Introduction

LeetCode is an online platform designed for coding practice and preparation for technical interviews. It offers a wide range of problems covering various algorithms, data structures, and other computer science topics.

Getting Started

- **Registration:** Easy and straightforward; free and premium options available.
- **User Interface:** Clean and intuitive; problems are categorized by difficulty, topic, and company tags.

Problem-Solving

- **Problem Types:** Includes algorithms, database, shell scripting, and concurrency.
- **Difficulty Levels:** Divided into Easy, Medium, and Hard.
- **Problem Format:** Each problem has a description, input/output examples, constraints, and a section for user submissions.

Coding Environment

- **Languages Supported:** Multiple programming languages such as Python, Java, C++, etc.
- **Code Editor:** Built-in editor with features like syntax highlighting, auto-completion, and debugging tools.

Features

- **Test Cases:** Problems come with sample and custom test cases for thorough testing.
- **Solutions:** Community and official solutions available for learning different approaches.
- **Discussion Forums:** Active community discussions for doubts, tips, and alternative solutions.
- **Contests:** Regular coding contests for skill assessment and improvement.

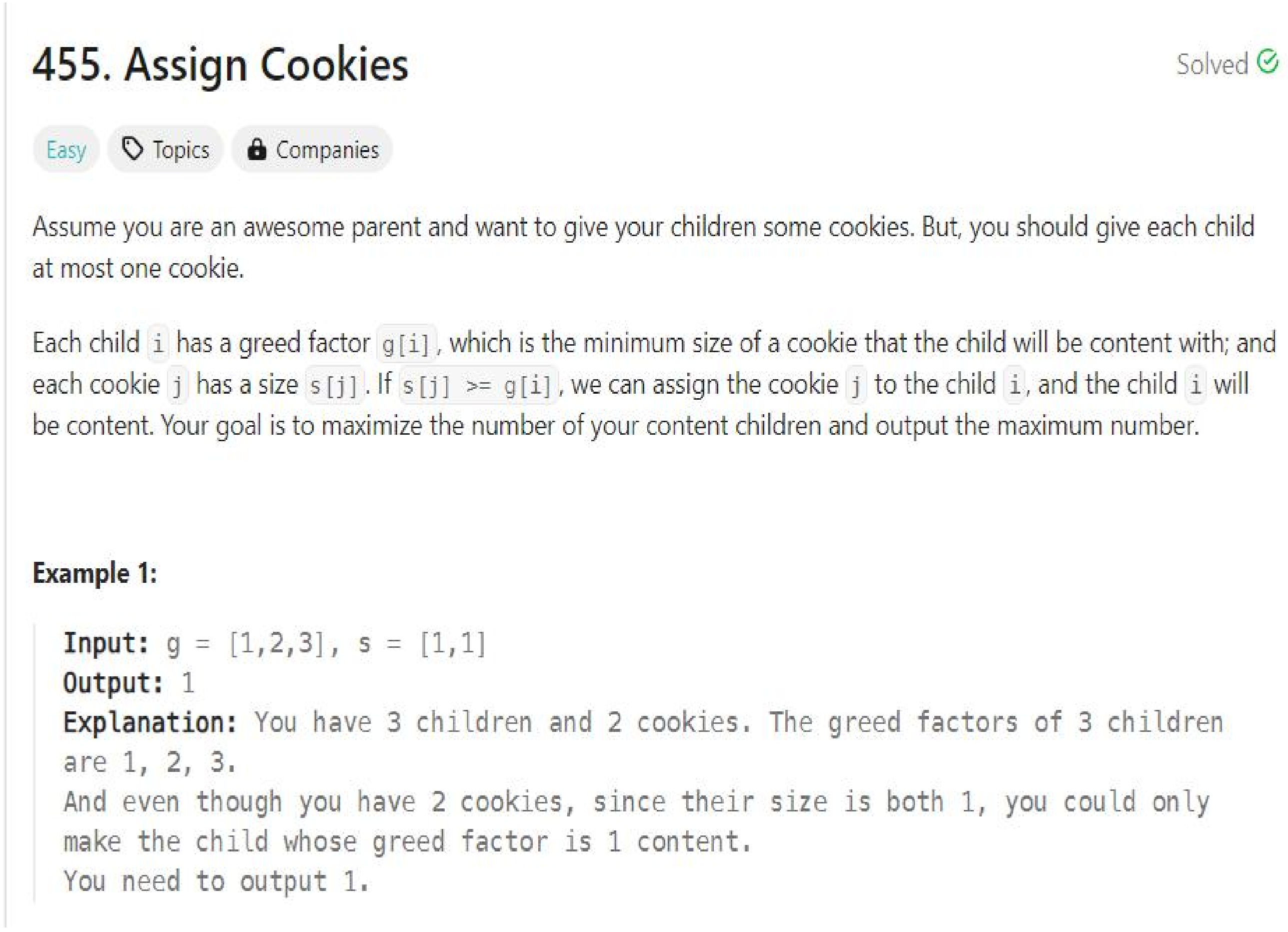
Premium Membership

- **Additional Features:** Access to premium problems, company-specific questions, and interview simulations.
- **Value:** Beneficial for serious job seekers preparing for interviews at top tech companies.

Learning and Improvement

- **Practice Effectiveness:** Regular practice leads to improved problem-solving skills and better understanding of algorithms and data structures.
- **Tracking Progress:** Features like progress tracking and analytics help users monitor their improvement over time.

Problem 1:



455. Assign Cookies

Solved ✓

Easy · Topics · Companies

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie.

Each child `i` has a greed factor `g[i]`, which is the minimum size of a cookie that the child will be content with; and each cookie `j` has a size `s[j]`. If `s[j] >= g[i]`, we can assign the cookie `j` to the child `i`, and the child `i` will be content. Your goal is to maximize the number of your content children and output the maximum number.

Example 1:

```
Input: g = [1,2,3], s = [1,1]
Output: 1
Explanation: You have 3 children and 2 cookies. The greed factors of 3 children
are 1, 2, 3.
And even though you have 2 cookies, since their size is both 1, you could only
make the child whose greed factor is 1 content.
You need to output 1.
```

Approach:

Greedy algorithm

- **Sort the Lists:**
  - ● First, the children's greed factors (`g`) and the cookies' sizes (`s`) are sorted in ascending order.
  - Sorting helps in efficiently matching the smallest available cookie that can satisfy the child's minimum greed factor.
- **Two-Pointer Technique:**
  - ● Two pointers, `i` and `j`, are initialized to 0.
    - `i` points to the current child in the sorted list of greed factors `g`.
    - `j` points to the current cookie in the sorted list of cookie sizes `s`.
- **Iterate and Match:**

- ● The algorithm uses a while loop to iterate through both lists until one of the pointers reaches the end of its respective list.
- Within the loop, it checks if the current cookie (`s[j]`) can satisfy the current child's greed factor (`g[i]`).
    - If `s[j] >= g[i]`, it means the current cookie can make the current child content.
        - The number of content children (`contentChildren`) is incremented.
        - Move to the next child by incrementing the pointer `i`.
    - Whether or not the current cookie satisfies the current child, move to the next cookie by incrementing the pointer `j`.

- **Return the Result:**

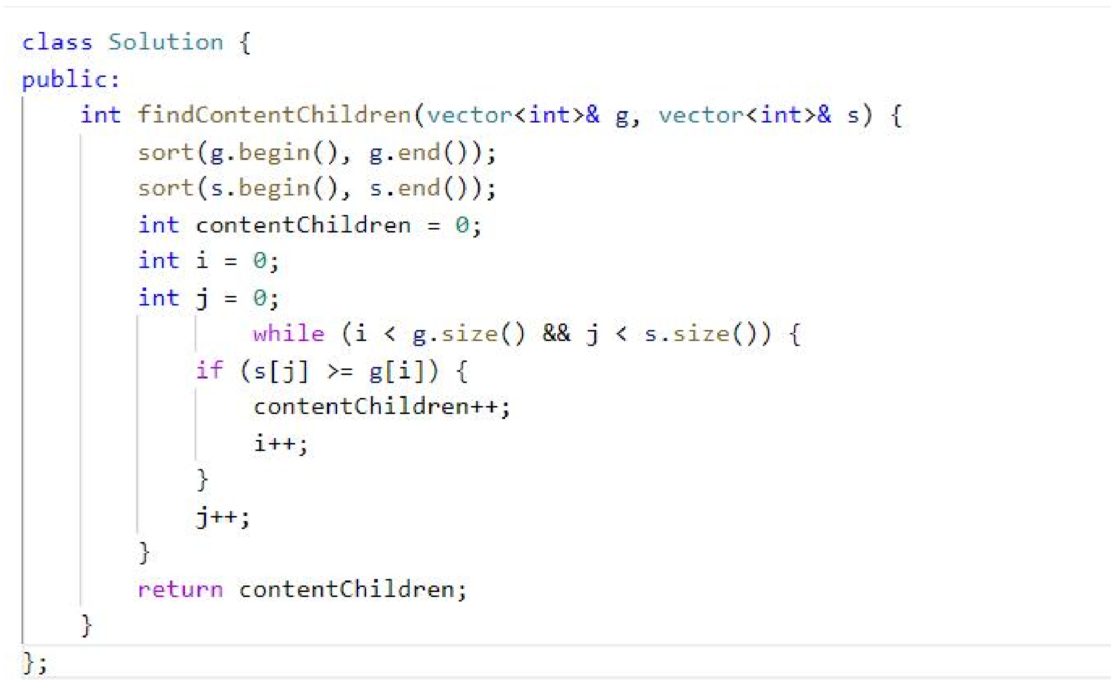
- ● After exiting the loop, the algorithm returns the count of content children (`contentChildren`).

Summary of the Approach

1. **Sorting:** By sorting both the greed factors and the cookie sizes, the solution leverages the ordered nature to efficiently find matches.
2. **Two-Pointer Technique:** The two-pointer technique allows simultaneous traversal of both lists, ensuring that each comparison and assignment is optimal.
3. **Maximizing Content Children:** By always attempting to satisfy the next most greedy child with the smallest sufficient cookie, the solution maximizes the number of content children.

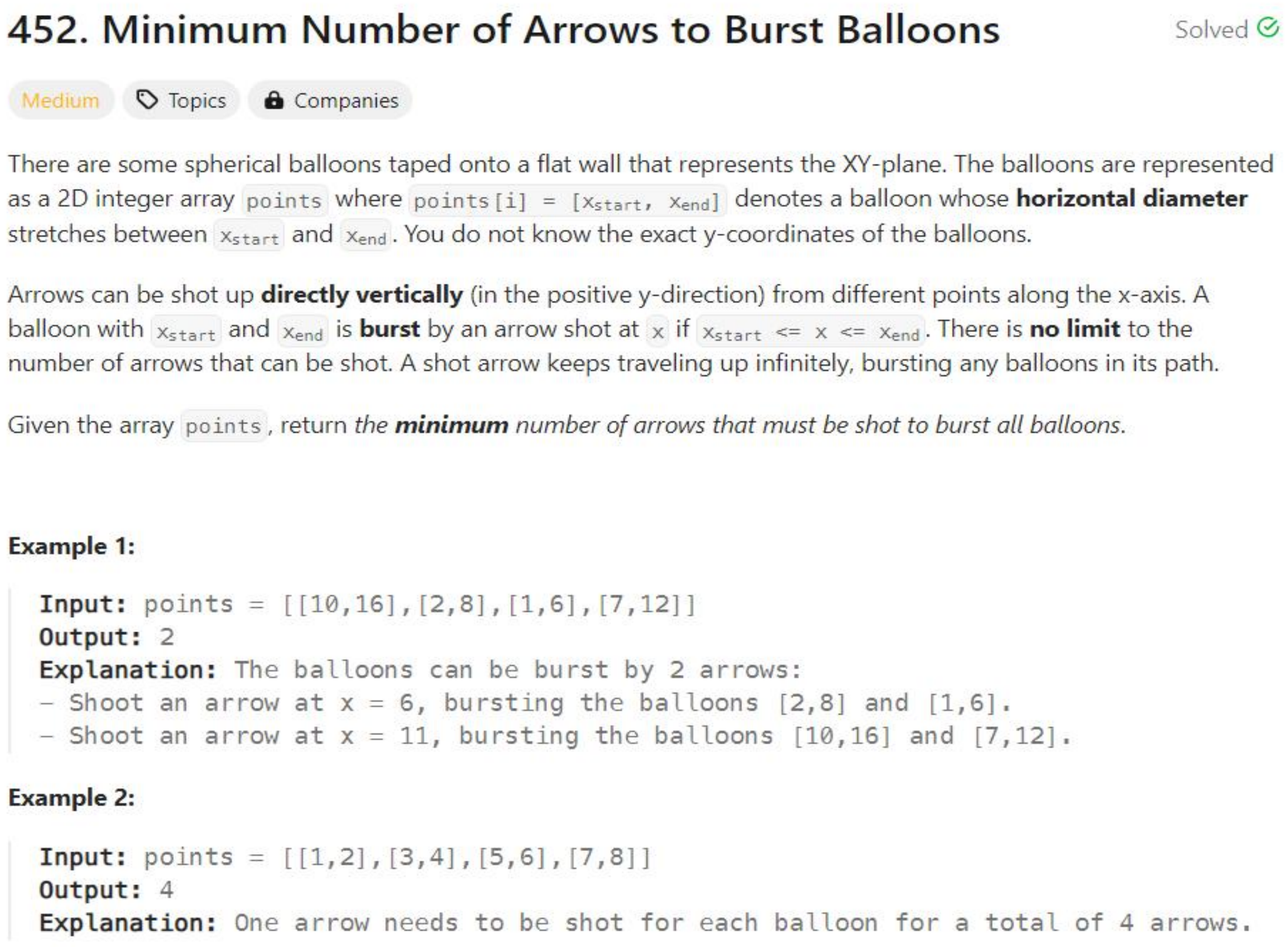
This greedy approach ensures that the solution is both optimal and efficient, with a time complexity dominated by the sorting steps, i.e., O(nlogn+mlogm), where n is the number of children and m is the number of cookies.

Solution:



```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(), g.end());
        sort(s.begin(), s.end());
        int contentChildren = 0;
        int i = 0;
        int j = 0;
                while (i < g.size() && j < s.size()) {
            if (s[j] >= g[i]) {
                contentChildren++;
                i++;
            }
            j++;
        }
        return contentChildren;
    }
};
```

Problem 2:



452. Minimum Number of Arrows to Burst Balloons

Solved ✓

Medium | Topics | Companies

There are some spherical balloons taped onto a flat wall that represents the XY-plane. The balloons are represented as a 2D integer array `points` where `points[i] = [`x_{start}`, `x_{end}`]` denotes a balloon whose **horizontal diameter** stretches between x_{start} and x_{end}. You do not know the exact y-coordinates of the balloons.

Arrows can be shot up **directly vertically** (in the positive y-direction) from different points along the x-axis. A balloon with x_{start} and x_{end} is **burst** by an arrow shot at `x` if x_{start} `<= x <=` x_{end}. There is **no limit** to the number of arrows that can be shot. A shot arrow keeps traveling up infinitely, bursting any balloons in its path.

Given the array `points`, return *the **minimum** number of arrows that must be shot to burst all balloons.*

Example 1:

```
Input: points = [[10,16],[2,8],[1,6],[7,12]]
Output: 2
Explanation: The balloons can be burst by 2 arrows:
- Shoot an arrow at x = 6, bursting the balloons [2,8] and [1,6].
- Shoot an arrow at x = 11, bursting the balloons [10,16] and [7,12].
```

Example 2:

```
Input: points = [[1,2],[3,4],[5,6],[7,8]]
Output: 4
Explanation: One arrow needs to be shot for each balloon for a total of 4 arrows.
```

Approach:

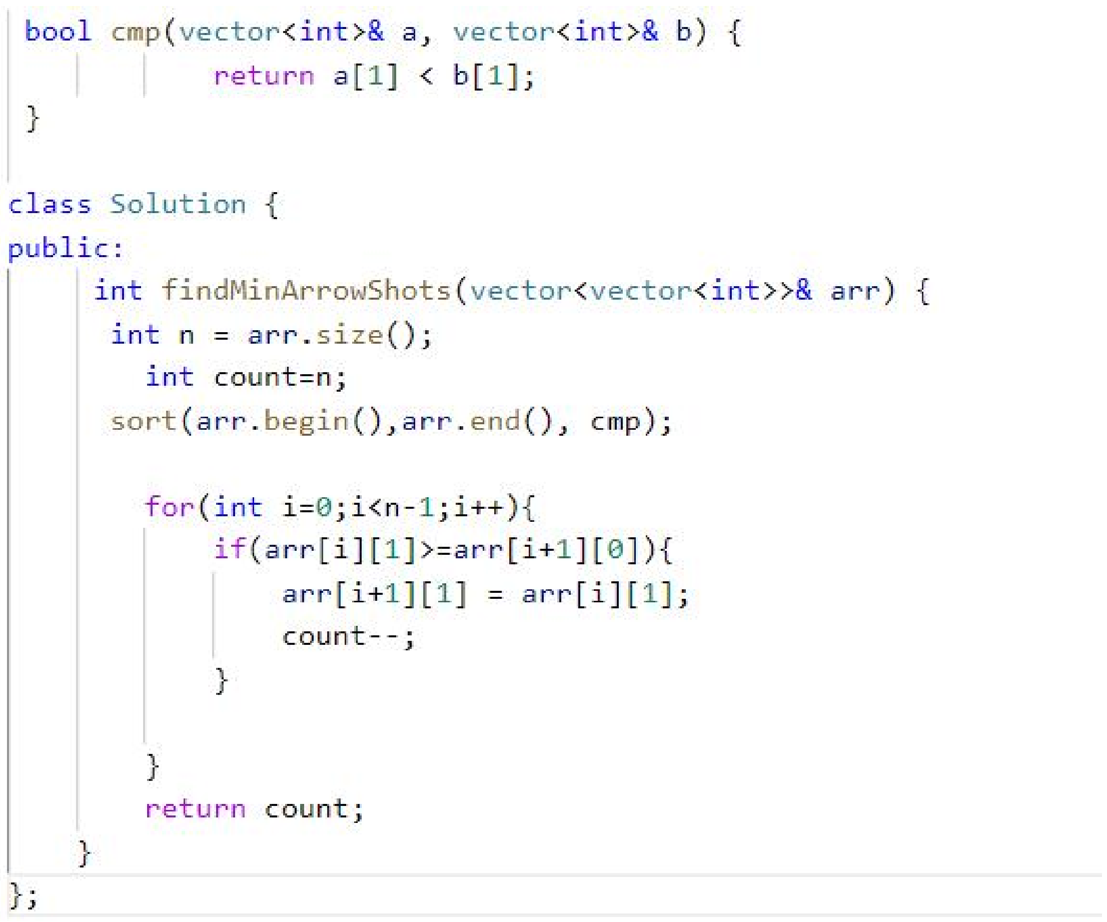
- **Sorting Balloons by End Coordinates:**
    - • The first step is to sort the array of balloons based on their end coordinates (`xend`). This ensures that we are always considering the balloon that ends the earliest, which helps in minimizing the number of arrows used.
- **Greedy Shooting Strategy:**
    - • The strategy is to shoot an arrow at the end coordinate of the current balloon, as this will maximize the number of balloons burst with a single arrow.
    - If the next balloon starts before or at the same point where the previous balloon ends, it means they overlap, and a single arrow can burst both balloons. We don't need an additional arrow for the next balloon.
    - Otherwise, we need a new arrow for the next balloon.
- **Count the Arrows:**
    - • Initialize the count of arrows as the total number of balloons.
    - Iterate through the sorted list of balloons, and for each pair of consecutive balloons, check if they overlap.
    - If they overlap, adjust the end coordinate of the next balloon to be the end coordinate of the current balloon (since one arrow can burst both), and decrement the arrow count.
    - The final arrow count gives the minimum number of arrows required.

Summary of the Approach

1. **Sorting:** By sorting the balloons based on their end coordinates, we ensure that we always deal with the earliest ending balloon first.
2. **Greedy Strategy:** By shooting an arrow at the end coordinate of the current balloon, we maximize the chances of bursting subsequent overlapping balloons with the same arrow.
3. **Overlap Checking:** If the next balloon starts before or at the end coordinate of the current balloon, it means they overlap, and we don't need an additional arrow.

This approach ensures that we minimize the number of arrows used while guaranteeing that all balloons are burst. The time complexity is dominated by the sorting step, which is O(nlogn), making it efficient for the problem constraints.

Solution:



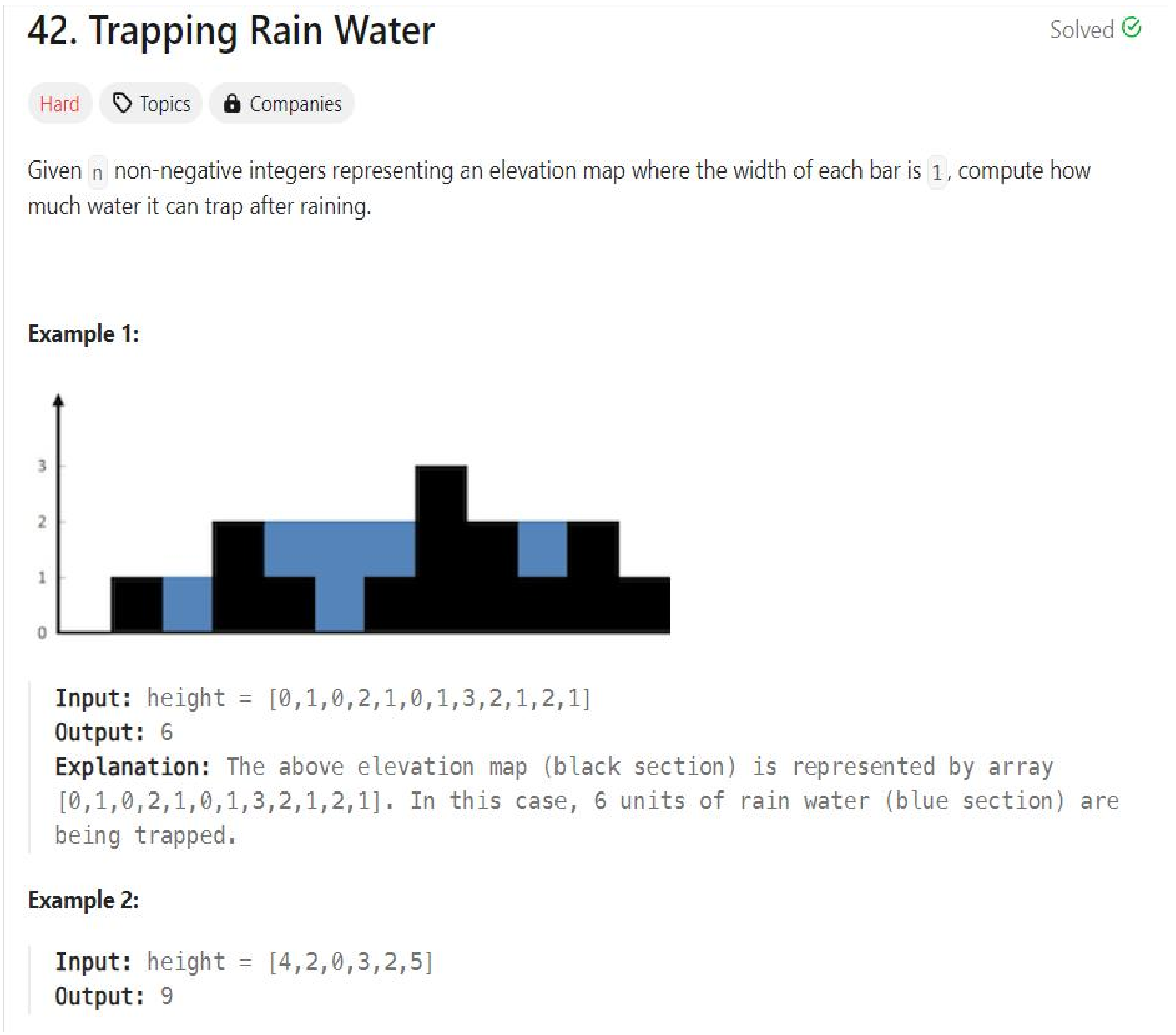
```cpp
bool cmp(vector<int>& a, vector<int>& b) {
            return a[1] < b[1];
}

class Solution {
public:
    int findMinArrowShots(vector<vector<int>>& arr) {
     int n = arr.size();
       int count=n;
     sort(arr.begin(),arr.end(), cmp);

       for(int i=0;i<n-1;i++){
           if(arr[i][1]>=arr[i+1][0]){
               arr[i+1][1] = arr[i][1];
               count--;
           }

       }
       return count;
    }
};
```

Problem 3:



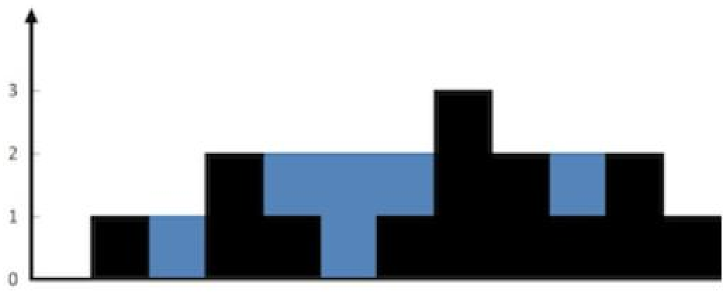
42. Trapping Rain Water

Solved ✓

Hard | Topics | Companies

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Example 1:



```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array
[0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are
being trapped.
```

Example 2:

```
Input: height = [4,2,0,3,2,5]
Output: 9
```

Approach:

Initial Setup:

- The input is a vector `height` representing the elevation map.
- The goal is to compute how much water can be trapped after raining.

- **Create Arrays for Left and Right Maximum Heights:**

  - • Two arrays `l` and `r` are used to store the maximum height to the left and right of each bar, respectively.
  - These arrays help in determining the amount of water that can be trapped above each bar by finding the minimum of the maximum heights from both sides.

- **Fill the Left Maximum Array (`l`):**

- ● Traverse the height array from left to right.
- For each position `i`, `l[i]` is the maximum height from the start to the current position.
- This is calculated as `l[i] = max(l[i-1], h[i])`, where `l[i-1]` is the maximum height up to the previous position and `h[i]` is the height at the current position.

- **Fill the Right Maximum Array (`r`):**

- ● Traverse the height array from right to left.
- For each position `i`, `r[i]` is the maximum height from the end to the current position.
- This is calculated as `r[i] = max(r[i+1], h[i])`, where `r[i+1]` is the maximum height up to the next position and `h[i]` is the height at the current position.

- **Calculate the Trapped Water:**

- ● Traverse the height array again.
- For each position `i`, the water that can be trapped above it is determined by the minimum of the maximum heights from both sides minus the height at that position.
- Accumulate this value for all positions to get the total trapped water.

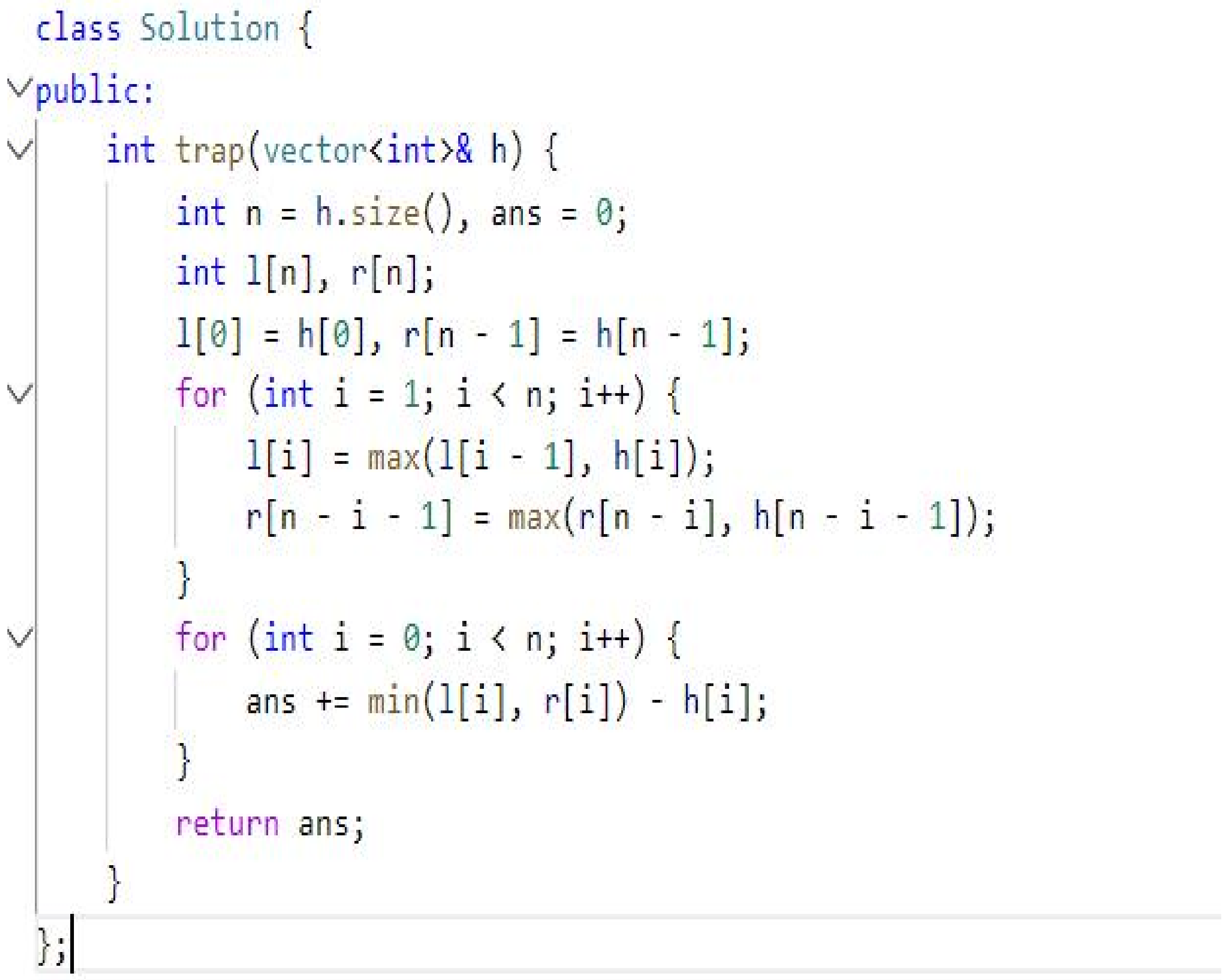
- **Return the Result:**

- ● The accumulated value gives the total amount of water trapped.

Summary of the Approach

1. **Dynamic Programming:** Use two auxiliary arrays to store the maximum heights to the left and right of each position.
2. **Precomputation:** Fill the left and right maximum arrays by traversing the height array from both ends.
3. **Calculation:** Determine the trapped water for each position by finding the minimum of the maximum heights from both sides and subtracting the height at that position.
4. **Efficiency:** This approach runs in O(n)O(n)O(n) time and uses O(n)O(n)O(n) space for the auxiliary arrays, making it efficient and suitable for large input sizes.

Solution:



```cpp
class Solution {
public:
    int trap(vector<int>& h) {
        int n = h.size(), ans = 0;
        int l[n], r[n];
        l[0] = h[0], r[n - 1] = h[n - 1];
        for (int i = 1; i < n; i++) {
            l[i] = max(l[i - 1], h[i]);
            r[n - i - 1] = max(r[n - i], h[n - i - 1]);
        }
        for (int i = 0; i < n; i++) {
            ans += min(l[i], r[i]) - h[i];
        }
        return ans;
    }
};
```