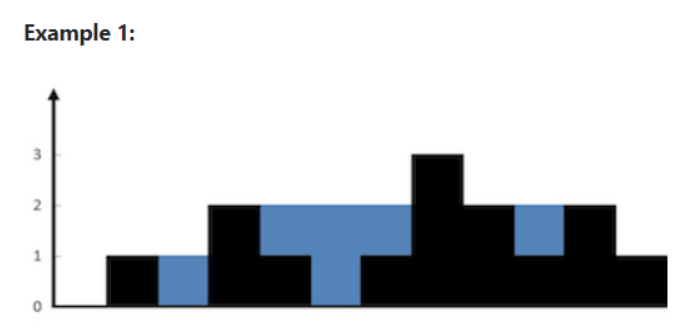
<https://leetcode.com/problems/trapping-rain-water/>

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

**Attempt 1: 2023-02-24**

**Solution 1: Two Pointers (30 min)**

class Solution {

public int trap(int[] height) {

int sum = 0;

int left = 0;

int right = height.length - 1;

int left\_max = height[left];

int right\_max = height[right];

while(left < right) {

if(left\_max < right\_max) {

left++;

if(left\_max > height[left]) {

sum += left\_max - height[left];

} else {

left\_max = height[left];

}

} else {

right--;

if(right\_max > height[right]) {

sum += right\_max - height[right];

} else {

right\_max = height[right];

}

}

}

return sum;

}

}

Time complexity: O(n). Single iteration of O(n).

Space complexity: O(1) extra space. Only constant space required for left, right, left\_max and right\_max.

**Refer to**

<http://www.cnblogs.com/grandyang/p/4402392.html>

最后我们来看一种只需要遍历一次即可的解法，这个算法需要left和right两个指针分别指向数组的首尾位置，从两边向中间扫描，在当前两指针确定的范围内，先比较两头找出较小值，如果较小值是left指向的值，则从左向右扫描，如果较小值是right指向的值，则从右向左扫描，若遇到的值比当较小值小，则将差值存入结果，如遇到的值大，则重新确定新的窗口范围，以此类推直至left和right指针重合

**Refer to**

<https://leetcode.com/problems/trapping-rain-water/editorial/>

#### **Approach 4: Using 2 pointers**

**Intuition**

As in [Approach 2](https://leetcode.com/problems/trapping-rain-water/editorial/" \l "approach-2-dynamic-programming), instead of computing the left and right parts separately, we may think of some way to do it in one iteration. From the figure in dynamic programming approach, notice that as long as right\_max[i]>left\_max[i] (from element 0 to 6), the water trapped depends upon the left\_max, and similar is the case when left\_max[i]>right\_max[i](from element 8 to 11). So, we can say that if there is a larger bar at one end (say right), we are assured that the water trapped would be dependent on height of bar in current direction (from left to right). As soon as we find the bar at other end (right) is smaller, we start iterating in opposite direction (from right to left). We must maintain left\_max and right\_max during the iteration, but now we can do it in one iteration using 2 pointers, switching between the two.

**Algorithm**

Initialize left pointer to 0 and right pointer to size-1

While left<right, do:

If height[left] is smaller than height[right]

If height[left]≥left\_max, update left\_max

Else add left\_max−height[left] to ans

Add 1 to left.

Else

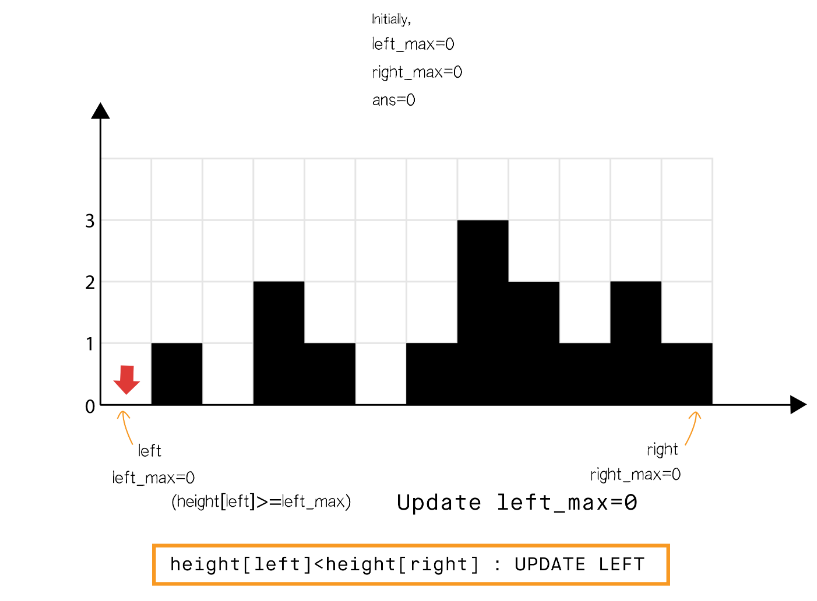
If height[right]≥right\_max, update right\_max

Else add right\_max−height[right] to ans

Subtract 1 from right.

Refer the example for better understanding:

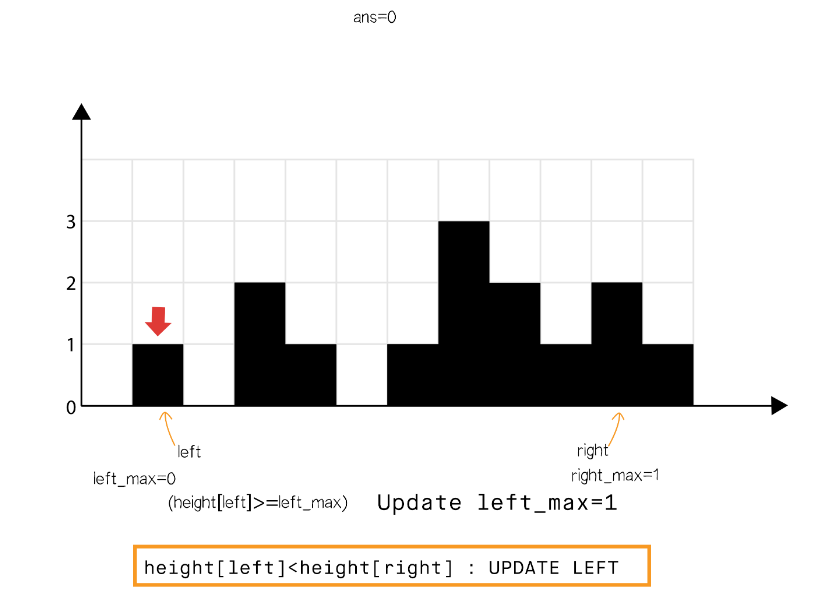
Step 1



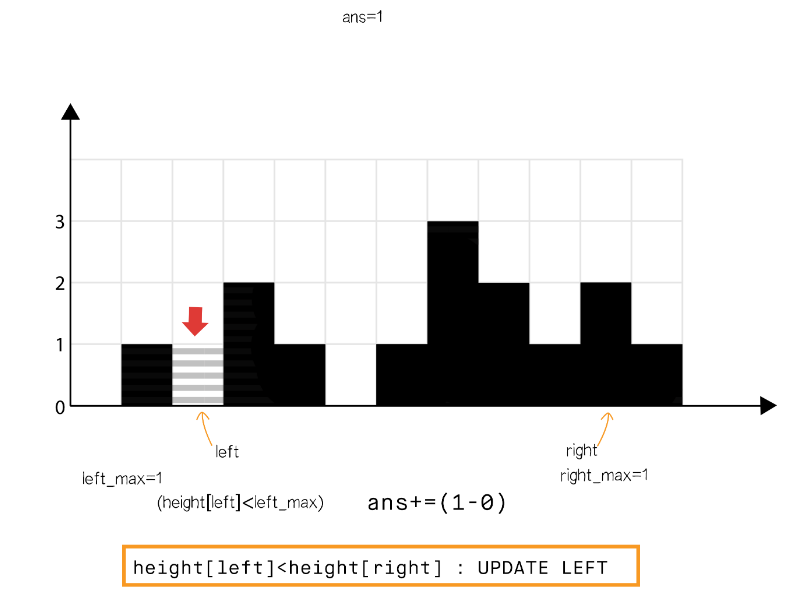
Step 2



Step 3



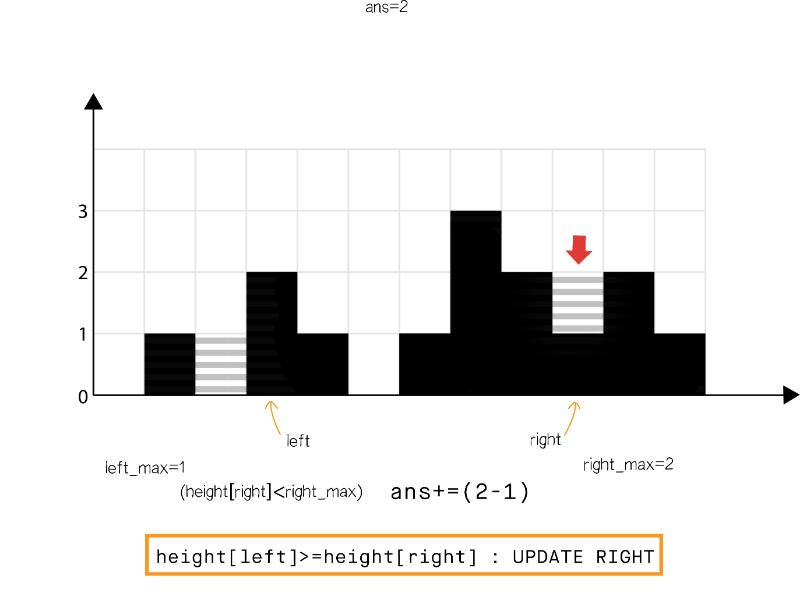
Step 4



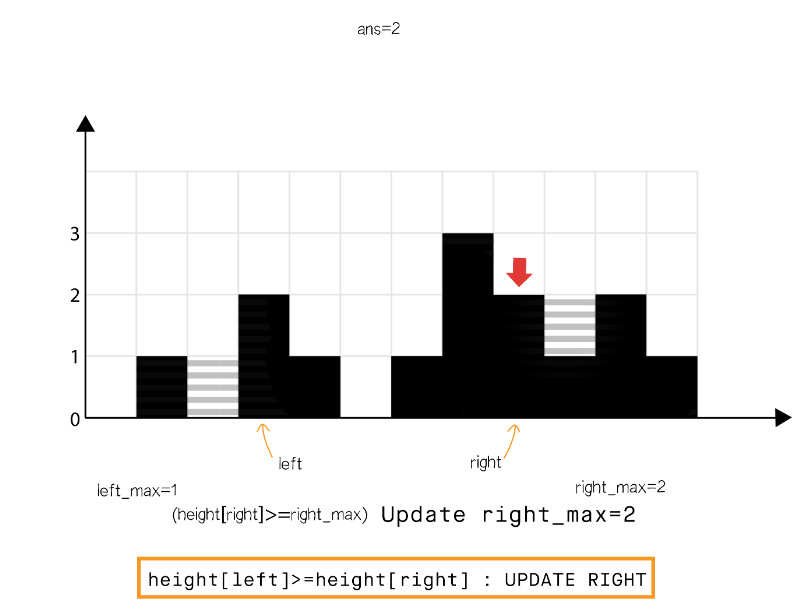
Step 5



Step 6



Step 7



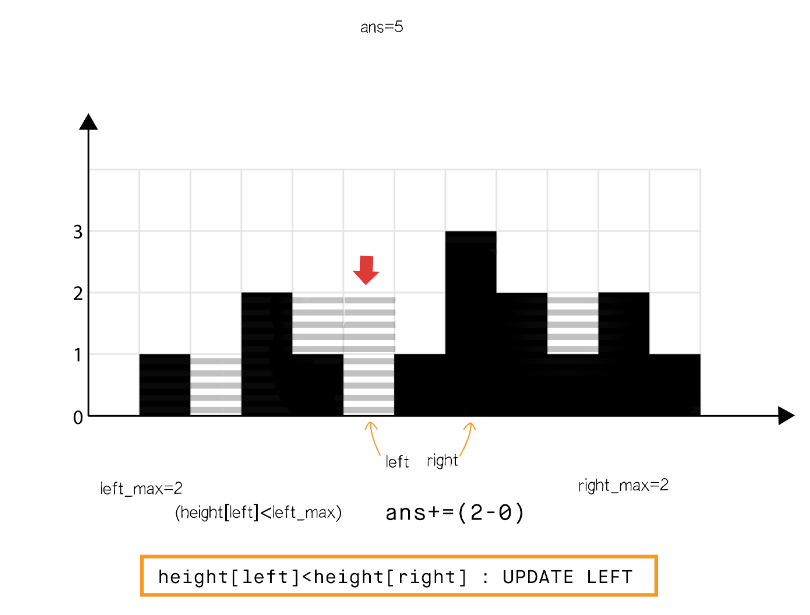
Step 8



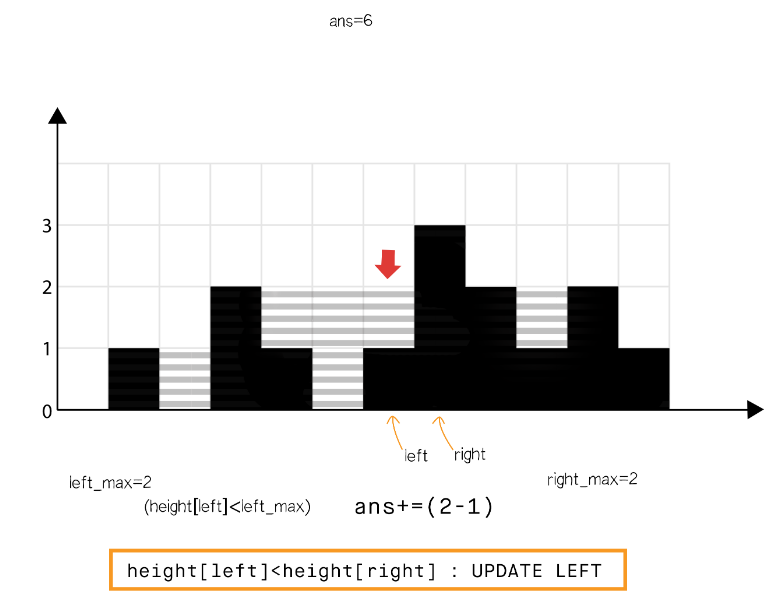
Step 9



Step 10



Step 11



class Solution {

public:

int trap(vector<int>& height)

{

int left = 0, right = height.size() - 1;

int ans = 0;

int left\_max = 0, right\_max = 0;

while (left < right) {

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

height[left] >= left\_max ? (left\_max = height[left]) : ans += (left\_max - height[left]);

++left;

}

else {

height[right] >= right\_max ? (right\_max = height[right]) : ans += (right\_max - height[right]);

--right;

}

}

return ans;

}

};

**Complexity analysis**

* Time complexity: O(n). Single iteration of O(n).
* Space complexity: O(1) extra space. Only constant space required for left, right, left\_max and right\_max.

**Solution 2: Brute Force (30 min)**

class Solution {

public int trap(int[] height) {

int sum = 0;

for(int i = 0; i < height.length; i++) {

int left\_max = 0;

int right\_max = 0;

for(int j = i; j >= 0; j--) {

left\_max = Math.max(left\_max, height[j]);

}

for(int j = i; j < height.length; j++) {

right\_max = Math.max(right\_max, height[j]);

}

sum += Math.min(left\_max, right\_max) - height[i];

}

return sum;

}

}

**Refer to**

<https://leetcode.com/problems/trapping-rain-water/editorial/>

#### **Approach 1: Brute force**

**Intuition**

Do as directed in question. For each element in the array, we find the maximum level of water it can trap after the rain, which is equal to the minimum of maximum height of bars on both the sides minus its own height.

**Algorithm**

Initialize ans=0

Iterate the array from left to right:

Initialize left\_max=0 and right\_max=0

Iterate from the current element to the beginning of array updating:

left\_max=max⁡(left\_max,height[j])

Iterate from the current element to the end of array updating:

right\_max=max⁡(right\_max,height[j])

Add min⁡(left\_max,right\_max)−height[i] to ans

class Solution {

public:

int trap(vector<int>& height)

{

int ans = 0;

int size = height.size();

for (int i = 1; i < size - 1; i++) {

int left\_max = 0, right\_max = 0;

for (int j = i; j >= 0; j--) { //Search the left part for max bar size

left\_max = max(left\_max, height[j]);

}

for (int j = i; j < size; j++) { //Search the right part for max bar size

right\_max = max(right\_max, height[j]);

}

ans += min(left\_max, right\_max) - height[i];

}

return ans;

}

};

**Solution 3: DP (30 min)**

class Solution {

public int trap(int[] height) {

int sum = 0;

int len = height.length;

int[] left\_max = new int[len];

int[] right\_max = new int[len];

left\_max[0] = height[0];

right\_max[len - 1] = height[len - 1];

for(int i = 1; i < len; i++) {

left\_max[i] = Math.max(left\_max[i - 1], height[i]);

}

for(int i = len - 2; i >= 0; i--) {

right\_max[i] = Math.max(right\_max[i + 1], height[i]);

}

for(int i = 1; i < len - 1; i++) {

sum += Math.min(left\_max[i], right\_max[i]) - height[i];

}

return sum;

}

}

Time complexity: O(n)

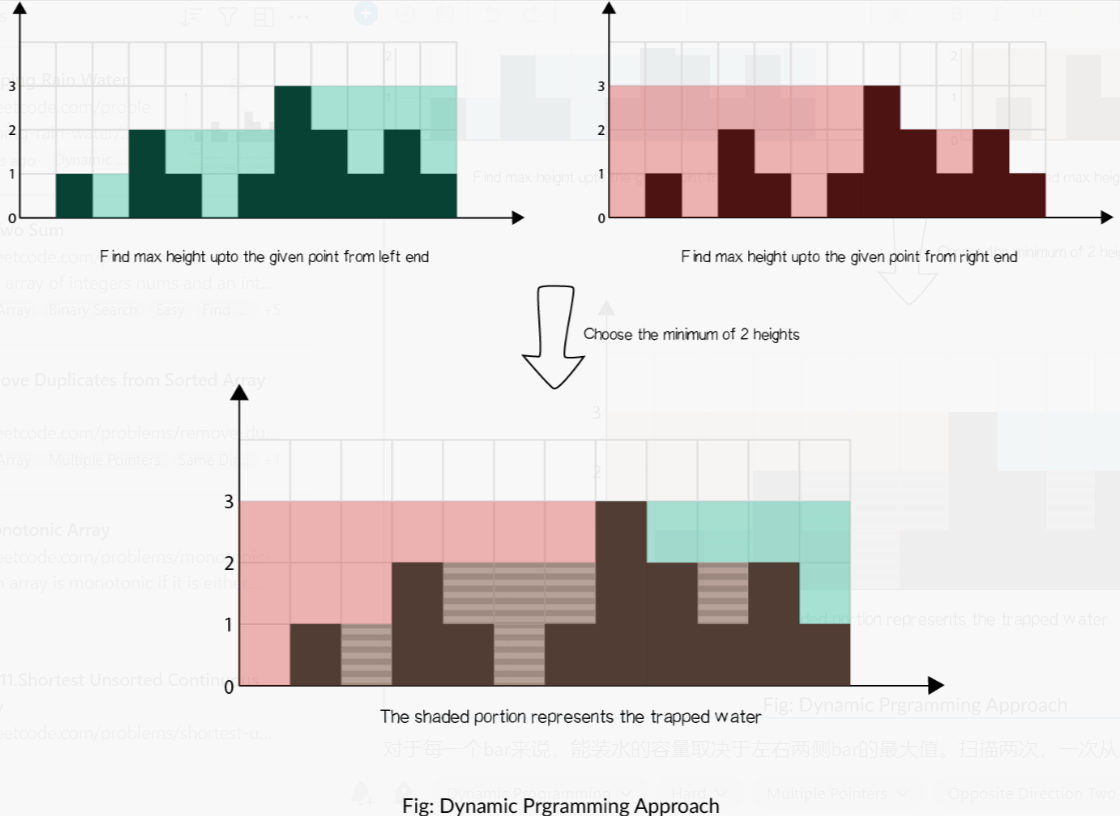
Space complexity: O(n)

**Refer to**

<https://leetcode.com/problems/trapping-rain-water/solution/>

<https://aaronice.gitbook.io/lintcode/two_pointers/trapping_rain_water>

Dynamic Programming



对于每一个bar来说，能装水的容量取决于左右两侧bar的最大值。扫描两次，一次从左向右，记录对于每一个bar来说其左侧的bar的最大高度left[i]，一次从右向左，记录每一个bar右侧bar的最大高度right[i]。第三次扫描，则对于每一个bar，计算

（1）左侧最大height和当前bar的height的差值（left[i] - heights[i]）

（2）右侧最大height和当前bar的height的差值（right[i] - heights[i]），

取（1）和（2）中结果小的那个作为当前bar的蓄水量。最终求和得到总蓄水量。或者直接找到bari两侧最大height中最小的那个，再找与heights[i]的大于零的差值即可。

The water each bar can trap depends on the maximum height on its left and right. Thus scan twice - from left to right,

and right to left and record the max height in each direction. The third time calculate the min difference between

left/right height and current bar height. Sum the trapped water to get the final result.

简而言之：找左边最大和右边最大可以通过两个dp数组来存起来.

dpLeft[i] = Math.max(dpLeft[i - 1], height[i]);

dpRight[i] = Math.max(dpRight[i + 1], height[i]);

class Solution {

public int trap(int[] height) {

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

return 0;

}

int size = height.length;

int[] left\_max = new int[size];

int[] right\_max = new int[size];

left\_max[0] = height[0];

right\_max[size - 1] = height[size - 1];

int sum = 0;

for(int i = 1; i < size; i++) {

left\_max[i] = Math.max(left\_max[i - 1], height[i]);

}

for(int i = size - 2; i >= 0; i--) {

right\_max[i] = Math.max(right\_max[i + 1], height[i]);

}

for(int i = 1; i < size - 1; i++) {

sum += Math.min(left\_max[i], right\_max[i]) - height[i];

}

return sum;

}

}

Time: O(N), Space: O(2\*N)