Experiment 8

Name: Manik Naharia
UID: 22BET10004
Branch: IT
Section: 22BET_IOT-701/A
Semester: 6th
Date of Performance: 28/03/25
Subject: Advanced Programming - 2
Subject Code: 22ITP-351

Problem 1. 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.

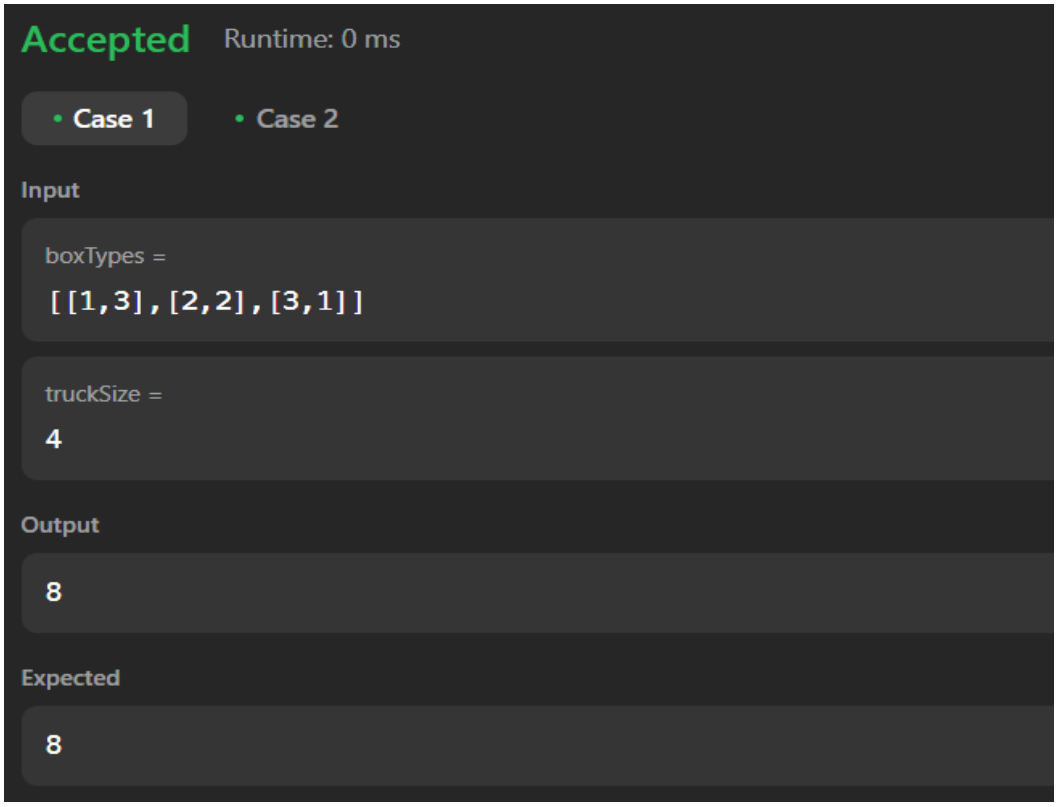
You are also given an integer truckSize, which is the **maximum** number of **boxes** that can be put on the truck. You can choose any boxes to put on the truck as long as the number of boxes does not exceed truckSize.

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) {
        priority_queue<pair<int,int>>pq;
        for(int i=0;i<boxTypes.size();i++){
            pq.push(make_pair(boxTypes[i][1],boxTypes[i][0]));
        }
        int ans = 0;
        while(!pq.empty() && truckSize>= pq.top().second){
            truckSize -= pq.top().second;
            ans += pq.top().second*pq.top().first;
            pq.pop();
        }
        if(!pq.empty() && truckSize){
            ans += pq.top().first*truckSize;
        }
        return ans;
    }
};
```

Output:



Problem 2. 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.

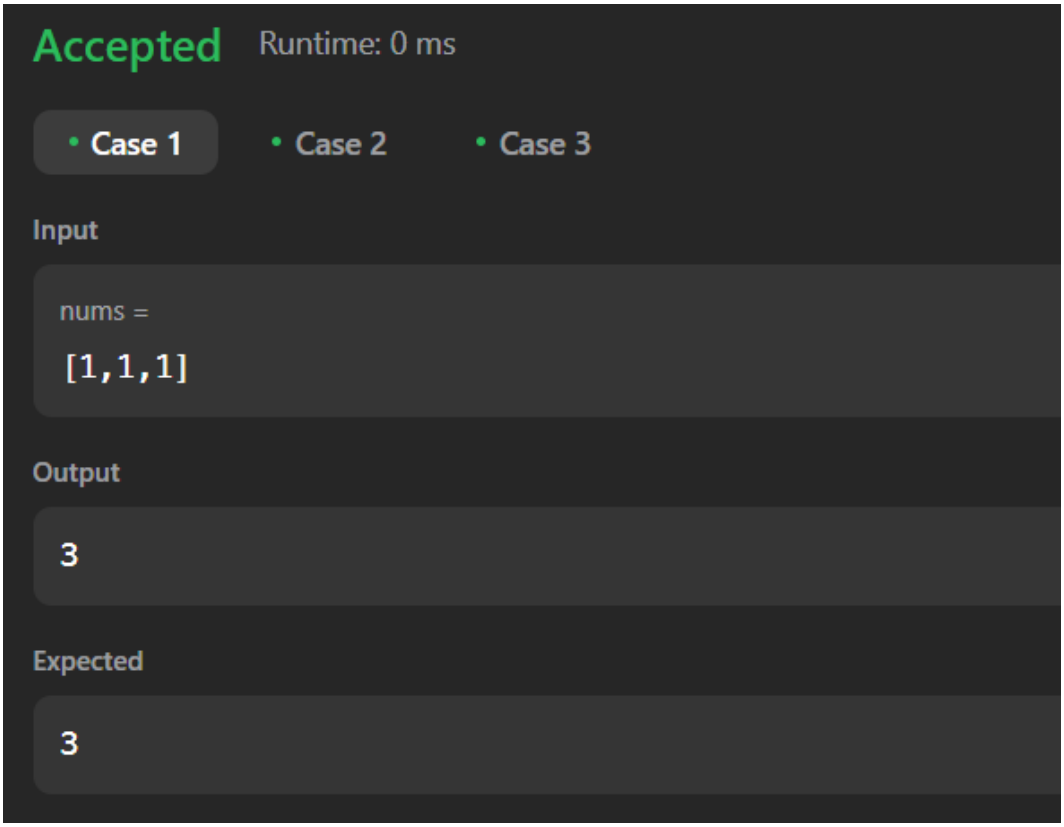
- For example, if nums = [1,2,3], you can choose to increment nums[1] to make nums = [1,**3**,3].

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 output=0;
        for(int i=0;i<nums.size()-1;i++){
            if(nums[i]<nums[i+1])
                continue;
            else{
                output=output+(nums[i]+1-nums[i+1]);
                nums[i+1]=nums[i]+1;
            }
        }
        return output;
    }
};
```

Output:



Problem 3. 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:

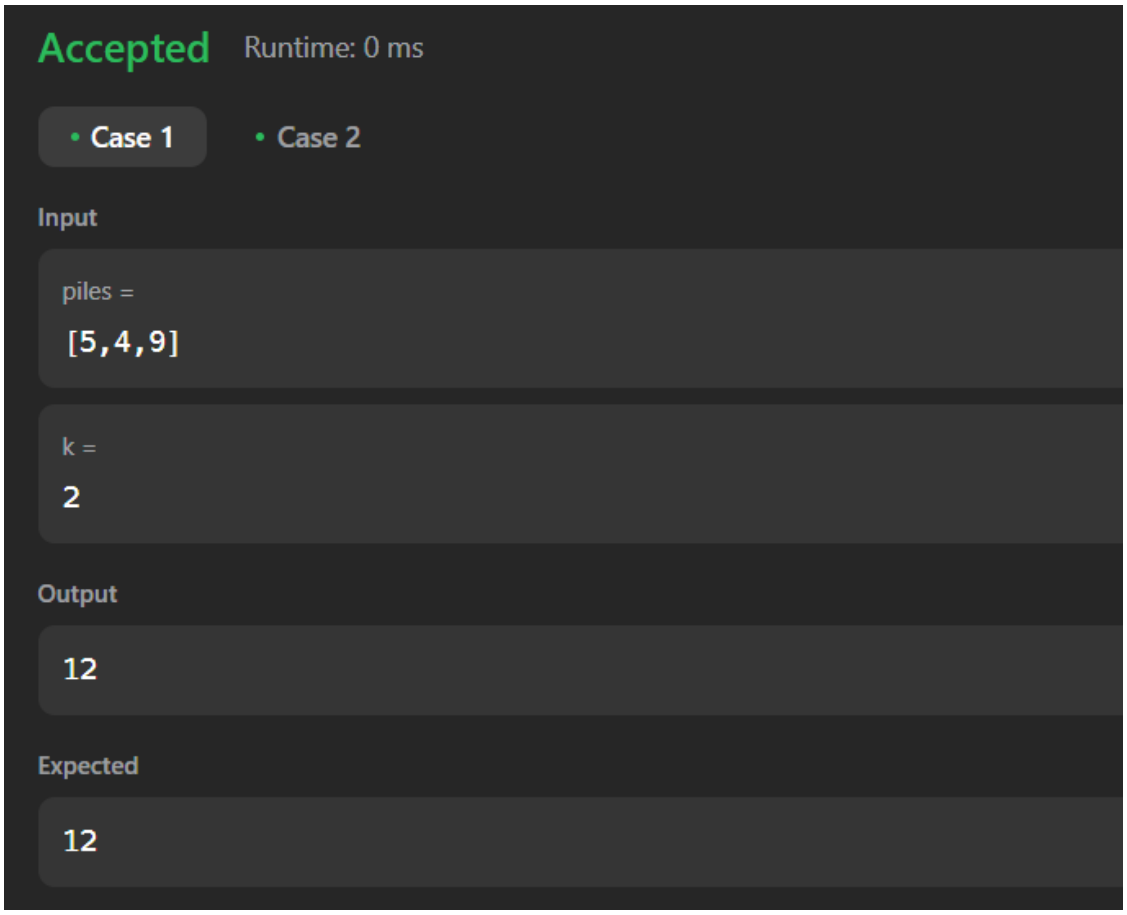
- Choose any piles[i] and **remove** floor(piles[i] / 2) stones from it.

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*.

Code:

```
class Solution {
public:
    int minStoneSum(vector<int>& A, int k) {
    priority_queue<int> pq(A.begin(), A.end());
    int res = accumulate(A.begin(), A.end(), 0);
    while (k--) {
       int a = pq.top();
       pq.pop();
       pq.push(a - a / 2);
       res -= a / 2;
    }
    return res;
  }
};
```

Output:



Problem 4. 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".

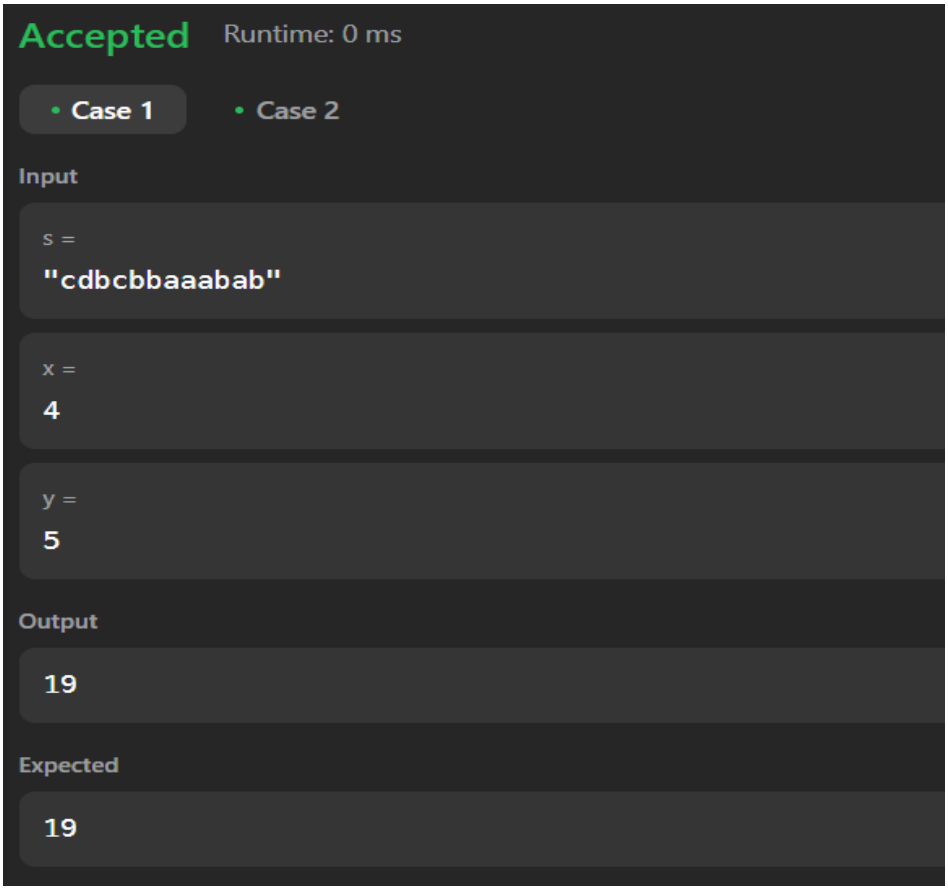
Return *the maximum points you can gain after applying the above operations on* s.

Code:

```
class Solution {
public:
    int maximumGain(string s, int x, int y) {
        int aCount = 0;
        int bCount = 0;
        int lesser = min(x, y);
        int result = 0;
        for (char c : s) {
            if (c > 'b') {
```

```
                result += min(aCount, bCount) * lesser;
                aCount = 0;
                bCount = 0;
            } else if (c == 'a') {
                if (x < y && bCount > 0) {
                    bCount--;
                    result += y;
                } else {
                    aCount++;
                }
            } else {
                if (x > y && aCount > 0) {
                    aCount--;
                    result += x;
                } else {
                    bCount++;
                }
            }
        }
        result += min(aCount, bCount) * lesser;
        return result;
    }
};
```

Output:



Problem 5. 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.

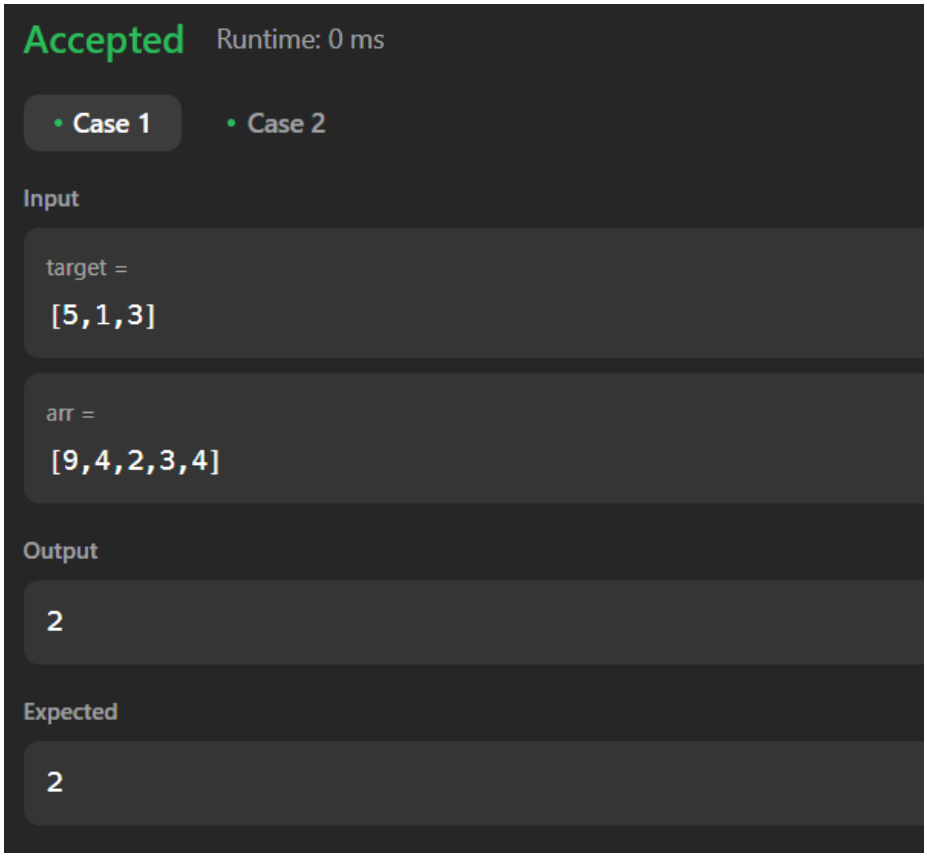
Return *the **minimum** number of operations needed to make* target *a **subsequence** of* arr.

Code:

```
class Solution {
public:
    int minOperations(vector<int>& target, vector<int>& arr) {
        unordered_map<int, int> mapping;
        int i = 0;
        for (auto& num : target)
            mapping[num] = ++i;

        vector<int> A;
        for (int& num : arr)
            if (mapping.find(num) != mapping.end())
                A.push_back(mapping[num]);
        return target.size() - lengthOfLIS(A);
    }
private:
    int lengthOfLIS(vector<int>& nums) {
        if (nums.empty()) return 0;
        vector<int> piles;
        for(int i=0; i<nums.size(); i++) {
            auto it = std::lower_bound(piles.begin(), piles.end(), nums[i]);
            if (it == piles.end())
                piles.push_back(nums[i]);
            else
                *it = nums[i];
        }
        return piles.size();
     }
};
```

Output:



Problem 6. 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 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.*

Code:

```
class Solution {
public:
    int maxTaskAssign(vector<int>& tasks, vector<int>& workers, int p, int strength) {
        int n = tasks.size(), m = workers.size();
        sort(tasks.begin(), tasks.end());
        sort(workers.begin(), workers.end());
        int lo = 0, hi = min(m, n);
        int ans;
```

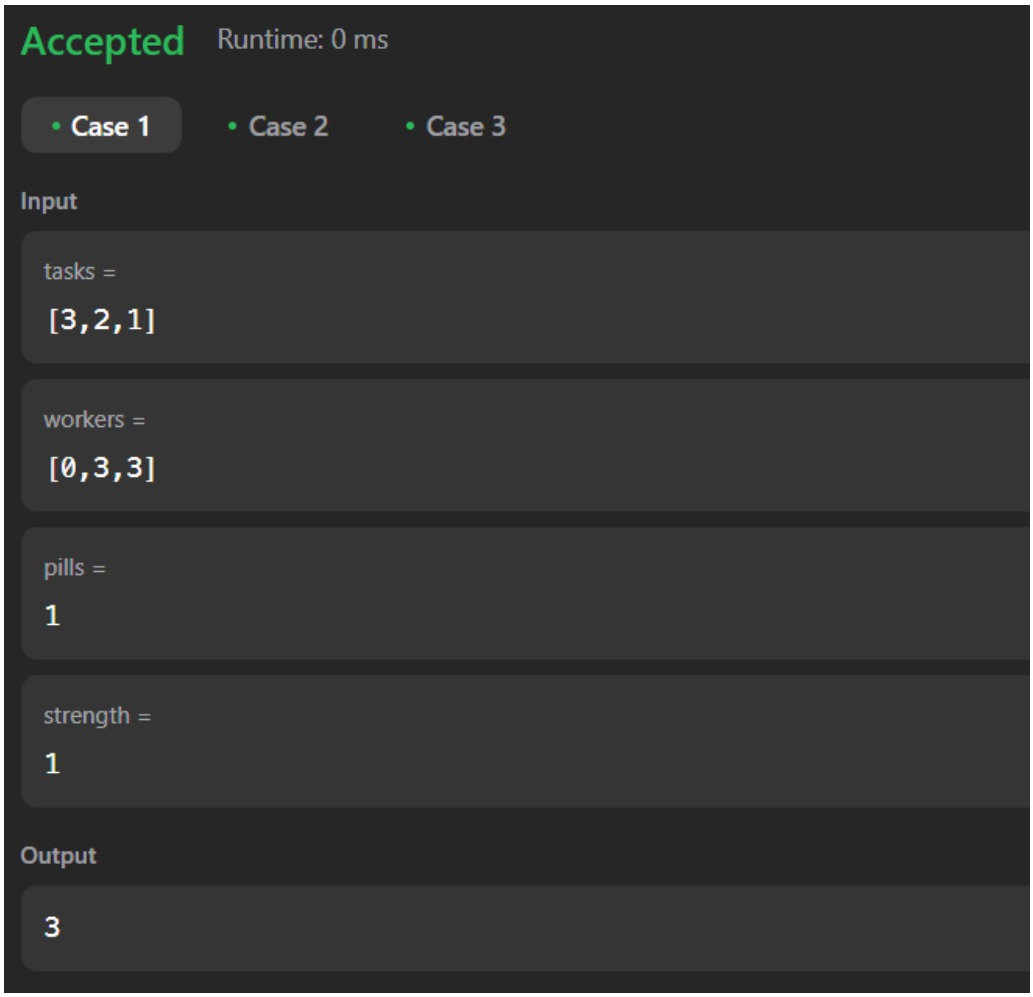
```
        while(lo <= hi) {
            int mid = lo + (hi - lo) / 2;
            int count = 0;
            bool flag = true;
            multiset<int> st(workers.begin(), workers.end());

            for(int i = mid - 1; i >= 0; i--) {
                auto it = prev(st.end());
                if(tasks[i] <= *it) {
                    st.erase(it);
                } else {
                    auto it = st.lower_bound(tasks[i] - strength);
                    if(it != st.end()) {
                        count++;
                        st.erase(it);
                    } else {
                        flag = false;
                        break;
                    }
                }

                if(count > p) {
                    flag = false;
                    break;
                }
            }

            if(flag) {
                ans = mid;
                lo = mid + 1;
            } else {
                hi = mid - 1;
            }
        }
        return ans;
    }
};
```

Output:


Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
tasks =
[3,2,1]
workers =
[0,3,3]
pills =
1
strength =
1
Output
3