 0 ms
Case 1
Case 2
Input

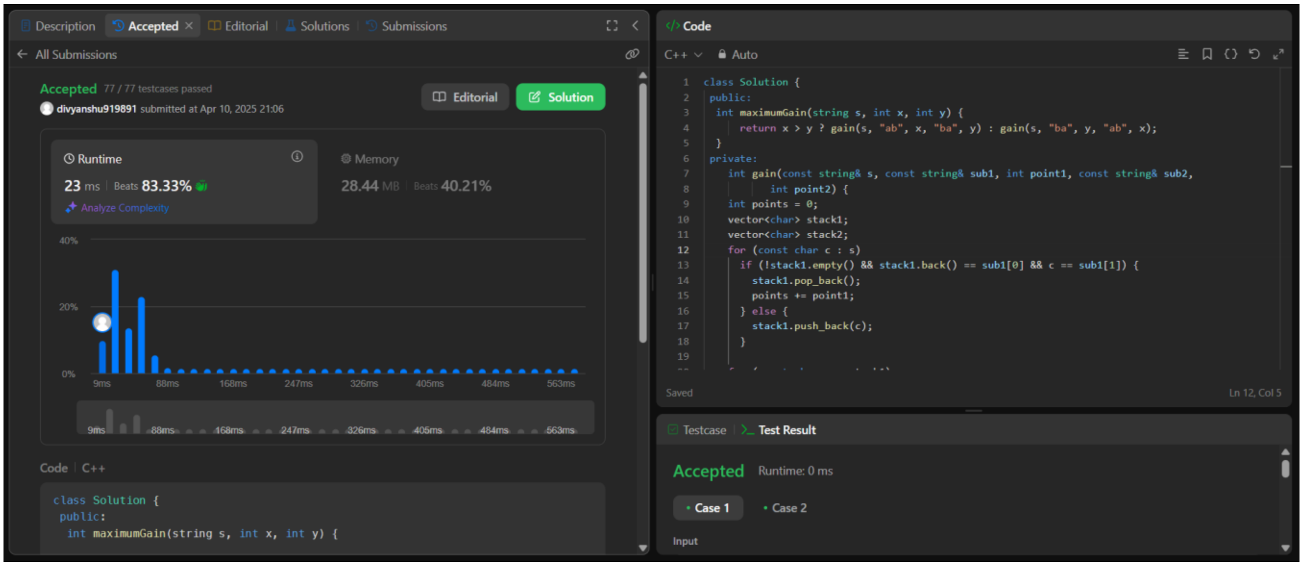
1.Aim: Maximum Score From Removing Substrings

Problem Statements: 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.

2.Implementation and output:



Accepted 77 / 77 testcases passed

divyanshu919891 submitted at Apr 10, 2025 21:06

Runtime 23 ms | Beats 83.33%

Memory 28.44 MB | Beats 40.21%

Code | C++

```
class Solution {
 public:
  int maximumGain(string s, int x, int y) {
```

```
class Solution {
 public:
  int maximumGain(string s, int x, int y) {
    return x > y ? gain(s, "ab", x, "ba", y) : gain(s, "ba", y, "ab", x);
  }
 private:
   int gain(const string& s, const string& sub1, int point1, const string& sub2,
              int point2) {
   int points = 0;
   vector<char> stack1;
   vector<char> stack2;
   for (const char c : s)
     if (!stack1.empty() && stack1.back() == sub1[0] && c == sub1[1]) {
       stack1.pop_back();
       points += point1;
     } else {
       stack1.push_back(c);
     }

```

Accepted Runtime: 0 ms

1.Aim:Minimum Operations to Make a Subsequence

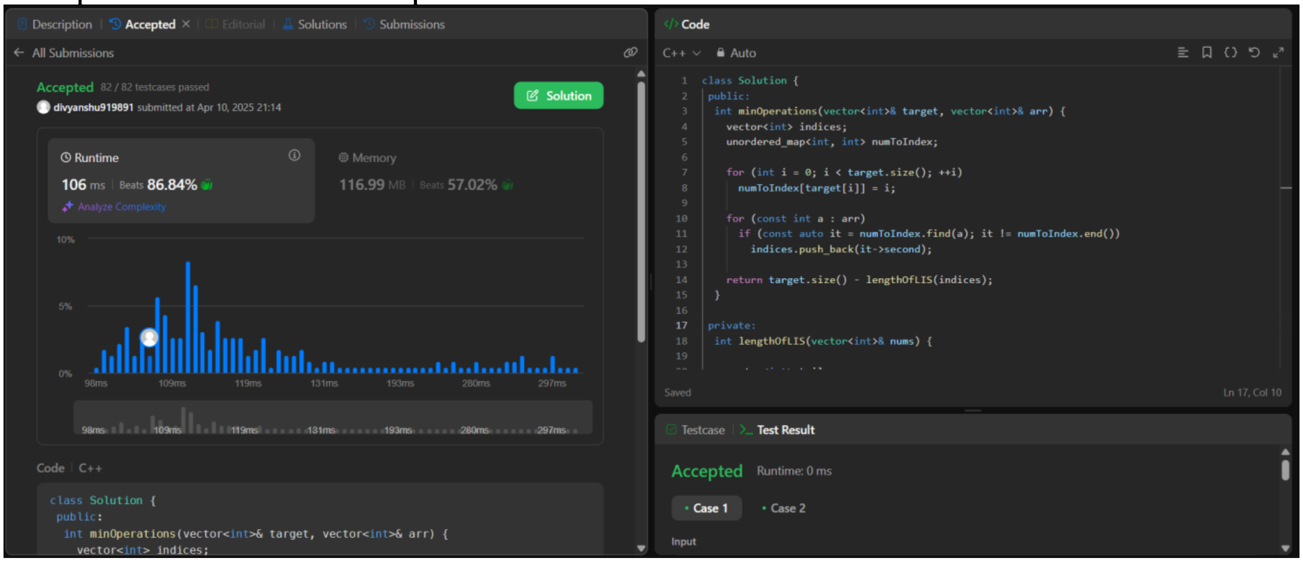
Problem Statement: You are given an array `target` that consists of **distinct** integers and another integer array `arr` that **can** have duplicates.

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.

2.Implementation and output:



Accepted 82 / 82 testcases passed

divyanshu919891 submitted at Apr 10, 2025 21:14

Runtime 106 ms | Beats 86.84%

Memory 116.99 MB | Beats 57.02%

Code | C++

```
class Solution {
 public:
  int minOperations(vector<int>& target, vector<int>& arr) {
    vector<int> indices;
```

```
class Solution {
 public:
  int minOperations(vector<int>& target, vector<int>& arr) {
    vector<int> indices;
    unordered_map<int, int> numToIndex;

    for (int i = 0; i < target.size(); ++i)
      numToIndex[target[i]] = i;

    for (const int a : arr)
      if (const auto it = numToIndex.find(a); it != numToIndex.end())
        indices.push_back(it->second);

    return target.size() - lengthOfLIS(indices);
  }

 private:
  int lengthOfLIS(vector<int>& nums) {

```

Accepted Runtime: 0 ms

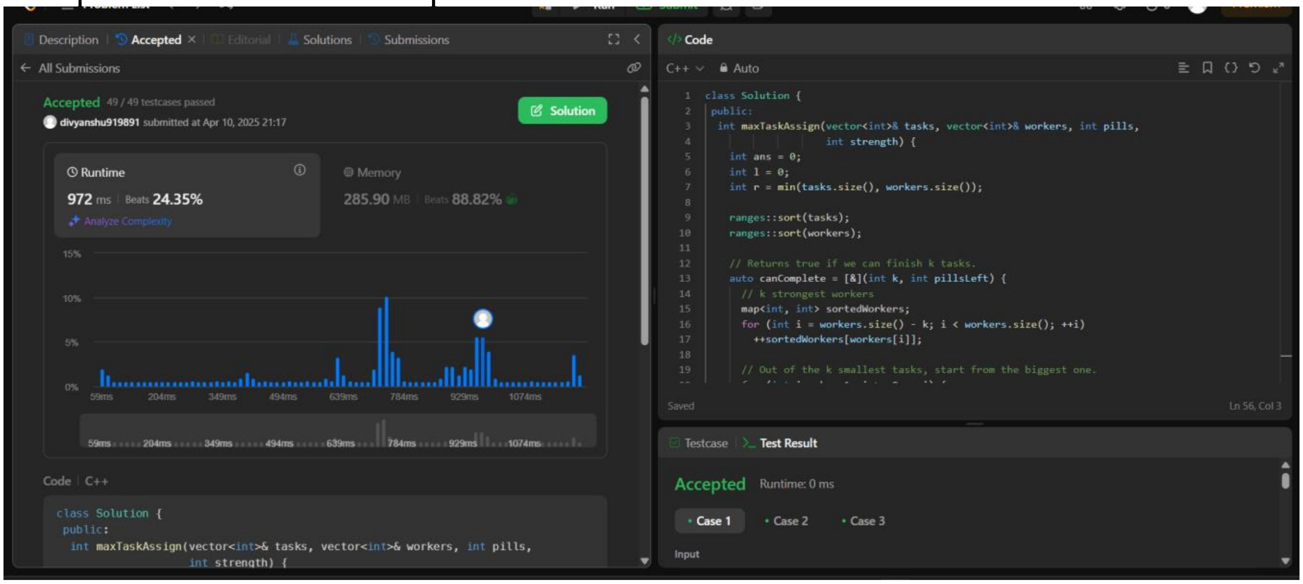
1.Aim:Maximum Number of Tasks You Can Assign

Problem Statement: You have `n` tasks and `m` workers. Each task has a strength requirement stored in a **0-indexed** integer array `tasks`, with the `i`th task requiring `tasks[i]` strength to complete. The strength of each worker is stored in a **0-indexed** integer array `workers`, with the `j`th worker having `workers[j]` strength. Each worker can only be assigned to a **single** task and must have a strength **greater than or equal** to the task's strength requirement (i.e., `workers[j] >= tasks[i]`).

Additionally, you have `pills` magical pills that will **increase a worker's strength** by `strength`. You can decide which workers receive the magical pills, however, you may only give each worker **at most one** magical pill.

Given the **0-indexed** integer arrays `tasks` and `workers` and the integers `pills` and `strength`, return *the **maximum** number of tasks that can be completed.*

2.Implementation And Output:



Accepted 49 / 49 testcases passed
divyanshu919891 submitted at Apr 10, 2025 21:17

Runtime
972 ms | Beats 24.35%

Memory
285.90 MB | Beats 88.82%

Code | C++

```
class Solution {
 public:
  int maxTaskAssign(vector<int>& tasks, vector<int>& workers, int pills,
                    int strength) {
```

```
class Solution {
 public:
  int maxTaskAssign(vector<int>& tasks, vector<int>& workers, int pills,
                    int strength) {
    int ans = 0;
    int l = 0;
    int r = min(tasks.size(), workers.size());

    ranges::sort(tasks);
    ranges::sort(workers);

    // Returns true if we can finish k tasks.
    auto canComplete = [&](int k, int pillsLeft) {
      // k strongest workers
      map<int, int> sortedWorkers;
      for (int i = workers.size() - k; i < workers.size(); ++i)
        ++sortedWorkers[workers[i]];

      // Out of the k smallest tasks, start from the biggest one.
```

Saved
Ln 56, Col 3

Accepted Runtime: 0 ms