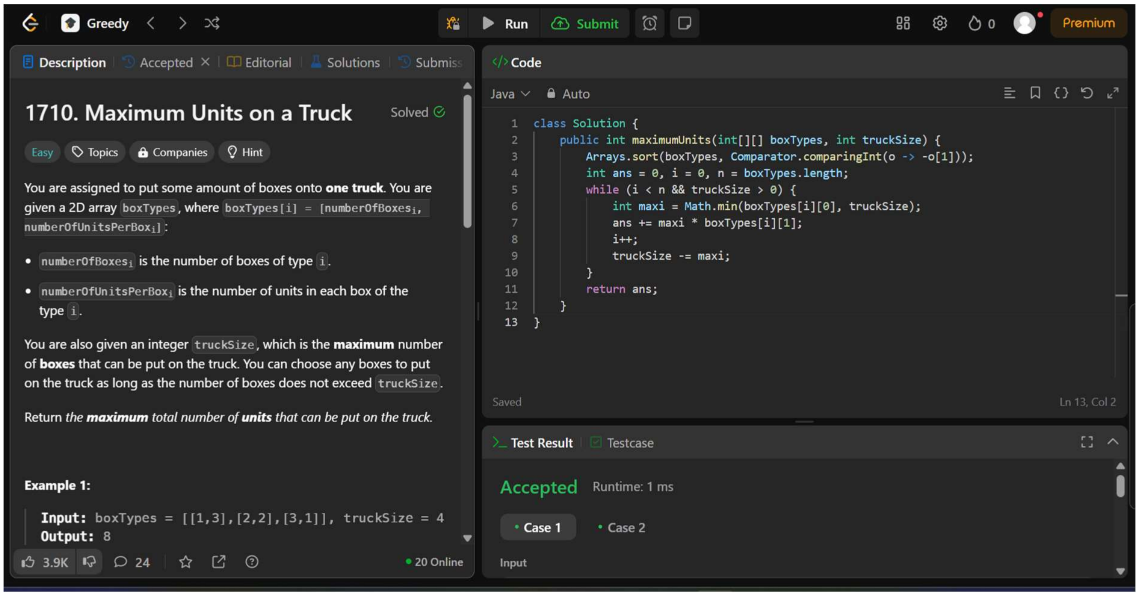
AP Experiment 8

SAHIL GUPTA

22BCS10608

IOT-614/B

1710. Maximum Units on a Truck



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.*

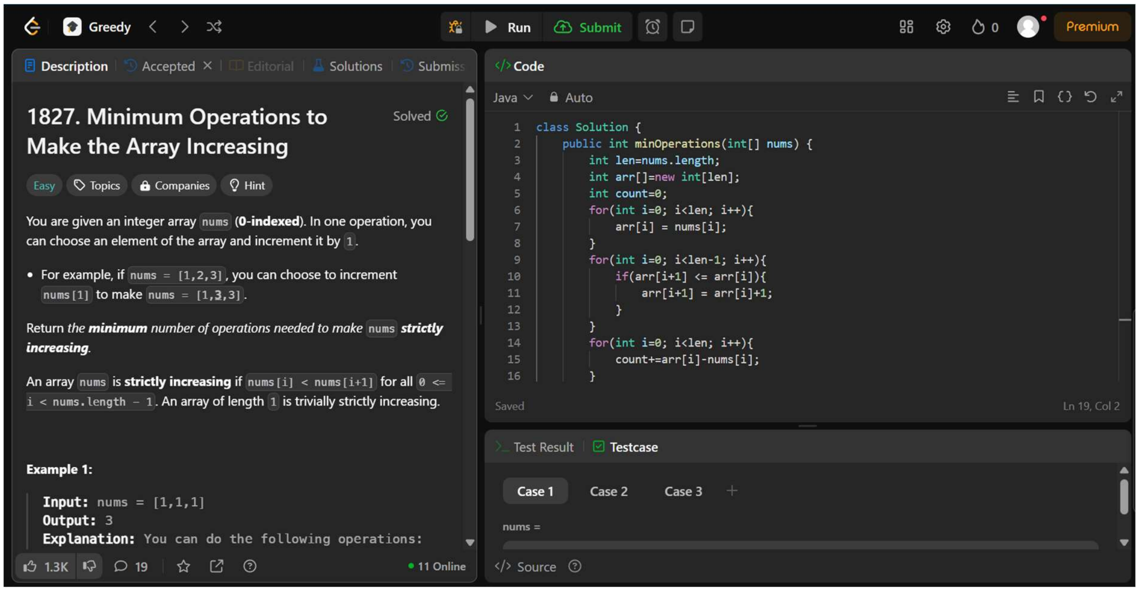
Example 1:

```
Input: boxTypes = [[1,3],[2,2],[3,1]], truckSize = 4
Output: 8
```

```java
class Solution {
    public int maximumUnits(int[][] boxTypes, int truckSize) {
        Arrays.sort(boxTypes, Comparator.comparingInt(o -> -o[1]));
        int ans = 0, i = 0, n = boxTypes.length;
        while (i < n && truckSize > 0) {
            int maxi = Math.min(boxTypes[i][0], truckSize);
            ans += maxi * boxTypes[i][1];
            i++;
            truckSize -= maxi;
        }
        return ans;
    }
}
```

Accepted Runtime: 1 ms

1827. Minimum Operations to Make the Array Increasing



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.

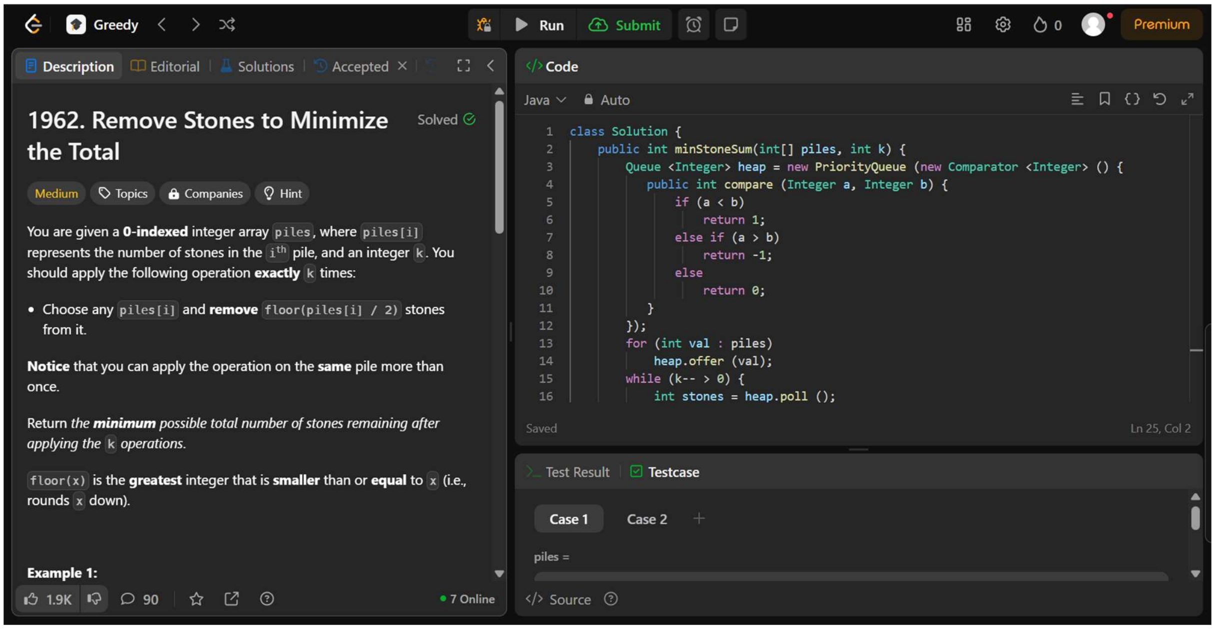
Example 1:

```
Input: nums = [1,1,1]
Output: 3
Explanation: You can do the following operations:
```

```java
class Solution {
    public int minOperations(int[] nums) {
        int len=nums.length;
        int arr[]=new int[len];
        int count=0;
        for(int i=0; i<len; i++){
            arr[i] = nums[i];
        }
        for(int i=0; i<len-1; i++){
            if(arr[i+1] <= arr[i]){
                arr[i+1] = arr[i]+1;
            }
        }
        for(int i=0; i<len; i++){
            count+=arr[i]-nums[i];
        }
```

nums =

1962. Remove Stones to Minimize the Total



1962. 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.

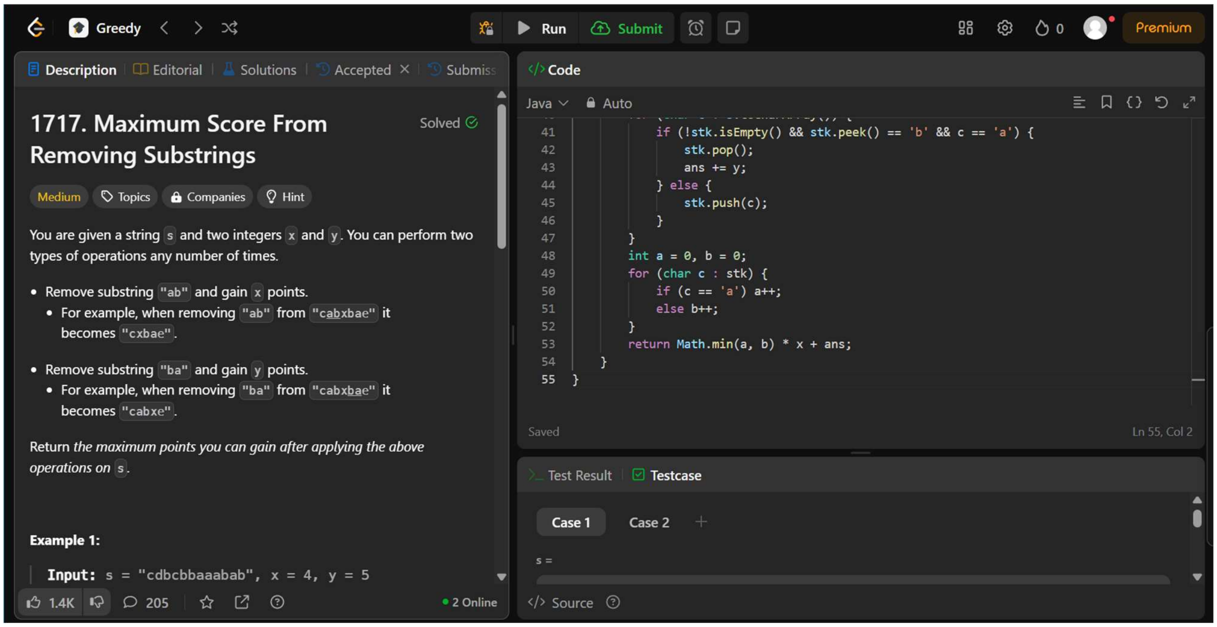
Return *the **minimum** possible total number of stones remaining after applying the* `k` *operations*.

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

Example 1:

```java
class Solution {
    public int minStoneSum(int[] piles, int k) {
        Queue <Integer> heap = new PriorityQueue (new Comparator <Integer> () {
            public int compare (Integer a, Integer b) {
                if (a < b)
                    return 1;
                else if (a > b)
                    return -1;
                else
                    return 0;
            }
        });
        for (int val : piles)
            heap.offer (val);
        while (k-- > 0) {
            int stones = heap.poll ();
```

1717. Maximum Score From Removing Substrings



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"`.

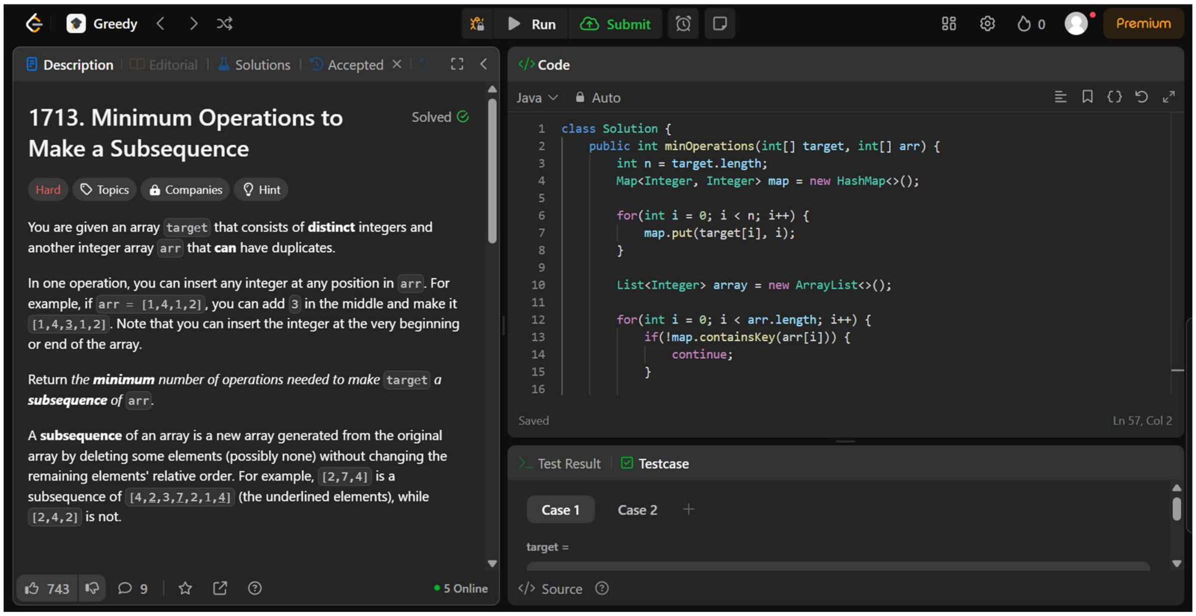
Return *the maximum points you can gain after applying the above operations on* `s`.

Example 1:

Input: `s = "cdbcbbaaabab", x = 4, y = 5`

```java
            if (!stk.isEmpty() && stk.peek() == 'b' && c == 'a') {
                stk.pop();
                ans += y;
            } else {
                stk.push(c);
            }
        }
        int a = 0, b = 0;
        for (char c : stk) {
            if (c == 'a') a++;
            else b++;
        }
        return Math.min(a, b) * x + ans;
    }
}
```

1713. Minimum Operations to Make a Subsequence



1713. 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.

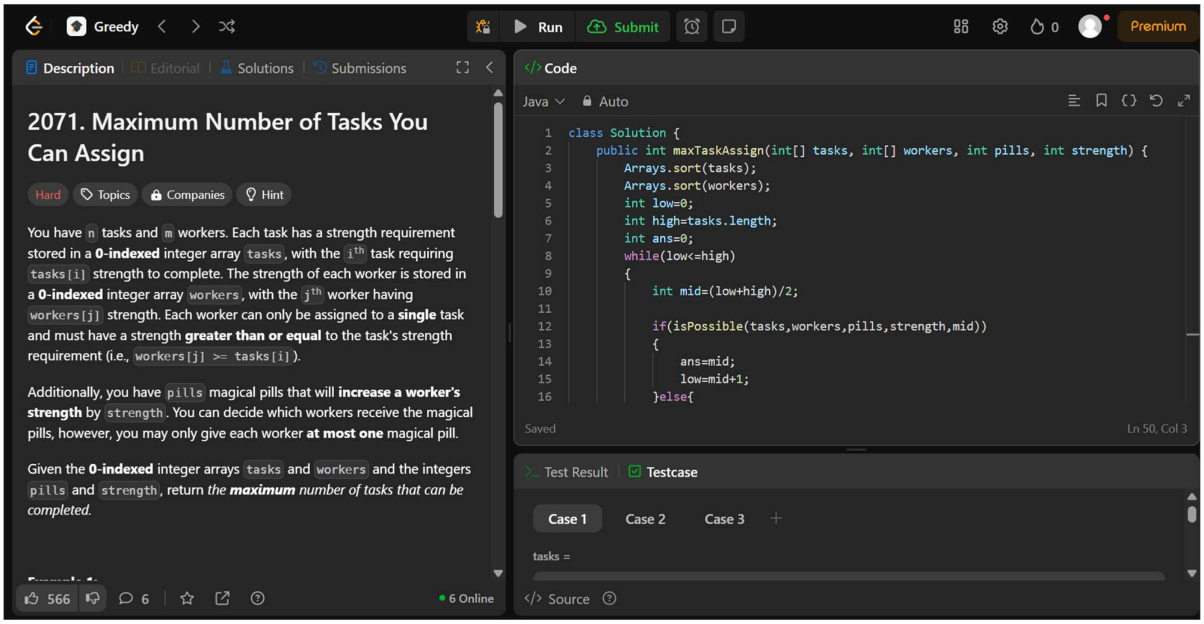
In one operation, you can insert any integer at any position in `arr`. For example, if `arr = [1,4,1,2]`, you can add `3` in the middle and make it `[1,4,3,1,2]`. Note that you can insert the integer at the very beginning or end of the array.

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.

```java
class Solution {
    public int minOperations(int[] target, int[] arr) {
        int n = target.length;
        Map<Integer, Integer> map = new HashMap<>();

        for(int i = 0; i < n; i++) {
            map.put(target[i], i);
        }

        List<Integer> array = new ArrayList<>();

        for(int i = 0; i < arr.length; i++) {
            if(!map.containsKey(arr[i])) {
                continue;
            }

```

2071. Maximum Number of Tasks You Can Assign



2071. Maximum Number of Tasks You Can Assign

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.*

```java
class Solution {
    public int maxTaskAssign(int[] tasks, int[] workers, int pills, int strength) {
        Arrays.sort(tasks);
        Arrays.sort(workers);
        int low=0;
        int high=tasks.length;
        int ans=0;
        while(low<=high)
        {
            int mid=(low+high)/2;

            if(isPossible(tasks,workers,pills,strength,mid))
            {
                ans=mid;
                low=mid+1;
            }else{
```