# Trapping Rain Water Problem - GeeksforGeeks

**Source:** https://www.geeksforgeeks.org/trapping-rain-water/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Trapping Rain Water Problem Last Updated : 3 Oct, 2025 Given an array arr[] of size n consisting of non-negative integers, where each element represents the height of a bar in an elevation map and the width of each bar is 1, determine the total amount of water that can be trapped between the bars after it rains. Trapping Rainwater Problem Examples: Input: arr[] = [3, 0, 1, 0, 4, 0, 2] Output: 10 Explanation: The expected rainwater to be trapped is shown in the above image. Input: arr[] = [3, 0, 2, 0, 4] Output: 7 Explanation: We trap 0 + 3 + 1 + 3 + 0 = 7 units. Input: arr[] = [1, 2, 3, 4] Output: 0 Explanation : We cannot trap water as there is no height bound on both sides Try it on GfG Practice Table of Content [Naive Approach] Brute Force - O(n^2) Time and O(1) Space [Better Approach] Prefix and suffix max for each index - O(n) Time and O(n) Space [Expected Approach] Using Two Pointers - O(n) Time and O(1) Space [Alternate Approach] Using Stack - O(n) Time and O(n) Space Intuition To trap water at any index in the elevation map, there must be taller bars on both its left and right sides. The water that can be stored at each position is determined by the height of the shorter of the two boundaries (left and right), minus the height of the current bar. We compute the trapped water at each index as: min(leftMax, rightMax) - height[i], if this value is positive. The total trapped water is the sum of water stored at all valid indices. If either side lacks a boundary, no water can be trapped at that position. [Naive Approach] Brute Force - O(n^2) Time and O(1) Space 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. C++ #include <iostream> #include <vector> using namespace std ; int maxWater ( vector < int >& arr ) { int res = 0 ; // For every element of the array for ( int i = 1 ; i < arr . size () - 1 ; i ++ ) { // Find the maximum element on its left int left = arr [ i ]; for ( int j = 0 ; j < i ; j ++ ) left = max ( left , arr [ j ]); // Find the maximum element on its right int right = arr [ i ]; for ( int j = i + 1 ; j < arr . size (); j ++ ) right = max ( right , arr [ j ]); // Update the maximum water res += ( min ( left , right ) - arr [ i ]); } return res ; } int main () { vector < int > arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; cout << maxWater ( arr ); return 0 ; } C #include <stdio.h> int maxWater ( int arr [], int n ) { int res = 0 ; // For every element of the array for ( int i = 1 ; i < n - 1 ; i ++ ) { // Find the maximum element on its left int left = arr [ i ]; for ( int j = 0 ; j < i ; j ++ ) if ( arr [ j ] > left ) left = arr [ j ]; // Find the maximum element on its right int right = arr [ i ]; for ( int j = i + 1 ; j < n ; j ++ ) if ( arr [ j ] > right ) right = arr [ j ]; // Update the maximum water res += ( left < right ? left : right ) - arr [ i ]; } return res ; } // Driver code int main () { int arr [] = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; int n = sizeof ( arr ) / sizeof ( arr [ 0 ]); printf ( "%d" , maxWater ( arr , n )); return 0 ; } Java class GfG { static int maxWater ( int [] arr ) { int res = 0 ; // For every element of the array for ( int i = 1 ; i < arr . length - 1 ; i ++ ) { // Find the maximum element on its left int left = arr [ i ] ; for ( int j = 0 ; j < i ; j ++ ) left = Math . max ( left , arr [ j ] ); // Find the maximum element on its right int right = arr [ i ] ; for ( int j = i + 1 ; j < arr . length ; j ++ ) right = Math . max ( right , arr [ j ] ); // Update the maximum water res += Math . min ( left , right ) - arr [ i ] ; } return res ; } public static void main ( String [] args ) { int [] arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; System . out . println ( maxWater ( arr )); } } Python def maxWater ( arr ): res = 0 # For every element of the array for i in range ( 1 , len ( arr ) - 1 ): # Find the maximum element on its left left = arr [ i ] for j in range ( i ): left = max ( left , arr [ j ]) # Find the maximum element on its right right = arr [ i ] for j in range ( i + 1 , len ( arr )): right = max ( right , arr [ j ]) # Update the maximum water res += ( min ( left , right ) - arr [ i ]) return res if __name__ == "__main__" : arr = [ 2 , 1 , 5 , 3 , 1 , 0 , 4 ] print ( maxWater ( arr )) C# using System ; class GfG { static int maxWater ( int [] arr ) { int res = 0 ; // For every element of the array for (

int i = 1 ; i < arr . Length - 1 ; i ++ ) { // Find the maximum element on its left int left = arr [ i ]; for ( int j = 0 ; j < i ; j ++ ) left = Math . Max ( left , arr [ j ]); // Find the maximum element on its right int right = arr [ i ]; for ( int j = i + 1 ; j < arr . Length ; j ++ ) right = Math . Max ( right , arr [ j ]); // Update the maximum water res += Math . Min ( left , right ) - arr [ i ]; } return res ; } static void Main () { int [] arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; Console . WriteLine ( maxWater ( arr )); } } JavaScript function maxWater ( arr ) { let res = 0 ; // For every element of the array for ( let i = 1 ; i < arr . length - 1 ; i ++ ) { // Find the maximum element on its left let left = arr [ i ]; for ( let j = 0 ; j < i ; j ++ ) left = Math . max ( left , arr [ j ]); // Find the maximum element on its right let right = arr [ i ]; for ( let j = i + 1 ; j < arr . length ; j ++ ) right = Math . max ( right , arr [ j ]); // Update the maximum water res += Math . min ( left , right ) - arr [ i ]; } return res ; } // Driver code let arr = [ 2 , 1 , 5 , 3 , 1 , 0 , 4 ]; console . log ( maxWater ( arr )); Output 9 [Better Approach] Prefix and suffix max for each index - O(n) Time and O(n) Space In the 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 first calculate 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 calculated values to find the amount of water stored in this index, which is the same as (min(left[i], right[i]) - arr[i]) C++ #include <iostream> #include <vector> using namespace std ; int maxWater ( vector < int >& arr ) { int n = arr . size (); // left[i] contains height of tallest bar to the // left of i'th bar including itself vector < int > left ( n ); // right[i] contains height of tallest bar to // the right of i'th bar including itself vector < int > right ( n ); int res = 0 ; // fill left array left [ 0 ] = arr [ 0 ]; for ( int i = 1 ; i < n ; i ++ ) left [ i ] = 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 ] = max ( right [ i + 1 ], arr [ i ]); // calculate the accumulated water element by element for ( int i = 1 ; i < n - 1 ; i ++ ) { int minOf2 = min ( left [ i ], right [ i ]); res += minOf2 - arr [ i ]; } return res ; } int main () { vector < int > arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; cout << maxWater ( arr ); return 0 ; } C #include <stdio.h> int maxWater ( int arr [], int n ) { // left[i] contains height of tallest bar to the // left of i'th bar including itself int left [ n ]; // right[i] contains height of tallest bar to // the right of i'th bar including itself int right [ n ]; int res = 0 ; // fill left array left [ 0 ] = arr [ 0 ]; for ( int i = 1 ; i < n ; i ++ ) left [ i ] = left [ i - 1 ] > arr [ i ] ? left [ i - 1 ] : arr [ i ]; // fill right array right [ n - 1 ] = arr [ n - 1 ]; for ( int i = n - 2 ; i >= 0 ; i -- ) right [ i ] = right [ i + 1 ] > arr [ i ] ? right [ i + 1 ] : arr [ i ]; // calcualte the accumulated water element by element for ( int i = 1 ; i < n - 1 ; i ++ ) { int minOf2 = left [ i ] < right [ i ] ? left [ i ] : right [ i ]; res += minOf2 - arr [ i ]; } return res ; } int main () { int arr [] = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; int n = sizeof ( arr ) / sizeof ( arr [ 0 ]); printf ( "%d" , maxWater ( arr , n )); return 0 ; } Java public class GFG { public static int maxWater ( int [] arr ) { int n = arr . length ; int [] left = new int [ n ] ; int [] right = new int [ n ] ; int res = 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 for ( int i = 1 ; i < n - 1 ; i ++ ) { int minOf2 = Math . min ( left [ i ] , right [ i ] ); res += minOf2 - arr [ i ] ; } return res ; } public static void main ( String [] args ) { int [] arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; System . out . println ( maxWater ( arr )); } } Python def maxWater ( arr ): n = len ( arr ) # left array left = [ 0 ] * n # right array right = [ 0 ] * n res = 0 # fill left array left [ 0 ] = arr [ 0 ] for i in range ( 1 , n ): left [ i ] = max ( left [ i - 1 ], arr [ i ]) # fill right array right [ n - 1 ] = arr [ n - 1 ] for i in range ( n - 2 , - 1 , - 1 ): right [ i ] = max ( right [ i + 1 ], arr [ i ]) # calculate the accumulated water element by element for i in range ( 1 , n - 1 ): min_of_2 = min ( left [ i ], right [ i ]) res += min_of_2 - arr [ i ] return res if __name__ == "__main__" : arr = [ 2 , 1 , 5 , 3 , 1 , 0 , 4 ] print ( maxWater ( arr )) C# using System ; class Program { static int MaxWater ( int [] arr ) { int n = arr . Length ; int [] left = new int [ n ]; int [] right = new int [ n ]; int res = 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 for ( int i = 1 ; i < n - 1 ; i ++ ) { int minOf2 = Math . Min ( left [ i ], right [ i ]); res += minOf2 - arr [ i ]; } return res ; } static void Main () { int [] arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; Console . WriteLine ( MaxWater ( arr )); } } JavaScript function maxWater ( arr ) { const n = arr . length ; let left = Array ( n ); let right = Array ( n ); let res = 0 ; // fill left array left [ 0 ] = arr [ 0 ]; for ( let i = 1 ; i < n ; i ++ ) { left [ i ] = Math . max ( left [ i - 1 ], arr [ i ]); } // fill right array right [ n - 1 ] = arr [ n - 1 ]; for ( let i = n - 2 ; i >= 0 ; i -- ) { right [ i ] = Math . max ( right [ i + 1 ], arr [ i ]); } // calculate the accumulated water element // by element for ( let i = 1 ; i < n - 1 ; i ++ ) { let minOf2 = Math . min ( left [ i ], right [ i ]); res += minOf2 - arr [ i ]; } return res ; } // Driver Code const arr = [ 2 , 1 , 5 , 3 , 1 , 0 , 4 ]; console . log ( maxWater ( arr )); Output 9 [Expected Approach] Using Two Pointers - O(n) Time and O(1) Space The approach is mainly based on the following facts: If we consider a subarray arr[left...right], we can decide the amount of water either for arr[left] or arr[right] if we know the left max (max element in arr[0...left-1]) and right max (max element in arr[right+1...n-1]. If left max is less than

the right max, then we can decide for arr[left]. Else we can decide for arr[right] If we decide for arr[left], then the amount of water would be left max - arr[left] and if we decide for arr[right], then the amount of water would be right max - arr[right]. How does this work? Let us consider the case when left max is less than the right max. For arr[left], we know left max for it and we also know that the right max for it would not be less than left max because we already have a greater value in arr[right...n-1]. So for the current bar, we can find the amount of water by finding the difference between the current bar and the left max bar. C++ #include <iostream> #include <vector> using namespace std ; int maxWater ( vector < int > & arr ) { int left = 1 ; int right = arr . size () - 2 ; // lMax : Maximum in subarray arr[0..left-1] // rMax : Maximum in subarray arr[right+1..n-1] int lMax = arr [ left - 1 ]; int rMax = arr [ right + 1 ]; int res = 0 ; while ( left <= right ) { // If rMax is smaller, then we can // decide the amount of water for arr[right] if ( rMax <= lMax ) { // Add the water for arr[right] res += max ( 0 , rMax - arr [ right ]); // Update right max rMax = max ( rMax , arr [ right ]); // Update right pointer as we have // decided the amount of water for this right -= 1 ; } else { // Add the water for arr[left] res += max ( 0 , lMax - arr [ left ]); // Update left max lMax = max ( lMax , arr [ left ]); // Update left pointer as we have // decided water for this left += 1 ; } } return res ; } int main () { vector < int > arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; cout << maxWater ( arr ) << endl ; return 0 ; } C #include <stdio.h> int maxWater ( int arr [], int n ) { int left = 1 ; int right = n - 2 ; // lMax : Maximum in subarray arr[0..left-1] // rMax : Maximum in subarray arr[right+1..n-1] int lMax = arr [ left - 1 ]; int rMax = arr [ right + 1 ]; int res = 0 ; while ( left <= right ) { // If rMax is smaller, then we can // decide the amount of water for arr[right] if ( rMax <= lMax ) { // Add the water for arr[right] res += ( rMax - arr [ right ]) > 0 ? ( rMax - arr [ right ]) : 0 ; // Update right max rMax = rMax > arr [ right ] ? rMax : arr [ right ]; // Update right pointer as we have // decided the amount of water for this right -= 1 ; } else { // Add the water for arr[left] res += ( lMax - arr [ left ]) > 0 ? ( lMax - arr [ left ]) : 0 ; // Update left max lMax = lMax > arr [ left ] ? lMax : arr [ left ]; // Update left pointer as we have decided water for this left += 1 ; } } return res ; } int main () { int arr [] = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; int n = sizeof ( arr ) / sizeof ( arr [ 0 ]); printf ( "%d \n " , maxWater ( arr , n )); return 0 ; } Java import java.util.* ; class GfG { static int maxWater ( int [] arr ) { int left = 1 ; int right = arr . length - 2 ; // lMax : Maximum in subarray arr[0..left-1] // rMax : Maximum in subarray arr[right+1..n-1] int lMax = arr [ left - 1 ] ; int rMax = arr [ right + 1 ] ; int res = 0 ; while ( left <= right ) { // If rMax is smaller, then we can decide // the amount of water for arr[right] if ( rMax <= lMax ) { // Add the water for arr[right] res += Math . max ( 0 , rMax - arr [ right ] ); // Update right max rMax = Math . max ( rMax , arr [ right ] ); // Update right pointer as we have decided the amount of water for this right -= 1 ; } else { // Add the water for arr[left] res += Math . max ( 0 , lMax - arr [ left ] ); // Update left max lMax = Math . max ( lMax , arr [ left ] ); // Update left pointer as we have // decided water for this left += 1 ; } } return res ; } public static void main ( String [] args ) { int [] arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; System . out . println ( maxWater ( arr )); } } Python def maxWater ( arr ): left = 1 right = len ( arr ) - 2 # lMax : Maximum in subarray arr[0..left-1] # rMax : Maximum in subarray arr[right+1..n-1] lMax = arr [ left - 1 ] rMax = arr [ right + 1 ] res = 0 while left <= right : # If rMax is smaller, then we can decide the # amount of water for arr[right] if rMax <= lMax : # Add the water for arr[right] res += max ( 0 , rMax - arr [ right ]) # Update right max rMax = max ( rMax , arr [ right ]) # Update right pointer as we have decided # the amount of water for this right -= 1 else : # Add the water for arr[left] res += max ( 0 , lMax - arr [ left ]) # Update left max lMax = max ( lMax , arr [ left ]) # Update left pointer as we have decided # the amount of water for this left += 1 return res if __name__ == "__main__" : arr = [ 2 , 1 , 5 , 3 , 1 , 0 , 4 ] print ( maxWater ( arr )) C# using System ; class GfG { static int maxWater ( int [] arr ) { int left = 1 ; int right = arr . Length - 2 ; // lMax : Maximum in subarray arr[0..left-1] // rMax : Maximum in subarray arr[right+1..n-1] int lMax = arr [ left - 1 ]; int rMax = arr [ right + 1 ]; int res = 0 ; while ( left <= right ) { // If rMax is smaller, then we can decide the // amount of water for arr[right] if ( rMax <= lMax ) { // Add the water for arr[right] res += Math . Max ( 0 , rMax - arr [ right ]); // Update right max rMax = Math . Max ( rMax , arr [ right ]); // Update right pointer as we have decided // the amount of water for this right -= 1 ; } else { // Add the water for arr[left] res += Math . Max ( 0 , lMax - arr [ left ]); // Update left max lMax = Math . Max ( lMax , arr [ left ]); // Update left pointer as we have decided // water for this left += 1 ; } } return res ; } static void Main () { int [] arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; Console . WriteLine ( maxWater ( arr )); } } JavaScript function maxWater ( arr ) { let left = 1 ; let right = arr . length - 2 ; // lMax : Maximum in subarray arr[0..left-1] // rMax : Maximum in subarray arr[right+1..n-1] let lMax = arr [ left - 1 ]; let rMax = arr [ right + 1 ]; let res = 0 ; while ( left <= right ) { // If rMax is smaller, then we can decide the // amount of water for arr[right] if ( rMax <= lMax ) { // Add the water for arr[right] res += Math . max ( 0 , rMax - arr [ right ]); // Update right max rMax = Math . max ( rMax , arr [ right ]); // Update right pointer as we have decided // the amount of water for this right -= 1 ; } else { // Add the water for arr[left] res += Math . max ( 0 , lMax - arr [ left ]); // Update left max lMax = Math . max ( lMax , arr [ left ]); // Update left pointer as

we have decided water for this left += 1 ; } } return res ; } // Driver code let arr = [ 2 , 1 , 5 , 3 , 1 , 0 , 4 ]; console . log ( maxWater ( arr )); Output 9 [Alternate Approach] Using Stack - O(n) Time and O(n) Space This approach involves using next greater and previous greater elements to solve the trapping rainwater problem . By utilizing a stack and a single traversal, we can compute both the next and previous greater elements for every item. For each element, the water trapped can be determined by the minimum height between the previous and next greater elements. The water is filled between these elements, and the process continues recursively with the next greater and previous greater elements. C++ #include <iostream> #include <vector> #include <stack> using namespace std ; int maxWater ( vector < int >& arr ) { stack < int > st ; int res = 0 ; for ( int i = 0 ; i < arr . size (); i ++ ) { // Pop all items smaller than arr[i] while ( ! st . empty () && arr [ st . top ()] < arr [ i ]) { int pop_height = arr [ st . top ()]; st . pop (); if ( st . empty ()) break ; // arr[i] is the next greater for the removed item // and new stack top is the previous greater int distance = i - st . top () - 1 ; // Take the minimum of two heights (next and prev greater) // and find the amount of water that we can fill in all // bars between the two int water = min ( arr [ st . top ()], arr [ i ]) - pop_height ; res += distance * water ; } st . push ( i ); } return res ; } int main () { vector < int > arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; cout << maxWater ( arr ); return 0 ; } C #include <stdio.h> #include <stdlib.h> struct Stack { int * array ; int top ; int capacity ; }; struct Stack * createStack ( int capacity ) { struct Stack * stack = ( struct Stack * ) malloc ( sizeof ( struct Stack )); stack -> capacity = capacity ; stack -> top = -1 ; stack -> array = ( int * ) malloc ( capacity * sizeof ( int )); return stack ; } // Stack is full when top is equal to the last index int isFull ( struct Stack * stack ) { return stack -> top == stack -> capacity - 1 ; } // Stack is empty when top is -1 int isEmpty ( struct Stack * stack ) { return stack -> top == -1 ; } // Function to add an item to stack void push ( struct Stack * stack , int item ) { if ( isFull ( stack )) return ; stack -> array [ ++ stack -> top ] = item ; } // Function to remove an item from stack int pop ( struct Stack * stack ) { if ( isEmpty ( stack )) return -1 ; return stack -> array [ stack -> top -- ]; } // Function to return the top from the stack without popping int top ( struct Stack * stack ) { return stack -> array [ stack -> top ]; } int maxWater ( int * arr , int n ) { struct Stack * st = createStack ( n ); int res = 0 ; for ( int i = 0 ; i < n ; i ++ ) { // Pop all items smaller than arr[i] while ( ! isEmpty ( st ) && arr [ top ( st )] < arr [ i ]) { int pop_height = arr [ pop ( st )]; if ( isEmpty ( st )) break ; // arr[i] is the next greater for the removed item // and new stack top is the previous greater int distance = i - top ( st ) - 1 ; // Take the minimum of two heights (next and prev greater) int water = ( arr [ top ( st )] < arr [ i ]) ? arr [ top ( st )] : arr [ i ]; // Find the amount of water water -= pop_height ; res += distance * water ; } push ( st , i ); } free ( st -> array ); free ( st ); return res ; } int main () { int arr [] = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; int n = sizeof ( arr ) / sizeof ( arr [ 0 ]); printf ( "%d \n " , maxWater ( arr , n )); return 0 ; } Java import java.util.Stack ; class GfG { static int maxWater ( int [] arr ) { Stack < Integer > st = new Stack <> (); int res = 0 ; for ( int i = 0 ; i < arr . length ; i ++ ) { // Pop all items smaller than arr[i] while ( ! st . isEmpty () && arr [ st . peek () ] < arr [ i ] ) { int pop_height = arr [ st . pop () ] ; if ( st . isEmpty ()) break ; // arr[i] is the next greater for the removed item // and new stack top is the previous greater int distance = i - st . peek () - 1 ; // Take the minimum of two heights (next and prev greater) int water = Math . min ( arr [ st . peek () ] , arr [ i ] ); // Find the amount of water water -= pop_height ; res += distance * water ; } st . push ( i ); } return res ; } public static void main ( String [] args ) { int [] arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; System . out . println ( maxWater ( arr )); } } Python def maxWater ( arr ): st = [] res = 0 for i in range ( len ( arr )): # Pop all items smaller than arr[i] while st and arr [ st [ - 1 ]] < arr [ i ]: pop_height = arr [ st . pop ()] if not st : break # arr[i] is the next greater for the removed item # and new stack top is the previous greater distance = i - st [ - 1 ] - 1 # Take the minimum of two heights (next and prev greater) water = min ( arr [ st [ - 1 ]], arr [ i ]) # Find the amount of water water -= pop_height res += distance * water st . append ( i ) return res if __name__ == "__main__" : arr = [ 2 , 1 , 5 , 3 , 1 , 0 , 4 ] print ( maxWater ( arr )) C# using System ; using System.Collections.Generic ; class GfG { static int maxWater ( int [] arr ) { Stack < int > st = new Stack < int > (); int res = 0 ; for ( int i = 0 ; i < arr . Length ; i ++ ) { // Pop all items smaller than arr[i] while ( st . Count > 0 && arr [ st . Peek ()] < arr [ i ]) { int pop_height = arr [ st . Pop ()]; if ( st . Count == 0 ) break ; // arr[i] is the next greater for the removed item // and new stack top is the previous greater int distance = i - st . Peek () - 1 ; // Take the minimum of two heights (next and prev greater) int water = Math . Min ( arr [ st . Peek ()], arr [ i ]); // Find the amount of water water -= pop_height ; res += distance * water ; } st . Push ( i ); } return res ; } static void Main () { int [] arr = { 2 , 1 , 5 , 3 , 1 , 0 , 4 }; Console . WriteLine ( maxWater ( arr )); } } JavaScript function maxWater ( arr ) { let st = []; let res = 0 ; for ( let i = 0 ; i < arr . length ; i ++ ) { // Pop all items smaller than arr[i] while ( st . length > 0 && arr [ st [ st . length - 1 ]] < arr [ i ]) { let pop_height = arr [ st . pop ()]; if ( st . length === 0 ) break ; // arr[i] is the next greater for the removed item // and new stack top is the previous greater let distance = i - st [ st . length - 1 ] - 1 ; // Take the minimum of two heights (next and prev greater) let water = Math . min ( arr [ st [ st . length -

1 ]], arr [ i ]); // Find the amount of water water -= pop_height ; res += distance * water ; } st . push ( i ); } return res ; } // Driver Code let arr = [ 2 , 1 , 5 , 3 , 1 , 0 , 4 ]; console . log ( maxWater ( arr )); Output 9 Comment Article Tags: Article Tags: Stack DSA Arrays Microsoft Amazon Adobe D-E-Shaw Accolite Payu MakeMyTrip two-pointer-algorithm + 7 More