geeksforgeeks.org

Trapping Rain Water - GeeksforGeeks

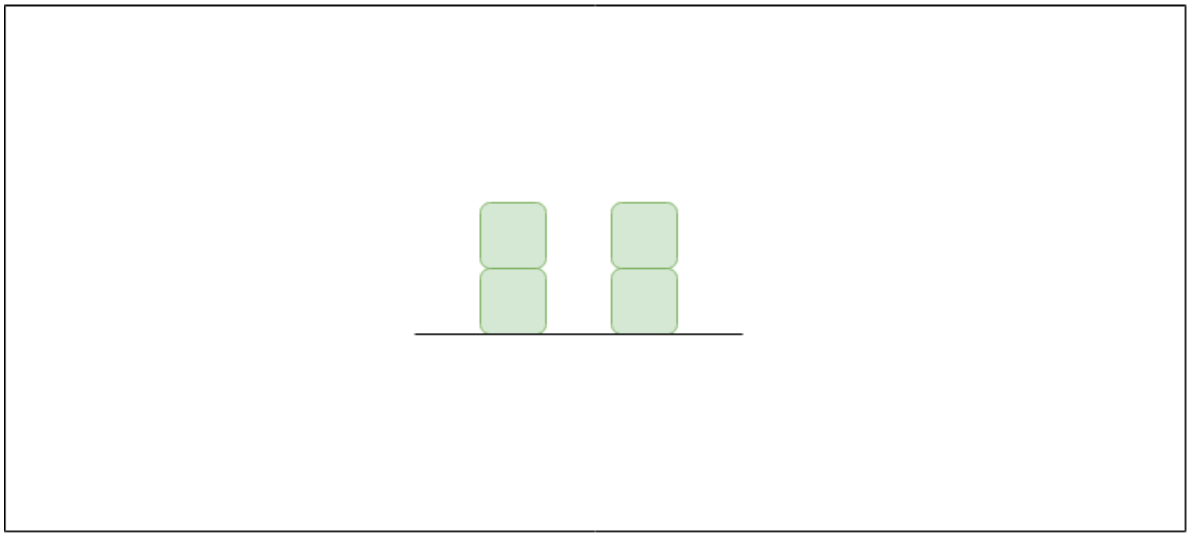
GeeksforGeeks

14-18 minutes

Given n non-negative integers 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



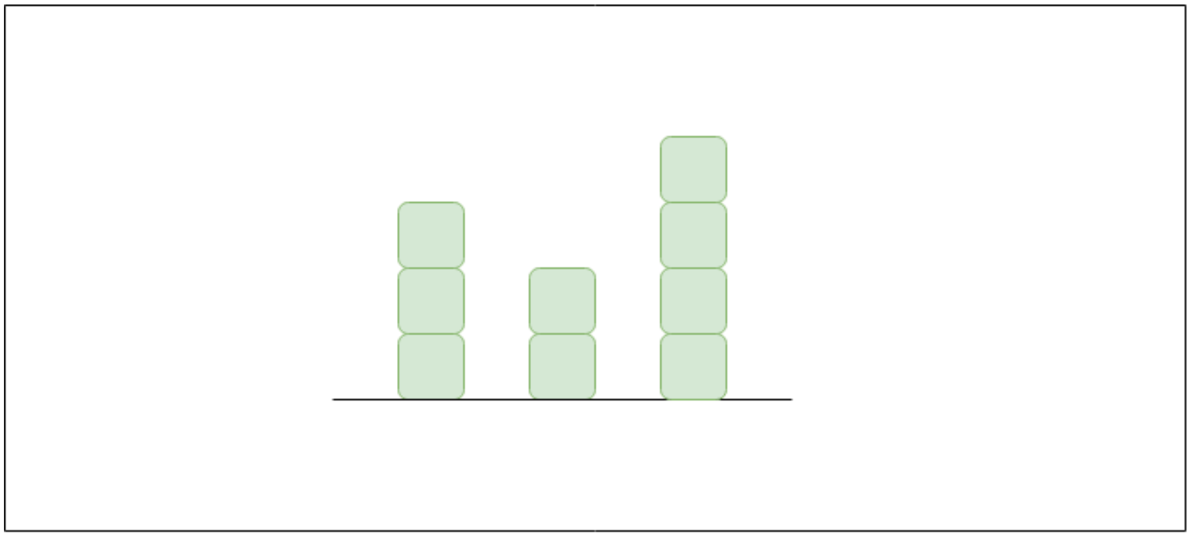
We can trap 2 units of water in the middle gap.

Input: arr[]   = {3, 0, 2, 0, 4}

Output: 7

Explanation:

Structure is like below



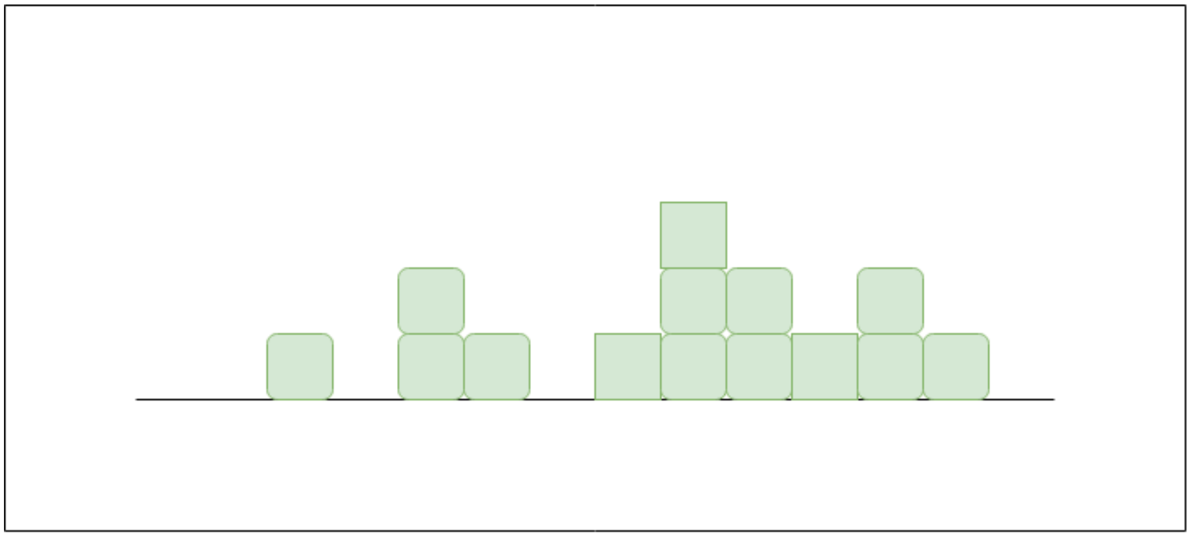
We can trap "3 units" of water between 3 and 2, "1 unit" on top of bar 2 and "3 units" between 2 and 4. See below diagram also.

Input: arr[] = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]

Output: 6

Explanation:

The structure is like below



Video Player is loading.

Current Time 0:00

Duration 0:00

Remaining Time 0:00

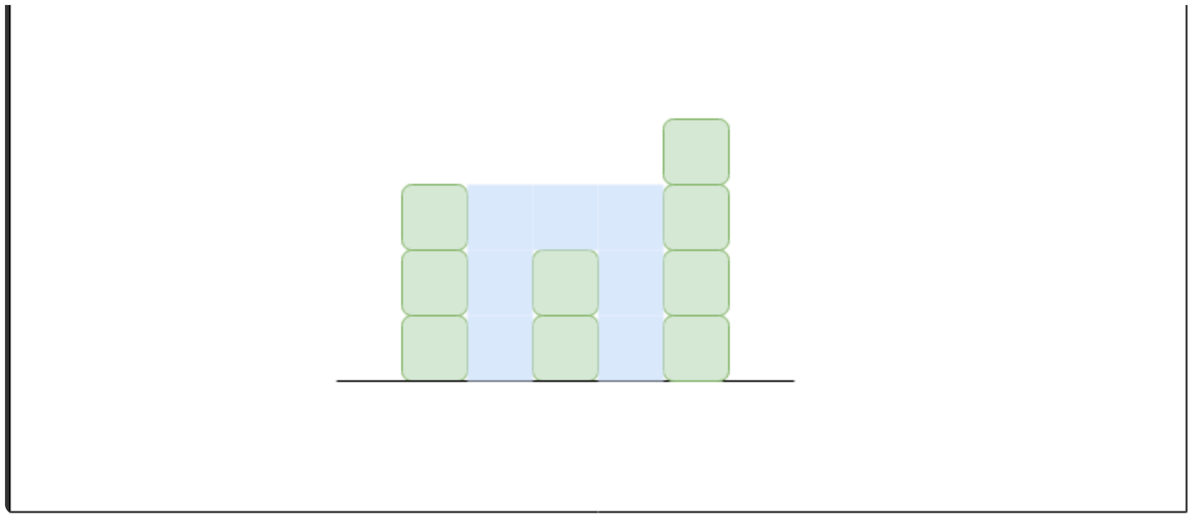
Trap "1 unit" between first 1 and 2, "4 units" between first 2 and 3 and "1 unit" between second last 1 and last 2

We strongly recommend that you click here and practice it, before moving on to the solution.

Basic Insight:
An element of the array can store water if there are higher bars on the left and right. The amount of water to be stored in every element can be found by finding the heights of bars on the left and right sides. The idea is to compute the amount of water that can be stored in every element of the array.

Example:
Consider the array {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.

> For Array[] = {3, 0, 2, 0, 4}
> Water stored = 0 + 3 + 1 + 3 + 0 = 7



Method 1: This is a simple solution to the above problem.

- **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.
- **Algorithm:**

1. Traverse the array from start to end.
2. 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 end, and find the maximum height *(b)*.
3. 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
4. Print the total amount of water stored.

- **Implementation:**
- C++
- C
- Java

- Python3
- C#
- Javascript

C++

```
#include<bits/stdc++.h>

using namespace std;

int maxWater(int arr[], int n)

{

    int res = 0;

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

        int left = arr[i];

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

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

        int right = arr[i];

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

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

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

    }

    return res;

}

int main()

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

    int n = sizeof(arr)/sizeof(arr[0]);

    cout << maxWater(arr, n);

    return 0;

}
```

C

Java

Python3

C#

Javascript

- **Complexity Analysis:**
- **Time Complexity:** O(n^2).
  There are two nested loops traversing the array, So the time Complexity is O(n^2).
- **Space Complexity:** O(1).
  No extra space is required.

Method 2: This is an efficient solution to the above problem.

- **Approach:** In the previous solution, to find the highest bar on the left and right, array traversal is needed, which reduces the efficiency of the solution. To make this efficient, one must pre-compute the highest bar on the left and right of every bar in linear time. Then use these pre-computed values to find the amount of water in every array element.

- **Algorithm:**

1. Create two arrays *left* and *right* of size n. create a variable *max_* = *INT_MIN*.
2. Run one loop from start to end. In each iteration update max_ as *max_ = max(max_, arr[i])* and also assign *left[i] = max_*
3. Update max_ = INT_MIN.
4. Run another loop from end to start. In each iteration update max_ as *max_ = max(max_, arr[i])* and also assign *right[i] = max_*
5. Traverse the array from start to end.
6. The amount of water that will be stored in this column is *min(a,b)* – *array[i]*,(where a = left[i] and b = right[i]) add this value to total amount of water stored
7. Print the total amount of water stored.

- **Implementation:**
- C++
- C
- Java
- Python3
- C#
- PHP
- Javascript

C++

```
#include <bits/stdc++.h>

using namespace std;

int findWater(int arr[], int n)

{

    int left[n];

    int right[n];

    int water = 0;

    left[0] = arr[0];

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

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

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

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

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

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

    {

      int var=min(left[i-1],right[i+1]);

      if(var > arr[i])

      {

        water += var - arr[i];

      }

    }

    return water;
}

int main()

{

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

    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Maximum water that can be accumulated is
"

         << findWater(arr, n);

    return 0;

}
```

C

Java

Python3

C#

PHP

Javascript

Output

Maximum water that can be accumulated is 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.

  Space Optimization for the **above Solution:**

  Instead of maintaining two arrays of size n for storing the left and a right max of each element, maintain two variables to store the maximum till that point. Since water trapped at any element = *min(max_left, max_right) – arr[i]*. Calculate water trapped on smaller elements out of A[lo] and A[hi] first, and move the pointers till *lo* doesn't cross *hi*.

- **Implementation:**

- C++
- Java
- Python3
- C#
- PHP
- Javascript
- C

C++

```
#include <iostream>

using namespace std;

int findWater(int arr[], int n)

{
    int result = 0;

    int left_max = 0, right_max = 0;

    int lo = 0, hi = n - 1;

    while (lo <= hi) {

        if (arr[lo] < arr[hi]) {

            if (arr[lo] > left_max)

                left_max = arr[lo];

            else

                result += left_max - arr[lo];

            lo++;

        }

        else {

            if (arr[hi] > right_max)

                right_max = arr[hi];

            else

                result += right_max - arr[hi];

            hi--;

        }

    }

    return result;

}

int main()
```

```
{
    int arr[] = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1
};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Maximum water that can be accumulated is
"
         << findWater(arr, n);
}
```

Java

Python3

C#

PHP

Javascript

C

Output

Maximum water that can be accumulated is 6

- **Complexity Analysis:**
- **Time Complexity**: O(n).
  Only one traversal of the array is needed.
- **Auxiliary Space**: O(1).
  As no extra space is required.

- *Thanks to Gaurav Ahirwar and Aditi Sharma for* the *above solution.*

Method 3: Here another efficient solution has been shown.

- **Approach:** The concept here is that if there is a larger wall to the right, then the water can be retained with a height equal to the smaller wall on the left. If there are no larger walls to the right, then start from the left. There must be a larger wall to the left now. Let's take an example of the heights are {….,3, 2, 1, 4,….}, So here 3 and 4 are boundaries the heights 2 and 1 are submerged and cannot act as boundaries. So at any point or index knowing the previous boundary is sufficient if there is a higher or equal length boundary in the remaining part of the array. If not, then traverse the array backward and now must be a larger wall to the left.

- **Algorithm:**
- Loop from index 0 to the end of the given array.
- If a wall greater than or equal to the previous wall is encountered, then make note of the index of that wall in a var called prev_index.
- Keep adding the previous wall's height minus the current (ith) wall to the variable water.
- Have a temporary variable that stores the same value as water.
- If no wall greater than or equal to the previous wall is found, then quit.
- If prev_index < size of the input array, then subtract the temp variable from water, and loop from the end of the input array to prev_index. Find a wall greater than or equal to the previous wall

(in this case, the last wall from backward).

- **Implementation:**
- C++
- Java
- Python3
- C#
- Javascript
- C

C++

```
#include<iostream>
using namespace std;
int maxWater(int arr[], int n)
{
    int size = n - 1;
    int prev = arr[0];
    int prev_index = 0;
    int water = 0;
    int temp = 0;
    for(int i = 1; i <= size; i++)
    {
        if (arr[i] >= prev)
        {
            prev = arr[i];

            prev_index = i;

            temp = 0;

        }

        else

        {

            water += prev - arr[i];

            temp += prev - arr[i];

        }

    }

    if(prev_index < size)

    {

        water -= temp;

        prev = arr[size];

        for(int i = size; i >= prev_index; i--)

        {

            if(arr[i] >= prev)

            {

                prev = arr[i];

            }

            else

            {
                water += prev - arr[i];
            }
        }
    }
    return water;
}
int main()
{
    int arr[] = { 0, 1, 0, 2, 1, 0,
                  1, 3, 2, 1, 2, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << maxWater(arr, n);
    return 0;
}
```

Java

Python3

C#

Javascript

C

- **Complexity Analysis:**
- **Time Complexity**: O(n).

As only one traversal of the array is needed.

- **Auxiliary Space**: O(1).
  As no extra space is required.

Method 4 (Using Stack):

We can use a Stack to track the bars that are bounded by the longer left and right bars. This can be done using only one iteration using stacks.

Approach:

1. Loop through the indices of the bar array.

2. For each bar, we can do the following:

- While the Stack is not empty and the current bar has a height greater than the top bar of the stack,
- Store the index of the top bar in **pop_height** and pop it from the Stack.
- Find the distance between the left bar(current top) of the popped bar and the current bar.
- Find the minimum height between the top bar and the current bar.
- The maximum water that can be trapped in **distance * min_height**.
- The water trapped, including the popped bar, is **(distance * min_height) – height[pop_height]**.
- Add that to the fans**.**

3. Final answer will the **ans**.

- C++

- Java
- Python3
- C#
- Javascript

C++

```
#include <bits/stdc++.h>

using namespace std;

int maxWater(int height[], int n)

{

    stack <int> st;

    int ans = 0;

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

    {

        while ((!st.empty()) &&

               (height[st.top()] < height[i]))

        {

            int pop_height = height[st.top()];

            st.pop();

            if (st.empty())

                break;

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

            int min_height = min(height[st.top()],
                                 height[i]) -
                             pop_height;
            ans += distance * min_height;
        }
        st.push(i);
    }
    return ans;
}
int main()
{
    int arr[] = { 0, 1, 0, 2, 1, 0,
                  1, 3, 2, 1, 2, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << maxWater(arr, n);
    return 0;
}
```

Java

Python3

C#

Javascript

Time Complexity: O(n)

Auxiliary Space: O(n)

Method 5 (Two Pointer Approach)

- **Approach**: At every index, The amount of rainwater stored is the difference between the current index height and a minimum of left max height and right max-height
- **Algorithm** :
- Take two pointers l and r. Initialize l to the starting index 0 and r to the last index n-1
- Since l is the first element, left_max would be 0, and right max for r would be 0
- While l <= r, iterate the array. We have two possible conditions
- **Condition1** : **left_max <= right max**
- Consider Element at index l
- Since we have traversed all elements to the left of l, **left_max is known**
- For the right max of l, We can say that the **right max would always be >= current r_max here**
- So, **min(left_max,right_max)** would always equal to **left_max** in this case
- Increment l
- **Condition2 : left_max > right max**
- Consider Element at index r
- Since we have traversed all elements to the right of r, **right_max is known**
- For the left max of l, We can say that the **left max would always**

be >= current l_max here

- So, **min(left_max,right_max)** would always equal to **right_max** in this case
- Decrement r
- **Implementation:**
- C++
- Java
- Python3
- C#
- Javascript
- C

C++

```
#include <iostream>

using namespace std;

int maxWater(int arr[], int n)

{

    int left = 0;

    int right = n-1;

    int l_max = 0;

    int r_max = 0;

    int result = 0;

    while (left <= right)
    {
      if(r_max <= l_max)
      {
        result += max(0, r_max-arr[right]);
        r_max = max(r_max, arr[right]);
        right -= 1;
      }
      else
      {
        result += max(0, l_max-arr[left]);
        l_max = max(l_max, arr[left]);
        left += 1;
      }
    }
    return result;
}
int main() {
    int arr[] = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << maxWater(arr, n) << endl;
    return 0;
}
```

Java

Python3

C#

Javascript

C

Time Complexity: O(n)
Auxiliary Space: O(1)

Method 6 (horizontal scan method)

- **Approach** If max_height is the height of the tallest block, then it will be the maximum possible height for any trapped rainwater. And if we traverse each row from left to right for each row from bottom to max_height, we can count the number of blocks that will contain water.
- **Algorithm**
- Find the total number of blocks, i.e., sum of the heights array, **num_blocks**
- find the maximum height, **max_height**
- store total water in a variable, **total** = 0
- keep two pointers, **left = 0** and **right = n -1**, we will use them later
- for each row **i** from 0 to max_height, do the following
- Scan heights array from left to right, if height > **i** (current row), this will be the boundary of one of the sections of trapped water, let us store these indexes.

- Let the boundary indexes be boundary = [i1, i2, i3 …]
- The water stored in current row between i1 and i2 = i2 – i1 – 1, between i2 and i3 = i3 – i2 – 1 and so on.
- Total water stored in current row = (i2 – i1 – 1) + (i3 – i2 – 1) + … (i_n – i_{n-1} -1) = **i_n – i_1 – (k – 1)**, where k in number of blocks in this row
- if we find i_1 and i_n i.e., first and last boundary index of each row, we don't need to scan the whole array. We will not find k, since it is a number of blocks in current row, it will accumulate and it will add up to total number of blocks, which we have found already, and the 1 will accumulate for each row, i.e., it will become max_height
- find i_1 by traversing from **left** towards right till you find h where h > **i** (current row), similarly, find i_n by traversing from **right** towards left.
- Set **left = i_1** and **right = i_n** and add i_n – i_1 to total
- subtract **num_blocks – max_height** from total, which was accumulated
- **total** will be the quantity of trapped water
- **Implementation**
- C++
- Java
- Python
- Javascript

C++

```
#include <bits/stdc++.h>

using namespace std;

int trappedWater(vector<int>&heights){

    int num_blocks = 0;

    int n = 0;

    int max_height = INT_MIN;

    for(auto height : heights){

        num_blocks += height;

        n += 1;

        max_height = max(max_height, height);

    }

    int total = 0;

    int left = 0;

    int right = n - 1;

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

    {

        while(heights[left] <= i)

            left += 1;

        while(heights[right] <= i)

            right -= 1;

        total += right - left;

    }
    total -= (num_blocks - max_height);

    return total;

}

int main()

{

  vector<int> heights = {0,1,0,2,1,0,1,3,2,1,2,1};

  cout << trappedWater(heights) << endl;

  return 0;

}
```

Java

Python

Javascript

Time Complexity: O(max_height * k) where k is total change in left and right pointers per row, k < n
Space Complexity: O(1)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.