**DSA Questions**

**42. Trapping Rain Water (https://leetcode.com/problems/trapping-rain-water/)**

Hard

21677293Add to ListShare

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

**Constraints:**

* n == height.length
* 1 <= n <= 2 \* 104
* 0 <= height[i] <= 105

Sln

Here's how we can arrive at this solution:

Let's think about the absolute simplest case: we've got a [2,1,3] array, telling us that we can trap 1 block of rainwater.

How we arrive to this, is pretty simple, we know that because we've got a two at the beginning, we can only fill up to two blocks of water per point, and we know that we can only do that at a point after two, and we know that we can do it at all because 3, at the end of the array, would be able to contain the water, so we can add water until we get to 3, and can only add 2 - the height of the point.

So, if we had something a little more complex, like [2, 1, 3, 1, 4], we could fill up to the 3 optimally, and then repear the same algorithm from the 3 onward. However, if we had, instead, [2, 1, 3, 1, 2] we would fill up to the 3, and then see that we cannot fill over to the 2 because we would overflow, so we instead mirror the algorithm and bring from the 2 backward.

class Solution {

public int trap(int[] height) {

if (height == null || height.length == 0) {

return 0;

}

int left = 0; int right = height.length - 1; // Pointers to both ends of the array.

int maxLeft = 0; int maxRight = 0;

int totalWater = 0;

while (left < right) {

// Water could, potentially, fill everything from left to right, if there is nothing in between.

if (height[left] < height[right]) {

// If the current elevation is greater than the previous maximum, water cannot occupy that point at all.

// However, we do know that everything from maxLeft to the current index, has been optimally filled, as we've

// been adding water to the brim of the last maxLeft.

if (height[left] >= maxLeft) {

// So, we say we've found a new maximum, and look to see how much water we can fill from this point on.

maxLeft = height[left];

// If we've yet to find a maximum, we know that we can fill the current point with water up to the previous

// maximum, as any more will overflow it. We also subtract the current height, as that is the elevation the

// ground will be at.

} else {

totalWater += maxLeft - height[left];

}

// Increment left, we'll now look at the next point.

left++;

// If the height at the left is NOT greater than height at the right, we cannot fill from left to right without over-

// flowing; however, we do know that we could potentially fill from right to left, if there is nothing in between.

} else {

// Similarly to above, we see that we've found a height greater than the max, and cannot fill it whatsoever, but

// everything before is optimally filled

if (height[right] >= maxRight) {

// We can say we've found a new maximum and move on.

maxRight = height[right];

// If we haven't found a greater elevation, we can fill the current elevation with maxRight - height[right]

// water.

} else {

totalWater += maxRight - height[right];

}

// Decrement left, we'll look at the next point.

right--;

}

}

// Return the sum we've been adding to.

return totalWater;

}

}