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

/\*\*

\* @param {number[]} height

\* @return {number}

\*/

var trap = function(height) {

};

Solution

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*ans*=0
* Iterate the array from left to right:
  + Initialize \text{left\\_max}=0left\_max=0 and \text{right\\_max}=0right\_max=0
  + Iterate from the current element to the beginning of array updating:
    - \text{left\\_max}=\max(\text{left\\_max},\text{height}[j])left\_max=max(left\_max,height[*j*])
  + Iterate from the current element to the end of array updating:
    - \text{right\\_max}=\max(\text{right\\_max},\text{height}[j])right\_max=max(right\_max,height[*j*])
  + Add \min(\text{left\\_max},\text{right\\_max}) - \text{height}[i]min(left\_max,right\_max)−height[*i*] to \text{ans}ans

**Complexity Analysis**

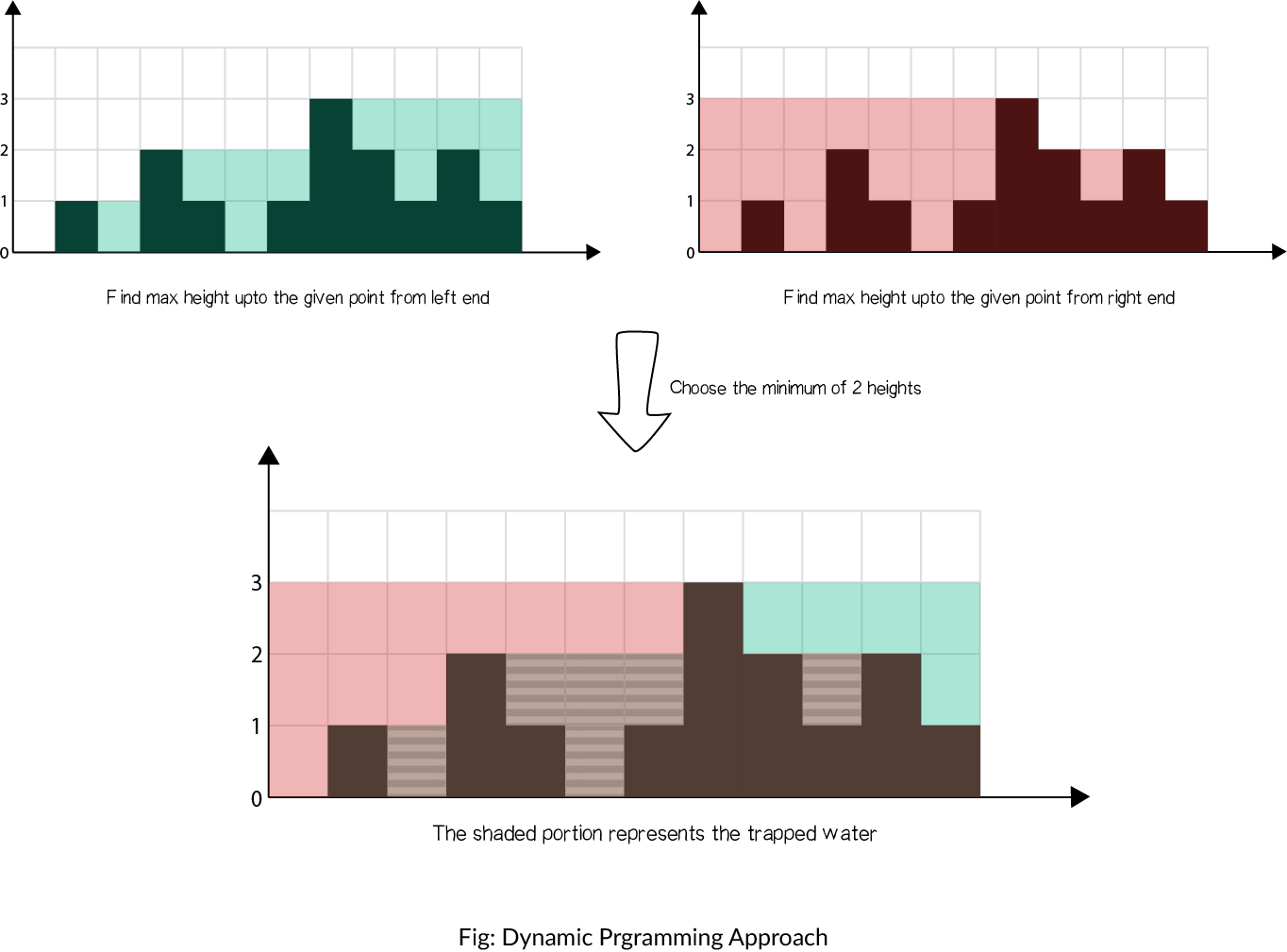
* Time complexity: O(n^2)*O*(*n*2). For each element of array, we iterate the left and right parts.
* Space complexity: O(1)*O*(1) extra space.

Approach 2: Dynamic Programming

**Intuition**

In brute force, we iterate over the left and right parts again and again just to find the highest bar size upto that index. But, this could be stored. Voila, dynamic programming.

The concept is illustrated as shown:



**Algorithm**

* Find maximum height of bar from the left end upto an index i in the array \text{left\\_max}left\_max.
* Find maximum height of bar from the right end upto an index i in the array \text{right\\_max}right\_max.
* Iterate over the \text{height}height array and update ans:
  + Add \min(\text{left\\_max}[i],\text{right\\_max}[i]) - \text{height}[i]min(left\_max[*i*],right\_max[*i*])−height[*i*] to \text{ans}ans

int trap(vector<int>& height)

{

if(height.empty())

return 0;

int ans = 0;

int size = height.size();

vector<int> left\_max(size), right\_max(size);

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

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

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

}

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

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

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

}

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

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

}

return ans;

}

**Complexity analysis**

* Time complexity: O(n)*O*(*n*).
  + We store the maximum heights upto a point using 2 iterations of O(n)*O*(*n*) each.
  + We finally update \text{ans}ans using the stored values in O(n)*O*(*n*).
* Space complexity: O(n)*O*(*n*) extra space.
  + Additional O(n)*O*(*n*) space for \text{left\\_max}left\_max and \text{right\\_max}right\_max arrays than in [Approach 1](https://leetcode.com/problems/trapping-rain-water/solution/#approach-1-brute-force).

Approach 3: Using stacks

**Intuition**

Instead of storing the largest bar upto an index as in [Approach 2](https://leetcode.com/problems/trapping-rain-water/solution/#approach-2-dynamic-programming), we can use stack to keep track of the bars that are bounded by longer bars and hence, may store water. Using the stack, we can do the calculations in only one iteration.

We keep a stack and iterate over the array. We add the index of the bar to the stack if bar is smaller than or equal to the bar at top of stack, which means that the current bar is bounded by the previous bar in the stack. If we found a bar longer than that at the top, we are sure that the bar at the top of the stack is bounded by the current bar and a previous bar in the stack, hence, we can pop it and add resulting trapped water to \text{ans}ans.

**Algorithm**

* Use stack to store the indices of the bars.
* Iterate the array:
  + While stack is not empty and \text{height[current]}>\text{height[st.top()]}height[current]>height[st.top()]
    - It means that the stack element can be popped. Pop the top element as \text{top}top.
    - Find the distance between the current element and the element at top of stack, which is to be filled. \text{distance} = \text{current} - \text{st.top}() - 1distance=current−st.top()−1
    - Find the bounded height \text{bounded\\_height} = \min(\text{height[current]}, \text{height[st.top()]}) - \text{height[top]}bounded\_height=min(height[current],height[st.top()])−height[top]
    - Add resulting trapped water to answer \text{ans} \mathrel{+}= \text{distance} \times \text{bounded\\_height}ans+=distance×bounded\_height
  + Push current index to top of the stack
  + Move \text{current}current to the next position

int trap(vector<int>& height)

{

int ans = 0, current = 0;

stack<int> st;

while (current < height.size()) {

while (!st.empty() && height[current] > height[st.top()]) {

int top = st.top();

st.pop();

if (st.empty())

break;

int distance = current - st.top() - 1;

int bounded\_height = min(height[current], height[st.top()]) - height[top];

ans += distance \* bounded\_height;

}

st.push(current++);

}

return ans;

}

Approach 4: Using 2 pointers

**Intuition**

As in [Approach 2](https://leetcode.com/problems/trapping-rain-water/solution/#approach-2-dynamic-programming), instead of computing the left and right parts seperately, we may think of some way to do it in one iteration. From the figure in dynamic programming approach, notice that as long as \text{right\\_max}[i]>\text{left\\_max}[i]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 \text{left\\_max}[i]>\text{right\\_max}[i]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 dependant 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 \text{left\\_max}left\_max and \text{right\\_max}right\_max during the iteration, but now we can do it in one iteration using 2 pointers, switching between the two.

**Algorithm**

* Initialize \text{left}left pointer to 0 and \text{right}right pointer to size-1
* While \text{left}< \text{right}left<right, do:
  + If \text{height[left]}height[left] is smaller than \text{height[right]}height[right]
    - If \text{height[left]} \geq \text{left\\_max}height[left]≥left\_max, update \text{left\\_max}left\_max
    - Else add \text{left\\_max}-\text{height[left]}left\_max−height[left] to \text{ans}ans
    - Add 1 to \text{left}left.
  + Else
    - If \text{height[right]} \geq \text{right\\_max}height[right]≥right\_max, update \text{right\\_max}right\_max
    - Else add \text{right\\_max}-\text{height[right]}right\_max−height[right] to \text{ans}ans
    - Subtract 1 from \text{right}right.

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