Experiment 8

Student Name: Ramit Chaturvedi **UID:22BCS15597**
Branch: CSE **Section/Group:614-B**
Semester: 6 **Date of Performance:10-04-25**
Subject Name: AP LAB-II **Subject Code: 22CSP-351**

Q 1) **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.

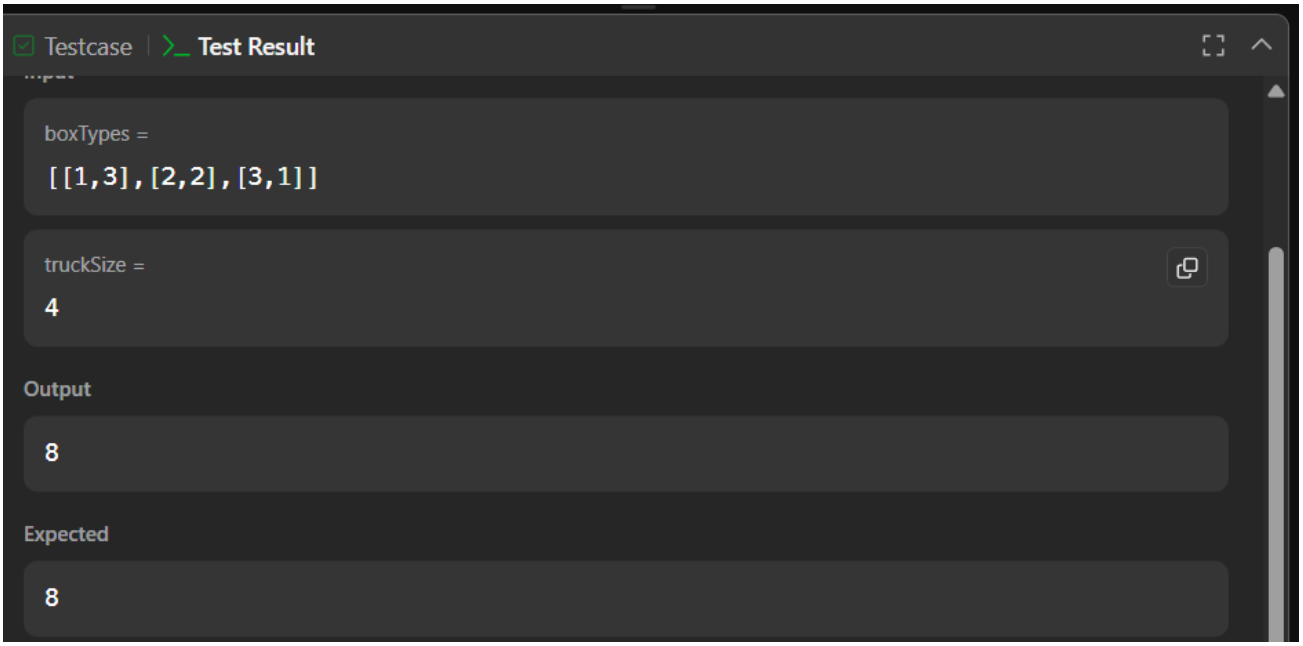
Return *the **maximum** total number of **units** that can be put on the truck.*

Code)

```
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;
  }
};
```

Output)



Q 2) **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].

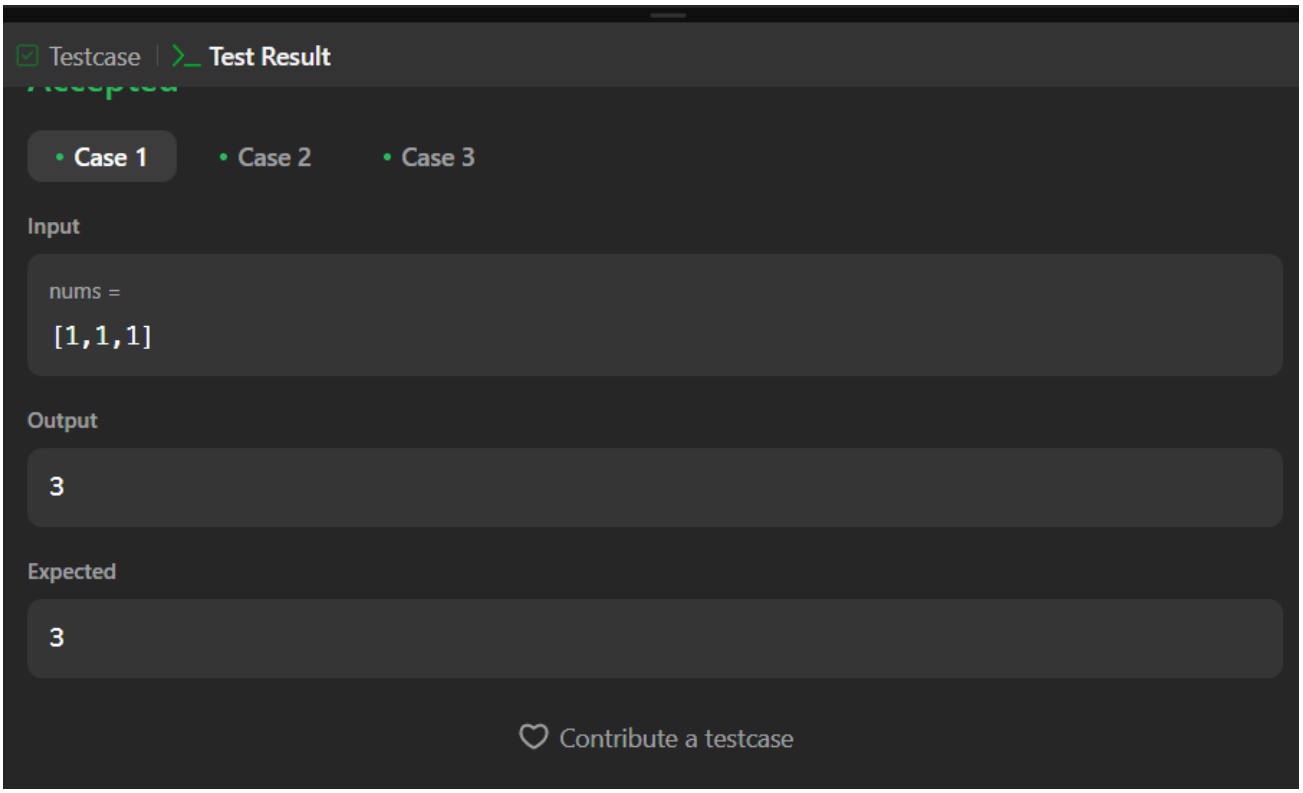
Return *the **minimum** number of operations needed to make* nums ***strictly increasing***.

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.

Code)

```
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;
  }
};
```

Output)



3) **Remove Stones to Minimize the Total**

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).

Code)

```
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;
  }
};
```

Output)

4) **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.

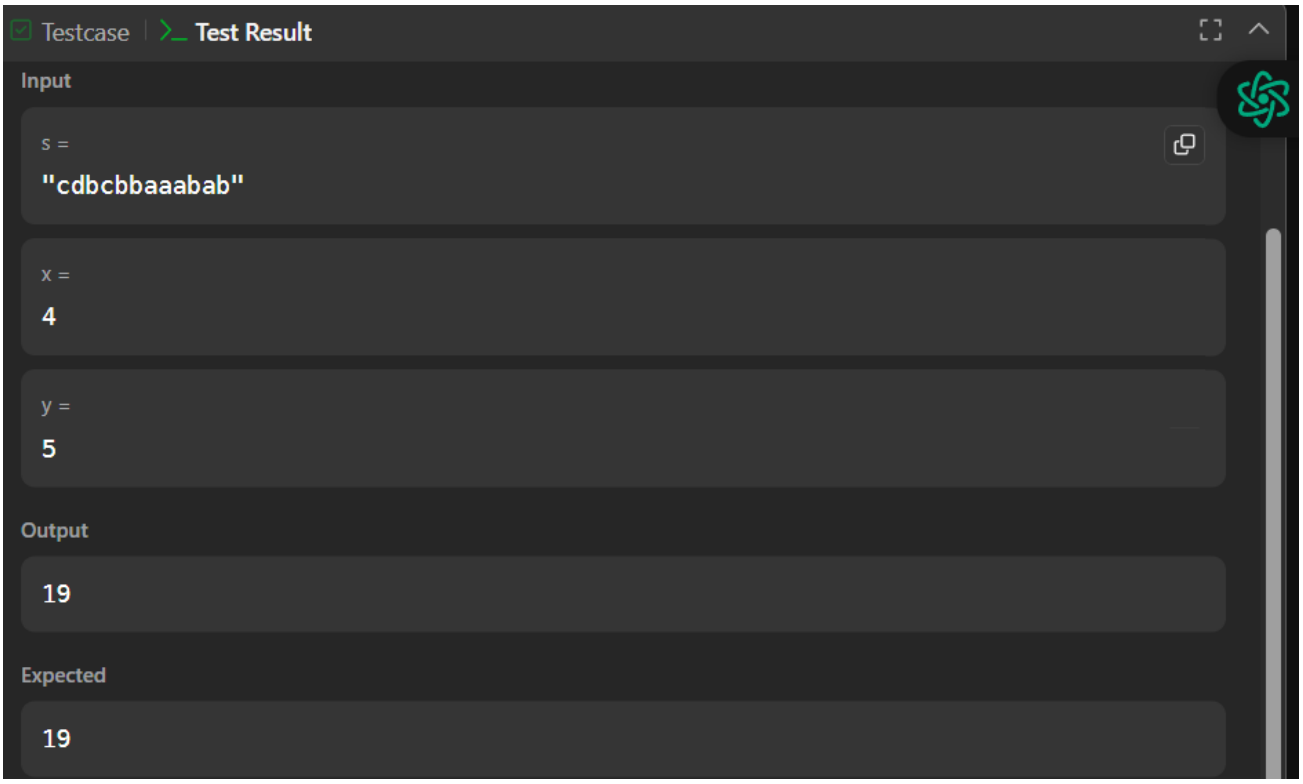
Code)

```
class Solution {
 public:
  int maximumGain(string s, int x, int y) {
    // The assumption that gain("ab") > gain("ba") while removing "ba" first is
```

```
    // optimal is contradicted. Only "b(ab)a" satisfies the condition of
    // preventing two "ba" removals, but after removing "ab", we can still
    // remove one "ba", resulting in a higher gain. Thus, removing "ba" first is
    // not optimal.
    return x > y ? gain(s, "ab", x, "ba", y) : gain(s, "ba", y, "ab", x);
  }

 private:
  // Returns the points gained by first removing sub1 ("ab" | "ba") from s with
  // point1, then removing sub2 ("ab" | "ba") from s with point2.
  int gain(const string& s, const string& sub1, int point1, const string& sub2,
           int point2) {
    int points = 0;
    vector<char> stack1;
    vector<char> stack2;

    // Remove "sub1" from s with point1 gain.
    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);
      }

    // Remove "sub2" from s with point2 gain.
    for (const char c : stack1)
      if (!stack2.empty() && stack2.back() == sub2[0] && c == sub2[1]) {
        stack2.pop_back();
        points += point2;
      } else {
        stack2.push_back(c);
      }
```

```
    return points;
  }
};
```

Output)



5) **Minimum 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.

Code)

```
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:
  // Same as 300. Longest Increasing Subsequence
  int lengthOfLIS(vector<int>& nums) {
    // tails[i] := the minimum tail of all the increasing subsequences having
    // length i + 1
    vector<int> tails;
    for (const int num : nums)
      if (tails.empty() || num > tails.back())
        tails.push_back(num);
      else
        tails[firstGreaterEqual(tails, num)] = num;
    return tails.size();
  }
```

```
 private:
  int firstGreaterEqual(const vector<int>& arr, int target) {
    return ranges::lower_bound(arr, target) - arr.begin();
  }
};
```

Output)