42. Trapping Rain Water

Problem Statement: Given an array of non-negative integers representation elevation of ground. Your task is to find the water that can be trapped after rain.

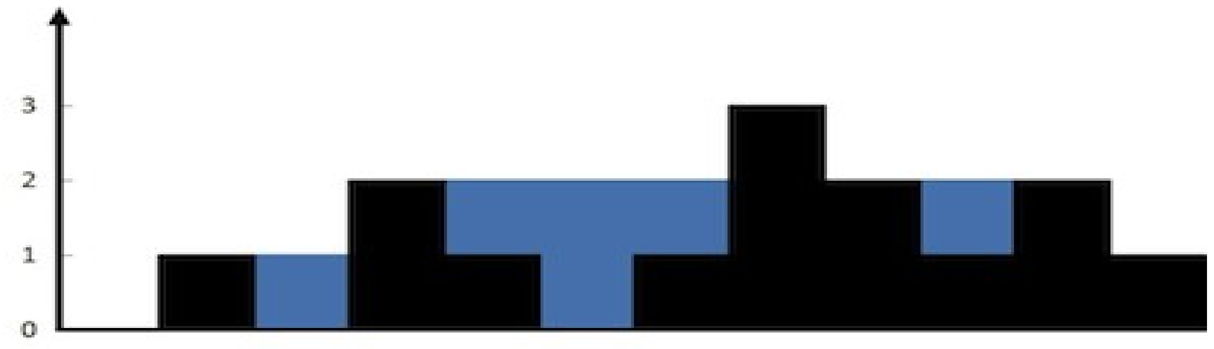
Examples:

Example 1:

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

Output: 6

Explanation: As seen from the diagram 1+1+2+1+1=6 unit of water can be trapped



Example 2:

Input: [4,2,0,3,2,5]

Output:9

Solution

Solution 1: Brute force

Approach: For each index, we have to find the amount of water that can be stored and we have to sum it up.If we observe carefully the amount the water stored at a particular index is the minimum of maximum elevation to the left and right of the index minus the elevation at that index.

Code:

```
import java.util.*;
class Main {
    static int trap(int[] arr) {
        int n = arr.length;
        int waterTrapped = 0;
        for (int i = 0; i < n; i++) {
            int j = i;
            int leftMax = 0, rightMax = 0;
            while (j >= 0) {
                leftMax = Math.max(leftMax, arr[j]);
                j--;
            }
            j = i;
            while (j < n) {
                rightMax = Math.max(rightMax, arr[j]);
                j++;
            }
            waterTrapped += Math.min(leftMax, rightMax) - arr[i];
        }
        return waterTrapped;
    }
    public static void main(String args[]) {
        int arr[] = {0,1,0,2,1,0,1,3,2,1,2,1};
        System.out.println("The duplicate element is " + trap(arr));
    }
}
```

Output: The water that can be trapped is 6

Time Complexity: O(N*N) as for each index we are calculating leftMax and rightMax so it is a nested loop.

Space Complexity: O(1).

Solution 2:Better solution

Intuition: We are taking O(N) for computing leftMax and rightMax at each index. The complexity can be boiled down to O(1) if we precompute the leftMax and rightMax at each index.

Approach: Take 2 array prefix and suffix array and precompute the leftMax and rightMax for each index beforehand. Then use the formula min(prefix[I], suffix[i])-arr[i] to compute water trapped at each index.

Code:

```
import java.util.*;
class Main {
    static int trap(int[] arr) {
        int n = arr.length;
        int prefix[] = new int[n];
        int suffix[] = new int[n];
        prefix[0] = arr[0];
        for (int i = 1; i < n; i++) {
            prefix[i] = Math.max(prefix[i - 1], arr[i]);
        }
        suffix[n - 1] = arr[n - 1];
        for (int i = n - 2; i >= 0; i--) {
            suffix[i] = Math.max(suffix[i + 1], arr[i]);
        }
        int waterTrapped = 0;
        for (int i = 0; i < n; i++) {
            waterTrapped += Math.min(prefix[i], suffix[i]) - arr[i];
        }
        return waterTrapped;
    }

    public static void main(String args[]) {
        int arr[] = {0,1,0,2,1,0,1,3,2,1,2,1};
        System.out.println("The duplicate element is " + trap(arr));
    }
}
```

Output: The water that can be trapped is 6

Time Complexity: O(3*N) as we are traversing through the array only once. And O(2*N) for computing prefix and suffix array.

Space Complexity: O(N)+O(N) for prefix and suffix arrays.

Solution 3:Optimal Solution(Two pointer approach)

Approach: Take 2 pointers l(left pointer) and r(right pointer) pointing to 0th and (n-1)th index respectively. Take two variables leftMax and rightMax and initialize them to 0. If height[l] is less than or equal to height[r] then if leftMax is less than height[l] update leftMax to height[l] else add leftMax-height[l] to your final answer and move the l pointer to the right i.e l++. If height[r] is less than height[l], then now we are dealing with the right block. If height[r] is greater than rightMax, then update rightMax to height[r] else add rightMax-height[r] to the final answer. Now move r to the left. Repeat these steps till l and r crosses each other.

Intuition: We need a minimum of leftMax and rightMax.So if we take the case when height[l]<=height[r] we increase l++, so we can surely say that there is a block with a height more than height[l] to the right of l. And for the same reason when height[r]<=height[l] we can surely say that there is a block to the left of r which is at least of height[r]. So by traversing these cases and using two pointers approach the time complexity can be decreased without using extra space.

Code:

```
import java.util.*;
class Main {
    static int trap(int[] height) {
        int n = height.length;
        int left = 0, right = n - 1;
        int res = 0;
        int maxLeft = 0, maxRight = 0;
        while (left <= right) {
            if (height[left] <= height[right]) {
                if (height[left] >= maxLeft) {
                    maxLeft = height[left];
                } else {
                    res += maxLeft - height[left];
                }
                left++;
            } else {
                if (height[right] >= maxRight) {
                    maxRight = height[right];
                } else {
                    res += maxRight - height[right];
                }
                right--;
            }
        }
        return res;
    }


    public static void main(String args[]) {
        int arr[] = {0,1,0,2,1,0,1,3,2,1,2,1};
        System.out.println("The duplicate element is " + trap(arr));
    }
}
```

Output: The water that can be trapped is 6

Time Complexity: O(N) because we are using 2 pointer approach.

Space Complexity: O(1) because we are not using anything extra.