**QPREP10-Trapping rain water**

**Module Introduction**

Write a program to take as input n positive integers representing the elevation map of an area and output how much water can be trapped during rains.

Problem Solving

#### Objective

Given n non-negative integers as input, representing an elevation map where the width of each bar is 1, compute how much water can get trapped during rains.



The above elevation map is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (grey section) are being trapped.

#### Note

This is a slightly harder problem - after spending some time thinking about the problem, feel free to see the Problem Pseudocode Hint and write the solution from there. The important realization from this module should be that if you have spent enough time thinking about the problem using examples and find out an approach that you can explain easily, writing code should be a cakewalk.

#### Examples**Example 1**

Input:

8 --> Number of elements in array

0 1 0 2 1 0 1 3 --> Array elements

Output:

5

**Example 2**

Input:

3

50 100 150

Output:

0

***SOLUTION STEPS FROM NEXT PAGE:***

**Write down at least 3 examples in the following format. Kindly stick to the format.**

**Suggestion:**

EXAMPLE#1

INPUT:

3

8 5 8

OUTPUT:

3

EXAMPLE#2

INPUT:

3

50 100 50

OUTPUT:

0

EXAMPLE#3

INPUT:

5

50 50 20 30 50

OUTPUT:

50

**Detail your problem understanding here**

**Suggestion:**

The input array of positive integers represents the elevation at each position. Each position is 1 unit wide. For each position, we have to find how much water can accumulate in that position during the rains.

Rain gets accumulated at a position only if it has higher bars on either side. For each position, we have to find the highest bars on either side. Take the minimum of these highest bars, then subtract the height of the elevation in the current position to find the number of units of rain water that can accumulate at this position.

**Does this problem follow a known algorithmic pattern or standard application of a data structure? If there are multiple approaches, which one would you choose and why? Write down your chosen approach in 2-3 sentences like you would explain to a 10 year old.**

We can use brute force to check the highest bars on the left and right of each position.

However, this can be optimized.

We can iterate once over the entire array to find the left maximum for each position and another iteration to find the right maximum for each position. During these iterations, we use the first calculated values to help calculate further values, reducing iteration. This is an application of Dynamic Programming.

**Write the pseudocode here in plain English**

Get the input array of integers

Iterate over the array to find the left maximum for each position

Iterate over the array to find the right maximum for each position

Iterate over the array to find the water that can be stored at each position using the left maximum and right maximum created above

Add up this water amount for each position and return the answer

IterateToFindLeftMaxForEachPosition() {

For each position starting from Left to Right

leftMax for this position = Maximum of the current height and previous leftMax

}

IterateToFindRightMaxForEachPosition() {

For each position starting from Right to Left

rightMax for this position = Maximum of the current height and previous rightMax

}

IterateToPopulateWaterAtEachPosition() {

For each position starting from Left to Right

Total water collected += (Minimum of Left Max and Right Max) - Current position height

}

**Can you specify a few boundary or edge cases here?**

**Edge cases**

EXAMPLE#1

INPUT:

5

1 1 1 1 1

OUTPUT:

0

EXAMPLE#2

INPUT:

5

1 2 3 4 5

OUTPUT:

0

EXAMPLE#3

INPUT:

4

9 8 7 6

OUTPUT:

0

EXAMPLE#4

INPUT:

1

99

OUTPUT:

0

**Write the functions you would create here**

int[] findLeftMaxForEachPosition(inputArray)

int[] findRightMaxForEachPosition(inputArray)

Int populateWaterAtEachPosition(leftMaxArray, rightMaxArray, inputArray)

#### Summary

Starting with a brief explanation of the problem statement followed by pseudocode and then implementing the solution helps you approach the problem in a systematic way. This methodology helps with easy as well as hard problems.

**Time Complexity: O(n)**

Where n is the length of the input array. We will iterate through the array one or two times leading to this complexity.

**Space Complexity: O(n)**

We store the left maximum and right maximum for each of the n indices.

You can further optimize this to have space complexity of O(1), think about it.

#### Concepts

Concepts covered in this Module

* Array
* Dynamic Programming
* Two Pointers

Similar problems

* <https://leetcode.com/problems/container-with-most-water/>
* <https://leetcode.com/problems/product-of-array-except-self/>
* <https://leetcode.com/problems/trapping-rain-water-ii/>

References

* <https://www.geeksforgeeks.org/solve-dynamic-programming-problem/>
* <https://www.geeksforgeeks.org/dynamic-programming/#concepts>
* <https://www.geeksforgeeks.org/two-pointers-technique/>

SOLUTION:

APPROACH 1:

import java.io.\*;

import java.util.\*;

class TrappingRainWater {

// complete the below function implementation

public int trap(int[] height) {

int waterValue =0;

int min;

for(int i =0;i<height.length;i++)

{

min = min(height,i);

waterValue = waterValue + (min-height[i]);

}

return waterValue;

}

public int min(int[] height, int index)

{

int min,leftMax=0,rightMax=0;

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

{

if(height[i]>leftMax)

leftMax=height[i];

}

for(int i =index;i<height.length;i++)

{

if(height[i]>rightMax)

rightMax=height[i];

}

if(leftMax<rightMax)

min = leftMax;

else

min = rightMax;

return min;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

int height[] = new int[n];

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

height[i] = scanner.nextInt();

}

scanner.close();

int result = new TrappingRainWater().trap(height);

System.out.println(result);

}

}

**Complexity Analysis:**

* **Time Complexity:**
* **Space Complexity:**

public int trap(int[] height) {

if(height.length==0)

return 0;

int[] left = new int[height.length];

left[0] = height[0];

for(int i=1;i<left.length;i++)

{

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

}

int[] right = new int[height.length];

right[right.length-1] = height[height.length-1];

for(int i=right.length-2;i>=0;i--)

{

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

}

int waterValue =0;

int min;

for(int i =0;i<height.length;i++)

{

min = Math.min(right[i], left[i]);

waterValue = waterValue + (min-height[i]);

}

return waterValue;

}