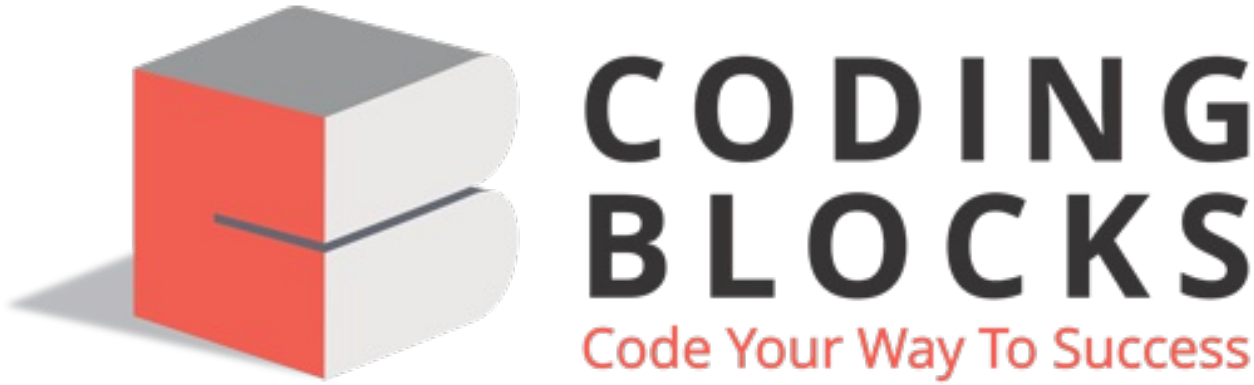

Problem Name : Rain Water Harvesting

EXPLANATION

An element of the array can store water if there are higher bars on left and right. We can find the amount of water to be stored in every element by finding the heights of bars on left and right sides. The idea is to compute the amount of water that can be stored in every element of array. For example, consider the array {3, 0, 0, 2, 0, 4}, we can store three units of water at indexes 1 and 2, and one unit of water at index 3, and three units of water at index 4.

A Simple Solution is to traverse every array element and find the highest bars on left and right sides. Take the smaller of two heights. The difference between the smaller height and height of the current element is the amount of water that can be stored in this array element. Time complexity of this solution is O(n2).

Naive Approach

```
int maxWater(int arr[], int n)
{
```

```
// To store the maximum water
// that can be stored
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++)
       left = max(left, arr[j]);

    // Find the maximum element on its right
    int right = arr[i];
    for (int j=i+1; j<n; j++)
       right = max(right, arr[j]);

   // Update the maximum water
   res = res + (min(left, right) - arr[i]);
}

return res;
```

}

#Better Approach

An element of an array can store water if there are higher bars on left and right. We can find the amount of water to be stored in every element 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. For example, consider the array {3, 0, 0, 2, 0, 4}, we can store two units of water at indexes 1 and 2, and one unit of water at index 2.

Pre-compute highest bar on left and right of every bar in O(n) time. Then use these pre-computed values to find the amount of water in every array element.

C++ Code

```
#include
using namespace std;
int a[1000000], l[1000000], r[1000000];
int main()
{
    int n, i, j;
    cin >> n;
    for (i = 0; i < n; i++)
    {
        cin >> a[i];
    }
    l[0] = a[0];
    r[n - 1] = a[n - 1];
    int leftmax = 0, rightmax = 0;
    for (i = 1; i < n; i++)
    {
        l[i] = max(l[i - 1], a[i]);
    }
    for (i = n - 2; i >= 0; i--)
    {
        r[i] = max(r[i + 1], a[i]);
    }
    int water = 0;
    for (i = 0; i < n; i++)
    {
        water += (min(l[i], r[i]) - a[i]);
    }
    cout << water;
```

```
return 0;
```

```
}
```

Java Code

```
import java.util.*;

public class Practice {

static int[] a = new int[1000000];

static int[] r = new int[1000000];

static int[] l = new int[1000000];
```

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    for (int i = 0; i < n; i++) {
        a[i] = sc.nextInt();
    }
    l[0] = a[0];
    r[n - 1] = a[n - 1];
    int leftmax = 0, rightmax = 0;
    for (int i = 1; i < n; i++) {
        l[i] = Math.max(l[i - 1], a[i]);
    }
    for (int i = n - 2; i >= 0; i--) {
        r[i] = Math.max(r[i + 1], a[i]);
    }
    int water = 0;
    for (int i = 0; i < n; i++) {
        water += (Math.min(l[i], r[i]) - a[i]);
    }
    System.out.println(water);
}
```

```
}
```

Best Approach

Use the two pointer approach.
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 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 end of the input array to prev_index and find a wall greater than or equal to the previous wall (in this case, the last wall from backwards).

C++ Code

```
int maxWater_optimized(int arr[], int n)
{
    int water = 0; // To store the final ans
```

```
int left_max = 0;  // Which stores the current max height of the left side
int right_max = 0; // Which stores the current max height of the right side

int lo = 0;     // Counter to traverse from the left_side
int hi = n - 1; // Counter to traverse from the right_side

while (lo <= hi)
{

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

        if (arr[lo] > left_max)
        {
            left_max = arr[lo]; // Updating left_max
        }
        else
        {

            water += left_max - arr[lo]; // Calculating the ans
        }
        lo++;
    }
    else
    {

        if (arr[hi] > right_max)
        {
            right_max = arr[hi]; // Updating right_max
        }
        else
        {
            water += right_max - arr[hi]; // Calculating the ans
        }
        hi--;
    }
}

return water;
```

```
}
```

Java Code

```
public static int maxWater_optimized(int[] arr){
```

```
int n = arr.length;
int water = 0;  // To store the final ans

int left_max = 0;     //Which stores the current max height of the left side
int right_max = 0;   //Which stores the current max height of the right side

int lo = 0;                //Counter to traverse from the left_side
int hi = n - 1;           //Counter to traverse from the right_side

while(lo <= hi){

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

    if(arr[lo] > left_max){
      left_max = arr[lo];             //Updating left_max
    }else{

      water += left_max - arr[lo];  //Calculating the ans
    }
    lo++;
  }else{

    if(arr[hi] > right_max){
      right_max = arr[hi];               //Updating right_max
    }else{
      water += right_max - arr[hi];  //Calculating the ans
    }
    hi--;
  }

}

return water;
```

```
}
```