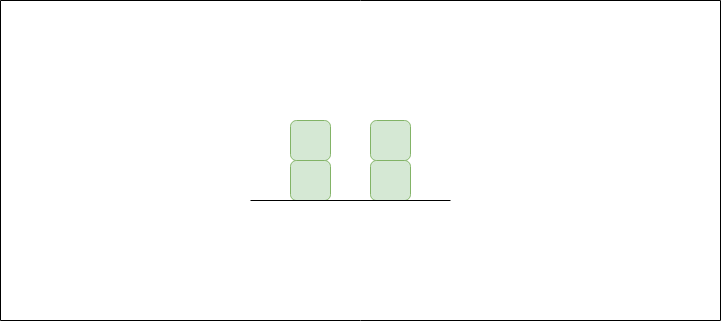
***Trapping Rain Water***

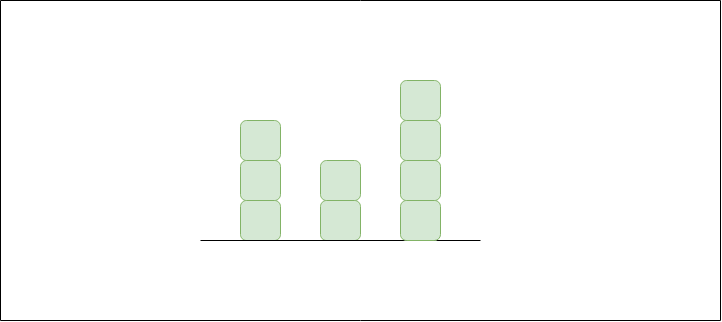
Given an array of **N** non-negative integers **arr[]** representing an elevation map where the width of each bar is **1**, compute how much water it is able to trap after raining.

**Examples**:

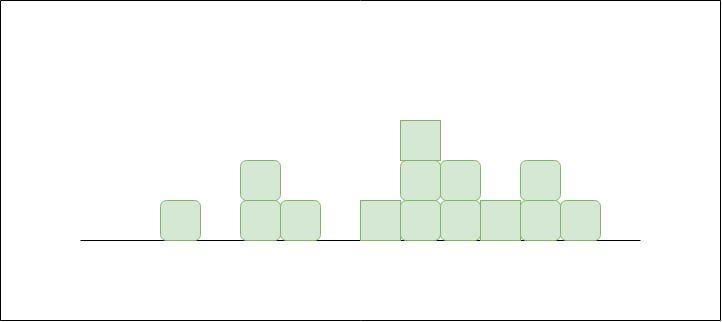
***Input:****arr[] = {2, 0, 2}*  
***Output:****2*  
***Explanation:****The structure is like below.*  
*We can trap 2 units of water in the middle gap.*

**

***Input:****arr[]   = {3, 0, 2, 0, 4}*  
***Output:****7*  
***Explanation:****Structure is like below.*  
*We can trap “3 units” of water between 3 and 2,*  
*“1 unit” on top of bar 2 and “3 units” between 2 and 4.*

**

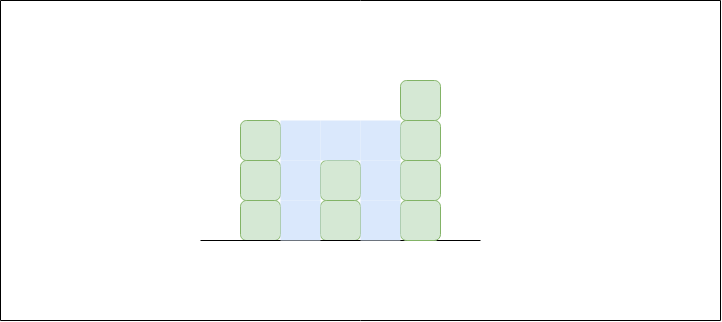
***Input:****arr[] = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1}*  
***Output:****6*  
***Explanation:****The structure is like below.*  
*Trap “1 unit” between first 1 and 2, “4 units” between*  
*first 2 and 3 and “1 unit” between second last 1 and last 2*

**

**Intuition:** The basic intuition of the problem is as follows:

* *An element of the array can store water if there are higher bars on the left and the right.*
* *The amount of water to be stored in every position can be found by finding the heights of bars on the left and right sides.*
* *The total amount of water stored is the summation of the water stored in each index.*

***For example****– Consider the array****arr[] = {3, 0, 2, 0, 4}****.*  
*Three units of water can be stored in two indexes 1 and 3, and one unit of water at index 2.*  
*Water stored in each index = 0 + 3 + 1 + 3 + 0 =****7***

**

**Approach 1 (Brute Approach):** This approach is the **brute approach**. The idea is to:

*Traverse every array element and find the highest bars on the left and right sides. Take the smaller of two heights. The difference between the smaller height and the height of the current element is the amount of water that can be stored in this array element.*

Follow the steps mentioned below to implement the idea:

* Traverse the array from start to end:
  + For every element:
    - Traverse the array from start to that index and find the maximum height *(a)* and
    - Traverse the array from the current index to the end, and find the maximum height *(b)*.
* The amount of water that will be stored in this column is min(a,b) – array[i], add this value to the total amount of water stored
* Print the total amount of water stored.

Below is the implementation of the above approach.

C++Java

// Java code to implement of the approach

class GFG {

// Function to return the maximum

// water that can be stored

public static int maxWater(int[] arr, int n)

{

// To store the maximum water

// that can be stored

int res = 0;

// For every element of the array

// except first and last element

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

// Find maximum element on its left

int left = arr[i];

for (int j = 0; j < i; j++) {

left = Math.max(left, arr[j]);

}

// Find maximum element on its right

int right = arr[i];

for (int j = i + 1; j < n; j++) {

right = Math.max(right, arr[j]);

}

// Update maximum water value

res += Math.min(left, right) - arr[i];

}

return res;

}

// Driver code

public static void main(String[] args)

{

int[] arr = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };

int n = arr.length;

System.out.print(maxWater(arr, n));

}

}

**Output**

6

**Complexity Analysis:**

* **Time Complexity:** O(N2). There are two nested loops traversing the array.
* **Space Complexity:** O(1). No extra space is required.

**Approach 2 (Precalculation):** This is an efficient solution based on the precalculation concept:

*In previous approach, for every element we needed to calculate the highest element on the left and on the right.*

*So, to reduce the time complexity:*

* *For every element we can precalculate and store the highest bar on the left and on the right (say stored in arrays****left[]****and****right[]****).*
* *Then iterate the array and use the precalculated values to find the amount of water stored in this index,*  
  *which is the same as (****min(left[i], right[i]) – arr[i]****)*

Follow the below illustration for a better understanding:

**Illustration:**

*Consider****arr[] = {3, 0, 2, 0, 4}***

*Therefore,****left[] = {3, 3, 3, 3, 4}****and****right[] = {4, 4, 4, 4, 4}***  
*Now consider iterating using****i****from 0 to end*

***For i = 0:***  
*=> left[0] = 3, right[0] = 4 and arr[0] = 3*  
*=> Water stored = min(left[0], right[0]) – arr[0] = min(3, 4) – 3 = 3 – 3 =****0***  
*=> Total = 0 + 0 =****0***

***For i = 1:***  
*=> left[1] = 3, right[1] = 4 and arr[1] = 0*  
*=> Water stored = min(left[1], right[1]) – arr[1] = min(3, 4) – 0 = 3 – 0 =****3***  
*=> Total = 0 + 3 =****3***

***For i = 2:***  
*=> left[2] = 3, right[2] = 4 and arr[2] = 2*  
*=> Water stored = min(left[2], right[2]) – arr[2] = min(3, 4) – 2 = 3 – 2 =****1***  
*=> Total = 3 + 1 =****4***

***For i = 3:***  
*=> left[3] = 3, right[3] = 4 and arr[3] = 0*  
*=> Water stored = min(left[3], right[3]) – arr[3] = min(3, 4) – 0 = 3 – 0 =****3***  
*=> Total = 4 + 3 =****7***

***For i = 4:***  
*=> left[4] = 4, right[4] = 4 and arr[4] = 4*  
*=> Water stored = min(left[4], right[4]) – arr[4] = min(4, 4) – 4 = 4 – 4 =****0***  
*=> Total = 7 + 0 =****7***

*So total rain water trapped =****7***

Follow the steps mentioned below to implement the approach:

* Create two arrays **left[]** and **right[]** of size **N**. Create a variable (say **max**) to store the maximum found till a certain index during traversal.
* Run one loop from start to end:
  + In each iteration update max and also assign **left[i] = max**.
* Run another loop from end to start:
  + In each iteration update max found till now and also assign **right[i] = max**.
* Traverse the array from start to end.
  + The amount of water that will be stored in this column is **min(left[i], right[i]) – array[i]**
  + Add this value to the total amount of water stored
* Print the total amount of water stored.

Below is the implementation of the above approach.

C++Java

// Java program to find maximum amount of water that can

// be trapped within given set of bars.

class Test {

static int arr[]

= new int[] { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };

// Method for maximum amount of water

static int findWater(int n)

{

// left[i] contains height of tallest bar to the

// left of i'th bar including itself

int left[] = new int[n];

// Right [i] contains height of tallest bar to

// the right of ith bar including itself

int right[] = new int[n];

// Initialize result

int water = 0;

// Fill left array

left[0] = arr[0];

for (int i = 1; i < n; i++)

left[i] = Math.max(left[i - 1], arr[i]);

// Fill right array

right[n - 1] = arr[n - 1];

for (int i = n - 2; i >= 0; i--)

right[i] = Math.max(right[i + 1], arr[i]);

// Calculate the accumulated water element by

// element consider the amount of water on i'th bar,

// the amount of water accumulated on this

// particular bar will be equal to min(left[i],

// right[i]) - arr[i] .

for (int i = 0; i < n; i++)

water += Math.min(left[i], right[i]) - arr[i];

return water;

}

// Driver method to test the above function

public static void main(String[] args)

{

System.out.println(findWater(arr.length));

}

}

**Output**

6

**Complexity Analysis:**

* **Time Complexity:** O(N). Only one traversal of the array is needed, So time Complexity is O(N).
* **Space Complexity:** O(N). Two extra arrays are needed, each of size N.