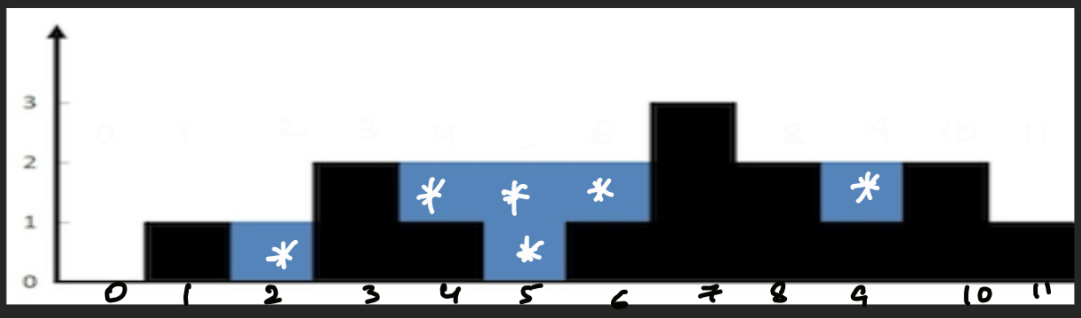
Leetcode Trapping Rain Water (Hard)
https://leetcode.com/problems/trapping-rain-water/description/

42. Trapping Rain Water

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
```

Constraints:

- `n == height.length`
- `1 <= n <= 2 * 10^4`
- `0 <= height[i] <= 10^5`

APPROACH 1 :-

GROUND RULE :- Water can be trapped between two high points. — leftMax & rightMax

In the given figure, water can be trapped at points (2, 4, 5, 6, 9) where the heights that trap water are at points

{(1,3), (3,7), (3,7), (3,7), (8,10)}

This is step ① - to figure the heights that trap water

Step ②: calculate water that can be trapped (in units)

Let's calculate water at point 2. The boundaries that trap water at 2 are (1,3)

ant of water at point 2 = min(1,3) − height at point 2

= 1 − 0

= 1 unit.

In the same way, total water trapped =

$$[\min(1,3) - h[2]] + [\min(3,7) - h[4]] + [\min(3,7) - h[5]] +$$

$$[\min(3,7) - h[6]] + [\min(8,10) - h[9]]$$

$$= (1-0) + (3-1) + (3-0) + (3-1) + (8-1)$$

$$= 1+2+3+2+7$$

$$= 15$$

Step ①:- FIGURE HEIGHTS THAT CAN TRAP WATER AT EACH POINT.

Let's denote left boundary as leftMax
right boundary as rightMax.

Two Arrays : leftMax[] rightMax[]

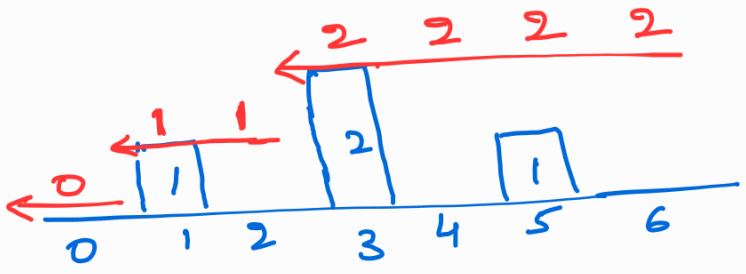
leftMax[i], rightMax[i] are left & right boundaries that trap water at i.

Determining leftMax[i]:

If height at point i is greater than leftMax of (i-1), then leftMax[i] = height at point i
= height[i].

$\Rightarrow$ leftMax[i] = max(leftMax[i-1], height[i])

Eg:-



At points 3, 4, 5, 6; the max height on left is 2.
At point 1, 2; the max height on left is 1.
At point 0; the max height on left is 0.

Hence,

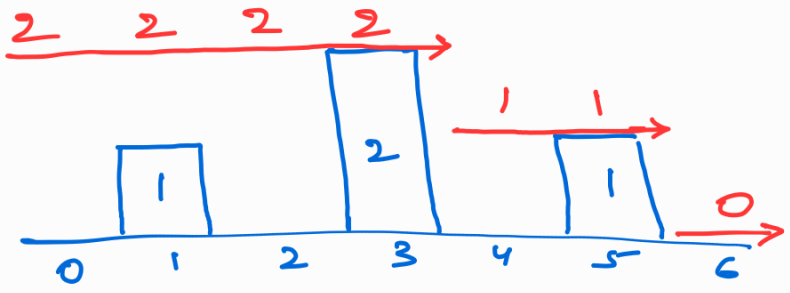
leftMax[] = { 0, 1, 1, 2, 2, 2, 2 }.

Determining rightMax[i]:

If height at point i is greater than rightMax of (i+1), then rightMax[i] = height at point i
= height[i].

$\Rightarrow$ rightMax[i] = max(rightMax[i+1], height[i]);

Eg:-

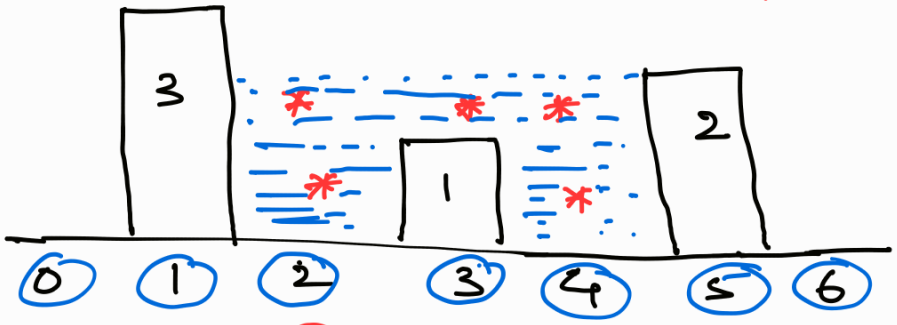


At points 0, 1, 2, 3; max height on right is 2
At points 4, 5; max height on right is 1.
At point 6; max height on right is 0.
Hence, rightMax[] = {2, 2, 2, 2, 1, 1, 0}.

Step ②: CALCULATE AMOUNT OF WATER THAT IS TRAPPED AT EACH POINT BASED ON LEFTMAX & RIGHTMAX.

Eg:- Let's calculate water trapped at 4.

leftMax:-	0	3	3	3	3	3	2
rightMax:-	3	3	2	2	2	2	0



0 + 0 + 2 + 1 + 2 + 0 + 0

min(leftMax[2], rightMax[2]) - h[2]

= min(3, 2) - 0

= 2 - 0

= 2

DP FORMULA:-

{water trapped at i} = min{leftMax[i], rightMax[i]} - h[i]

for i: 0 to 6, totalWater = totalWater + water trapped at i

```
public int trapWater(int[] height) {
      int n = height.length;

       if (n <= 2) return 0;

        int[] leftMax = new int[n];
        int[] rightMax = new int[n];

        leftMax[0] = height[0];
        rightMax[n - 1] = height[n - 1];

        for (int i = 1; i < n; i++) {
               int j = n - i - 1;
                leftMax[i] = Math.max(leftMax[i - 1], height[i]);
                rightMax[j] = Math.max(rightMax[j + 1], height[j]);
         }

         int totalWater = 0;

         for (int k = 0; k < n; k++) {
               int water = Math.min(leftMax[k], rightMax[k]);
               totalWater += (water - height[k]);
          }
          return totalWater;
}
```

Time Complexity : O(n)

Space Complexity : O(n)

APPROACH 2 :-

GROUNDRULE :-

If there are two points leftMax & rightMax, water can be trapped atmost at a height of

(1) leftMax at left++ if leftMax < rightMax

(2) rightMax at right-- if rightMax < leftMax.

Determine leftMax & rightMax with two pointers left and right such that if height at position left h[left] is greater than leftMax, then update leftMax as h[left]

(3) if h[left] > leftMax, then leftMax = h[left]

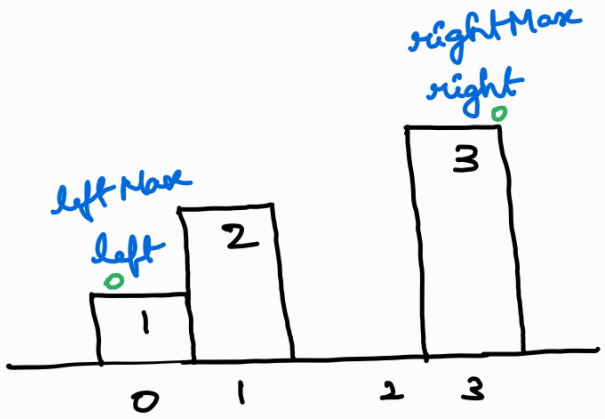
(4) if h[right] > rightMax, then rightMax = h[right]

NOTE :- left, right ⟶ positions.

leftMax, rightMax ⟶ heights

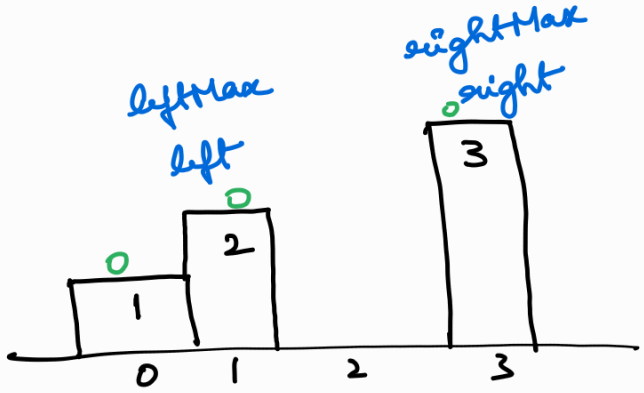
EXAMPLE :-

(1)



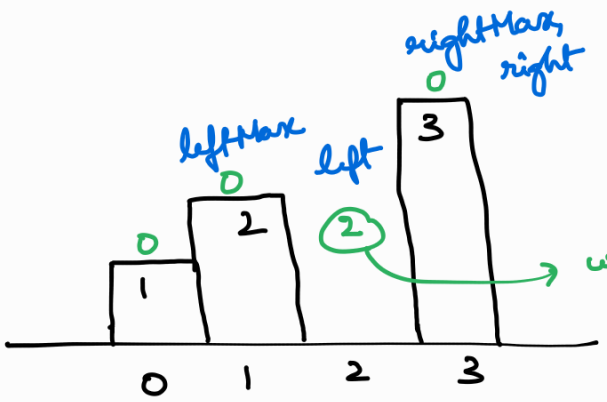
applying conditions (3), (4)

(2)



applying conditions (1), (3)

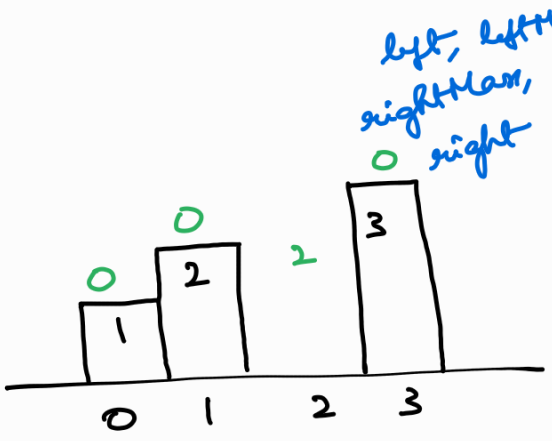
(3)



applying condition (1)

water trapped = leftMax − h[2]
= 2 − 0 = 2.

(4)

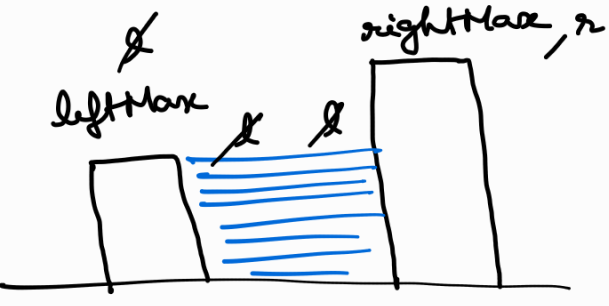


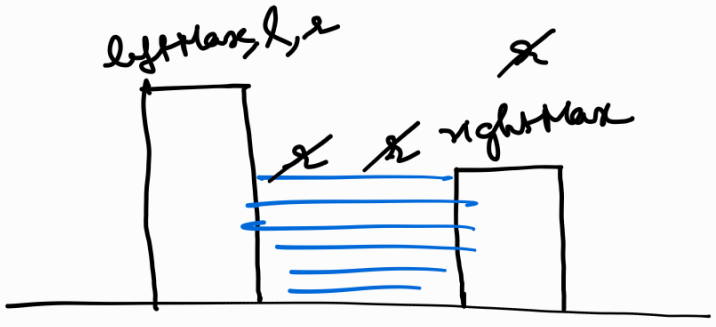
applying condition (1), (3)

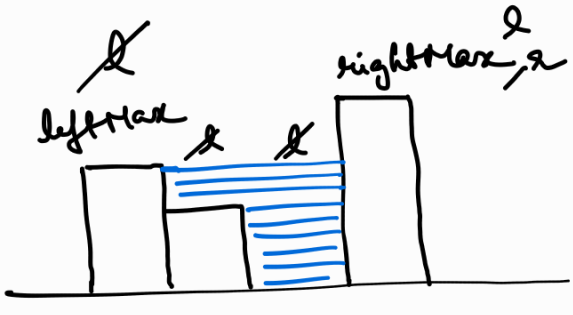
when left = right, stop process & return totalWater.

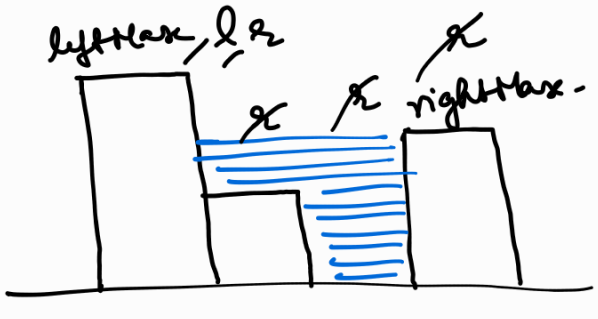
totalWater = 0 + 0 + 2 + 0 = 2.

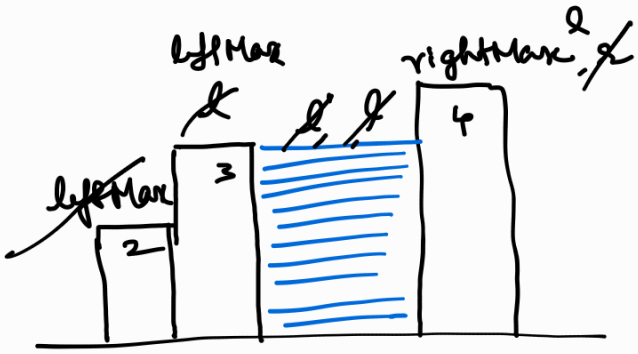
* Similarly, more such different examples are below:

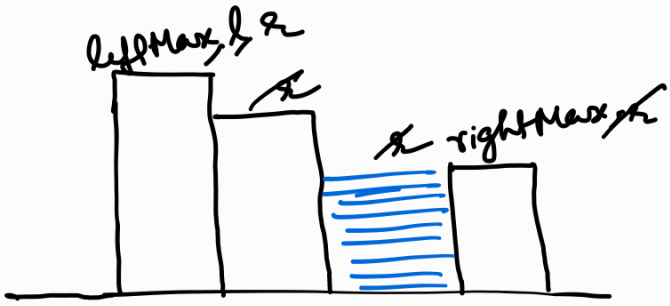


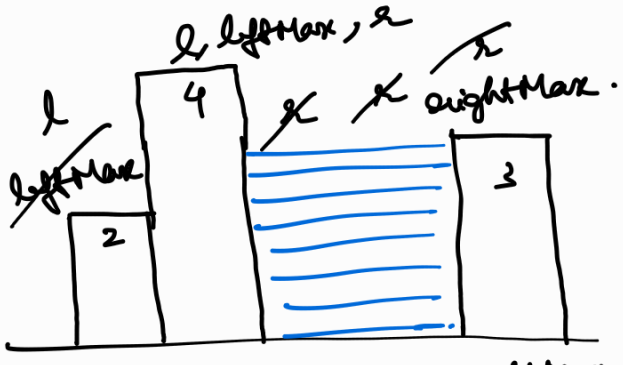


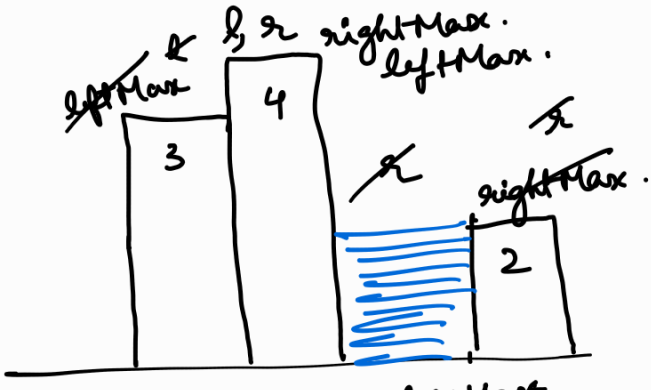


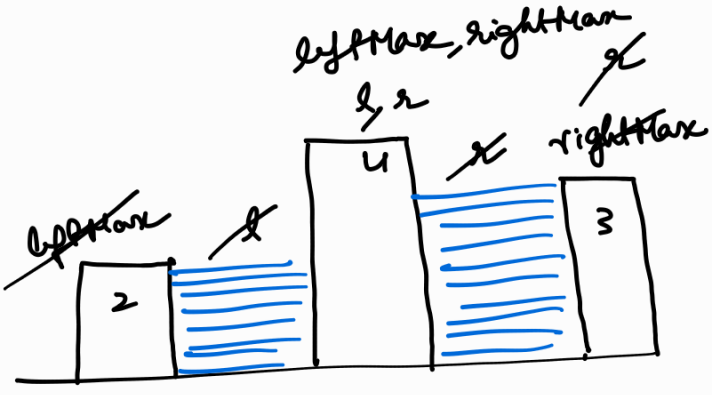


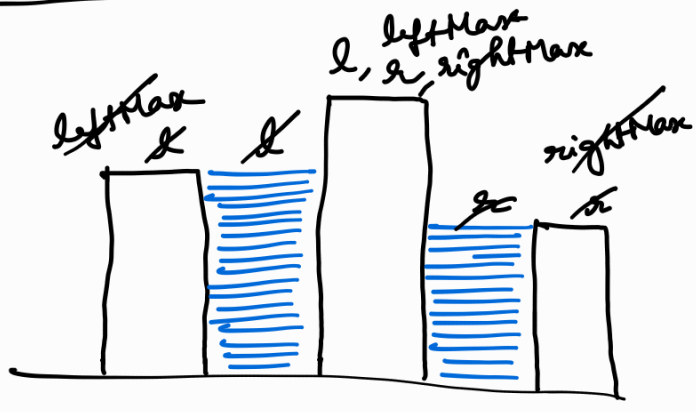












Algorithm:-

① If leftMax < rightMax

fill water at next position, l++

if h[l] > leftMax, then leftMax = h[l]

totalWater = totalWater + {leftMax - h[l]}

② If rightMax < leftMax.

fill water at next position, r--

if h[r] > rightMax, then rightMax = h[r]

totalWater = totalWater + {rightMax - h[r]}.

```
public int trapWater(int[] height) {
        if (height == null || height.length == 0) {
            return 0;
        }
        int left = 0, right = height.length - 1;
        int maxLeft = height[left], maxRight = height[right];
        int totalWater = 0;

        while (left < right) {
            if (maxLeft < maxRight) {
                left++;
                maxLeft = Math.max(maxLeft, height[left]);
                totalWater += (maxLeft - height[left]);
            } else {
                right--;
                maxRight = Math.max(maxRight, height[right]);
                totalWater += (maxRight - height[right]);
            }
        }
        return totalWater;
}
```

Time Complexity: O(n)

Space Complexity: O(1)