Experiment 8

Student Name: KHUSHBOO **UID: 22BET10197**
Branch: BE-IT **Section/Group: 22BET_IOT_701/A**
Semester: 6th **Date of Performnce:28-03-25**
Subject Name: AP-II **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] = [numberOfBoxesi, numberOfUnitsPerBoxi]:

Algorithm:

1. **Sort:** Sort the boxTypes array in descending order based on the number of units per box (the second element in each inner vector).
2. **Iterate and Load:** Iterate through the sorted boxTypes. For each box type:
   - Calculate how many boxes of this type we can fit in the remaining truck capacity. This will be the minimum of the available number of boxes of this type and the remaining truck capacity.
   - Add the total units loaded from these boxes to our overall totalUnits.
   - Decrease the remaining truck capacity by the number of boxes loaded.
   - If the truck capacity becomes zero, we've filled the truck, so we can stop.
3. **Return:** Return the totalUnits.

Code:

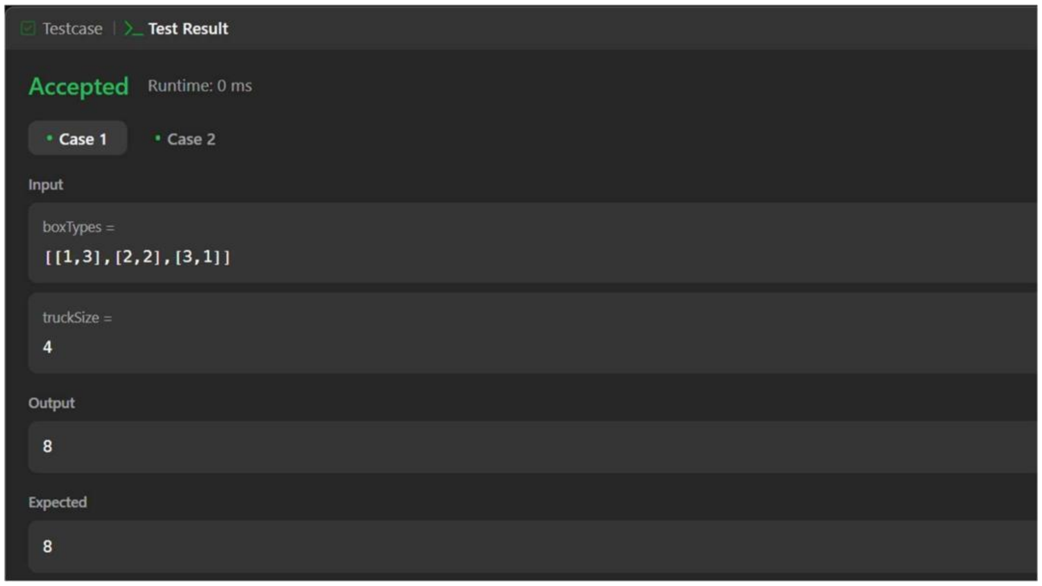
```
class Solution {
public:
    int maximumUnits(vector<vector<int>>& boxTypes, int truckSize) {
        sort(boxTypes.begin(), boxTypes.end(), [](const vector<int>& a, const vector<int>& b) {
            return a[1] > b[1];
        });

        int totalUnits = 0;
        for (const auto& boxType : boxTypes) {
            int numBoxes = boxType[0];
            int unitsPerBox = boxType[1];

            int canLoad = min(numBoxes, truckSize);
            totalUnits += canLoad * unitsPerBox;
            truckSize -= canLoad;

            if (truckSize == 0) {
                break;
            }
        }
        return totalUnits;
    }
};
```

Output:



Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
boxTypes =
[[1,3],[2,2],[3,1]]

truckSize =
4
```

Output

```
8
```

Expected

```
8
```

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.

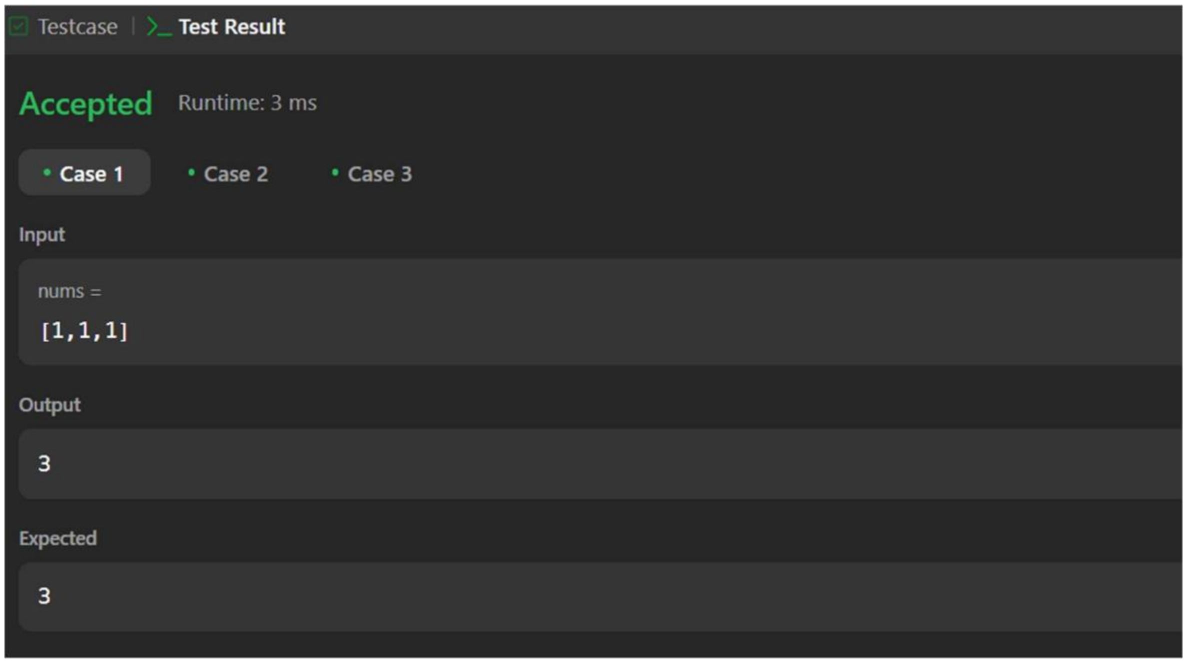
Algorithm:

1. Initialize: Initialize an operations counter to 0.
2. Iterate: Iterate through the array starting from the second element (index 1).
3. Check and Increment: For each element nums[i], compare it with the previous element nums[i-1].
   - If nums[i] <= nums[i-1], we need to make nums[i] greater. The minimum increment needed is nums[i-1] - nums[i] + 1.
   - Add this increment to operations.
   - Update nums[i] to nums[i-1] + 1 to ensure the increasing order.
4. Return: Return the total operations.

Code:

```
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]) {
                int diff = nums[i - 1] - nums[i] + 1;
                operations += diff;
                nums[i] += diff;
            }
        }
        return operations;
    }
};
```

Output:



Testcase | Test Result

Accepted Runtime: 3 ms

Case 1 Case 2 Case 3

Input

```
nums =
[1,1,1]
```

Output

```
3
```

Expected

```
3
```

Problem 3. You are given a 0-indexed integer array piles, where piles[i] represents the number of stones in the ith 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.

Algorithm:

1. **Use a Max-Heap:** Maintain a max-heap (priority queue in C++) to store the number of stones in each pile. This allows us to efficiently access the pile with the maximum number of stones.
2. **Initialize:** Insert the initial number of stones in each pile into the max-heap.
3. **Perform Operations:** Repeat k times (or until the heap is empty):
   - Extract the maximum number of stones from the top of the heap.
   - Calculate the number of stones to remove: floor(maxStones / 2).
   - Update the number of stones in that pile: maxStones - floor(maxStones / 2).
   - Insert the updated number of stones back into the max-heap (if it's not zero).
4. **Calculate Total:** After performing k operations, iterate through the remaining elements in the max-heap and sum them up to get the minimum total number of stones.

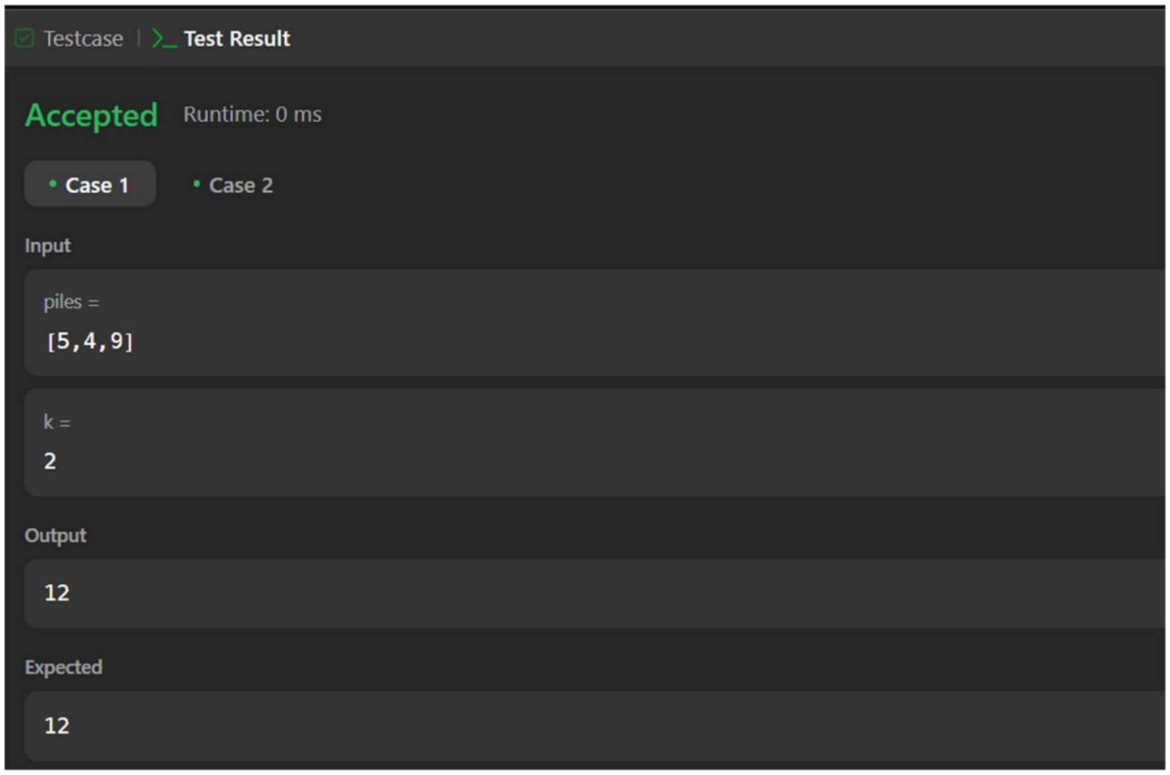
Code:

```
class Solution {
public:
    int minStoneSum(vector<int>& piles, int k) {
        priority_queue<int> pq(piles.begin(), piles.end());

        for (int i = 0; i < k; ++i) {
            if (pq.empty()) {
                break;
            }
            int maxStones = pq.top();
            pq.pop();
            int removedStones = floor(maxStones / 2.0);
            pq.push(maxStones - removedStones);
        }

        int minSum = 0;
        while (!pq.empty()) {
            minSum += pq.top();
            pq.pop();
        }
        return minSum;
    }
};
```

Output:



```
Input
piles =
[5,4,9]
k =
2
Output
12
Expected
12
```

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.

Algorithm:

1. **Identify Higher Score:** Determine which substring has the higher score. Let's say first is the substring with the higher score (firstScore) and second is the other substring (secondScore). If the scores are equal, the order doesn't matter.
2. **Iterate and Remove First Substring:** Iterate through the string and repeatedly find and remove all

occurrences of the first substring. For each removal, add firstScore to the total score. To avoid issues with overlapping matches, after removing a substring, you might need to adjust your search starting position. A simple way is to rebuild the string after each removal or use a more careful indexing approach.

3. **Iterate and Remove Second Substring:** After all occurrences of the first substring have been removed, iterate through the remaining string and repeatedly find and remove all occurrences of the second substring. For each removal, add secondScore to the total score.
4. **Return:** Return the total accumulated score.

Code:

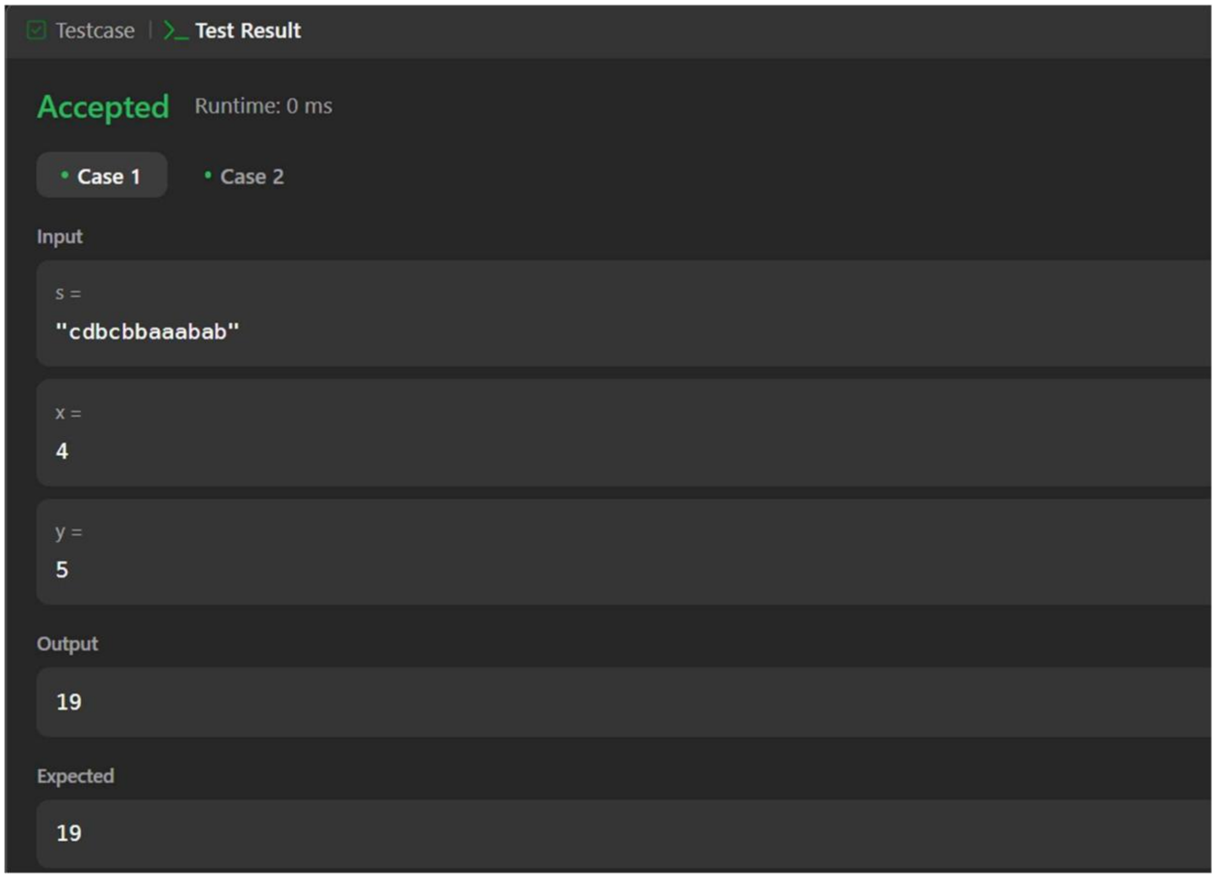
```
class Solution {
public:
    int maximumGain(string s, int x, int y) {
         int totalScore = 0;
        string s1 = "ab";
        string s2 = "ba";
        int score1 = x;
        int score2 = y;

        auto removeSubstrings = [&](string& str, const string& sub, int score) {
            string temp = "";
            for (char ch : str) {
                temp += ch;
                if (temp.length() >= sub.length() && temp.substr(temp.length() - sub.length()) == sub) {
                    totalScore += score;
                    temp.resize(temp.length() - sub.length());
                }
            }
            str = temp;
        };

        if (score1 >= score2) {
            removeSubstrings(s, s1, score1);
            removeSubstrings(s, s2, score2);
        } else {
            removeSubstrings(s, s2, score2);
            removeSubstrings(s, s1, score1);
        }

        return totalScore;
    }
};
```

Output:



Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
s =
"cdbcbbaaabab"
x =
4
y =
5
```

Output

```
19
```

Expected

```
19
```

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.

Algorithm:

1. **Map Target Indices:** Create a map to store the index of each element in the target array. This allows for quick lookups.
2. **Build Increasing Subsequence:** Iterate through the arr. For each element num in arr:
   - Check if num exists in the target array. If not, we can ignore it as it won't contribute to making target a subsequence.
   - If num is in target at index targetIndex, we want to find the longest increasing subsequence of the indices in target that we have encountered so far in arr.
   - Maintain a list (or vector) tails which stores the smallest tail of all increasing subsequences of the target indices found in arr so far.
   - For the current targetIndex:
     - If tails is empty or targetIndex is greater than the last element of tails, it means we can extend the longest increasing subsequence, so append targetIndex to tails.

- Otherwise, use binary search (specifically lower_bound) to find the smallest element in tails that is greater than or equal to targetIndex. Replace that element with targetIndex. This maintains the property that tails stores the smallest tails of increasing subsequences.

3. **Calculate Operations:** The length of the tails list at the end will be the length of the longest subsequence of target that we can form using elements from arr in the correct order. The minimum number of operations (insertions) needed is target.size() - tails.size().

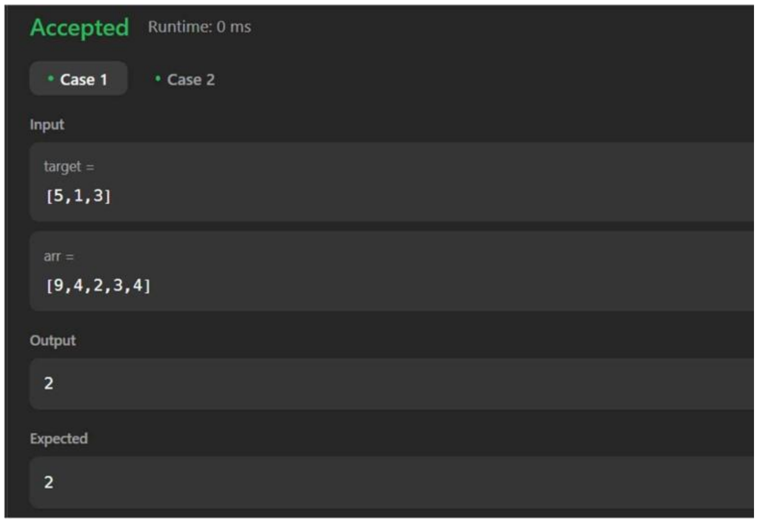
Code:

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

        vector<int> tails;
        for (int num : arr) {
            if (targetIndex.count(num)) {
                int index = targetIndex[num];
                auto it = lower_bound(tails.begin(), tails.end(), index);
                if (it == tails.end()) {
                    tails.push_back(index);
                } else {
                    *it = index;
                }
            }
        }

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

Output:



```
Input
target =
[5,1,3]
arr =
[9,4,2,3,4]
Output
2
Expected
2
```

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 ith task requiring tasks[i] strength to complete. The strength of each worker is stored in 0-indexed integer array workers, with the jth 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]).

Algorithm:

1. **Sort:** Sort workers and tasks.
2. **Binary Search:** Search for the maximum number of assignable tasks (k).
3. **canAssign(k):**
   - Take k weakest workers and k easiest tasks.
   - Use a multiset for worker strengths.
   - Iterate through tasks (hardest to easiest):
     - Find a strong enough worker (or use a pill on the strongest).
     - If assignment is possible for all k tasks, return true.
   - Return false.
4. **Return:** The largest k for which canAssign(k) is true.

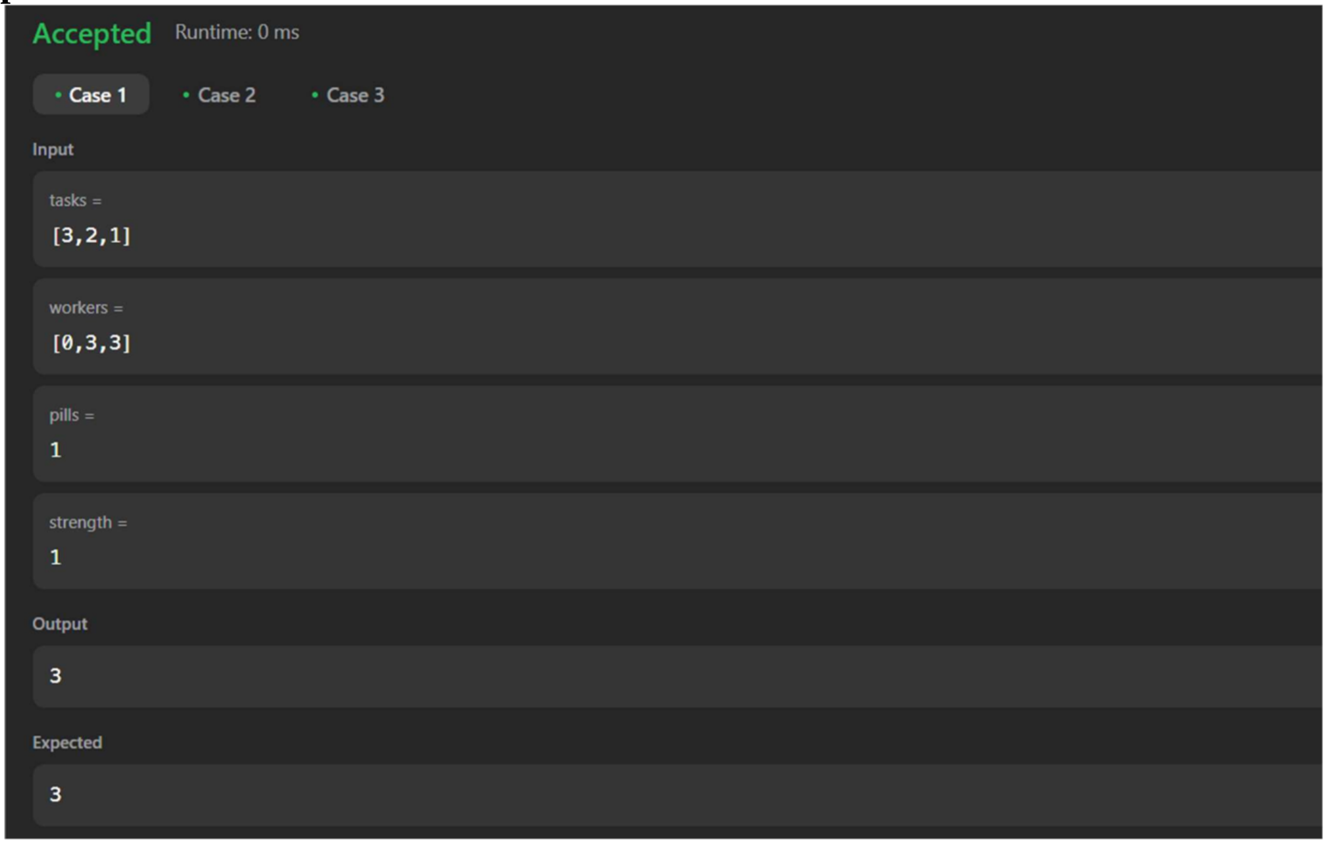
Code:

```
class Solution {
public:
    int maxTaskAssign(vector<int>& tasks, vector<int>& workers, int pills, int strength) {
        int n = workers.size();
        int m = tasks.size();
        sort(workers.begin(), workers.end());
        sort(tasks.begin(), tasks.end());

        auto canAssign = [&](int k) {
            if (k > n || k > m) return false;
            multiset<int> workerSet(workers.begin(), workers.begin() + k);
            int currentPills = pills;
            for (int i = k - 1; i >= 0; --i) {
                auto it = workerSet.lower_bound(tasks[i]);
                if (it != workerSet.end()) {
                    workerSet.erase(it);
                } else if (currentPills > 0 && !workerSet.empty()) {
                    int strongestWorker = *workerSet.rbegin();
                    if (strongestWorker + strength >= tasks[i]) {
                        workerSet.erase(prev(workerSet.end()));
                        currentPills--;
                    } else {
                        return false;
                    }
                } else {
                    return false;
                }
            }
            return true;
        };

        int left = 0, right = min(n, m);
        int ans = 0;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (canAssign(mid)) {
                ans = mid;
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return ans;
    }
};
```

Output:



```
Input
tasks =
[3,2,1]
workers =
[0,3,3]
pills =
1
strength =
1
Output
3
Expected
3
```