1. Problem Identification

Problem Title: Generalized Trapping Rainwater (Variable Widths & Negative Topography)

Core Concept: Imagine a cross-section of a landscape represented by a series of vertical bars. After a heavy rainfall, water collects in the depressions between these bars. The objective is to calculate exactly how much water remains trapped after the rain stops. Water trapped in a valley is determined by the height of the walls surrounding it; specifically, water can only rise as high as the shorter of the two tallest boundaries on its left and right sides.

This specific version of the problem introduces two unique complexities that differ from the standard textbook example:

1. Negative Heights (Deep Trenches): In standard problems, the ground is flat (zero height) and bars rise up from it. In this variation, the input list contains negative numbers.

- **Interpretation: A negative number acts as a trench, hole, or basement that dips below the main ground level (sea level).**
- **Impact: These deep pockets drastically increase the volume of water. If a hole has a height of "negative 5" and the water level settles at "positive 3," the total depth of the water at that specific point is 8 units (5 units to reach the surface, plus 3 units above ground).**

2. Variable Widths: In standard problems, every bar is assumed to be exactly 1 unit wide. In this variation, every bar has a specific width coupled with its height.

- **Interpretation: The landscape is irregular. Some hills might be narrow spikes, while others are wide plateaus.**
- **Impact: The calculation cannot simply count "blocks" of water. Instead, for every segment of the terrain, we must calculate the volume by multiplying the Depth of the water by the specific Width of that segment.**

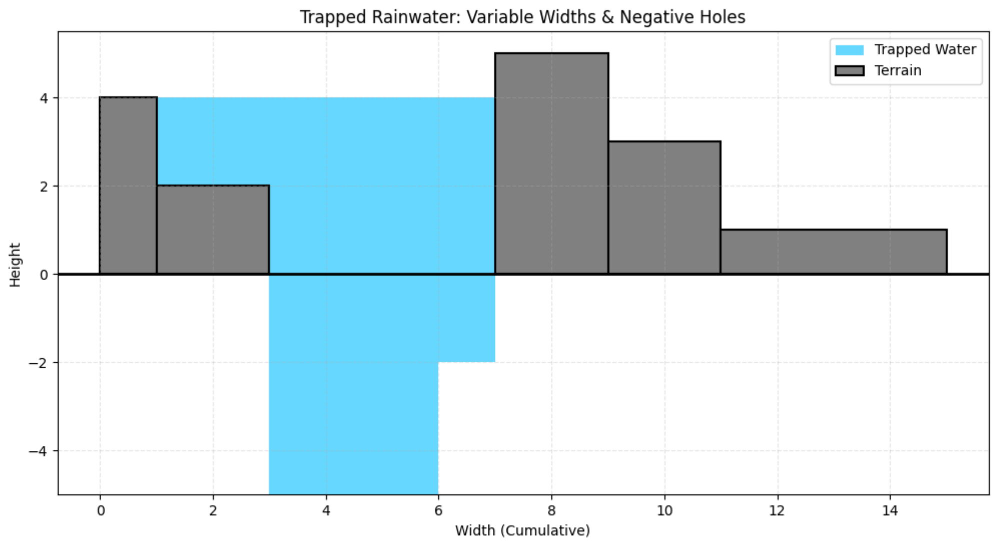
The Rules of Water: To solve this, we must simulate two physical rules:

- **Gravity: Water always flows downwards. It cannot form a hill of water; it must have a flat surface.**
- **Containment: Water at any specific point can only rise to the level of the *limiting boundary*. The limiting boundary is the lower of the two maximum heights found to the far left and far right of the current position. If the terrain height is below this limit, water fills the gap.**

2. Examples

Example 1:

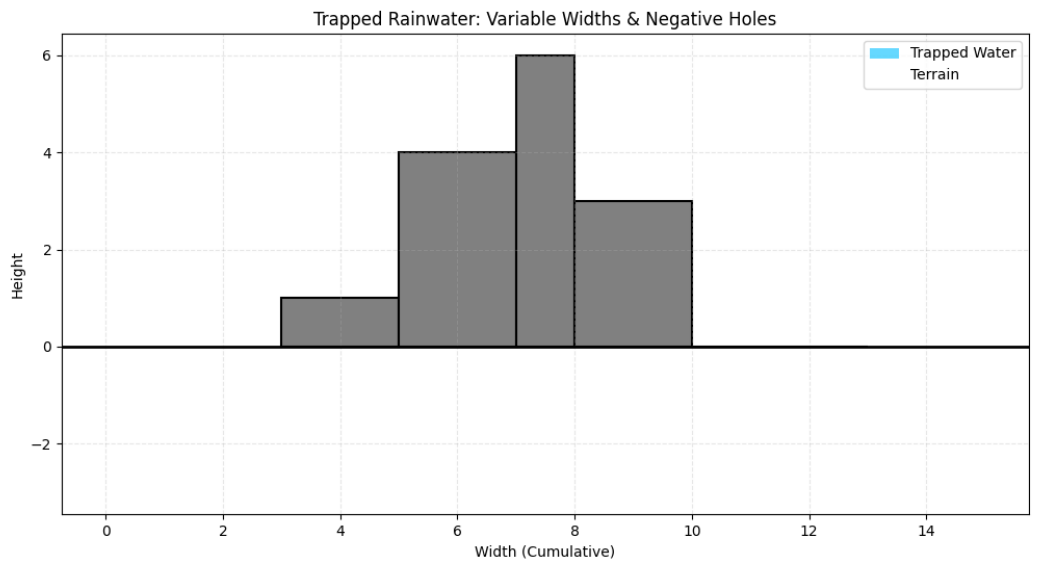
- **Heights (H):** [4, 2, -5, -2, 5, 3, 1]
- **Widths (W):** [1, 2, 3, 1, 2, 2, 4]
- **Trapped water:** 34



Example 2: Edge Case

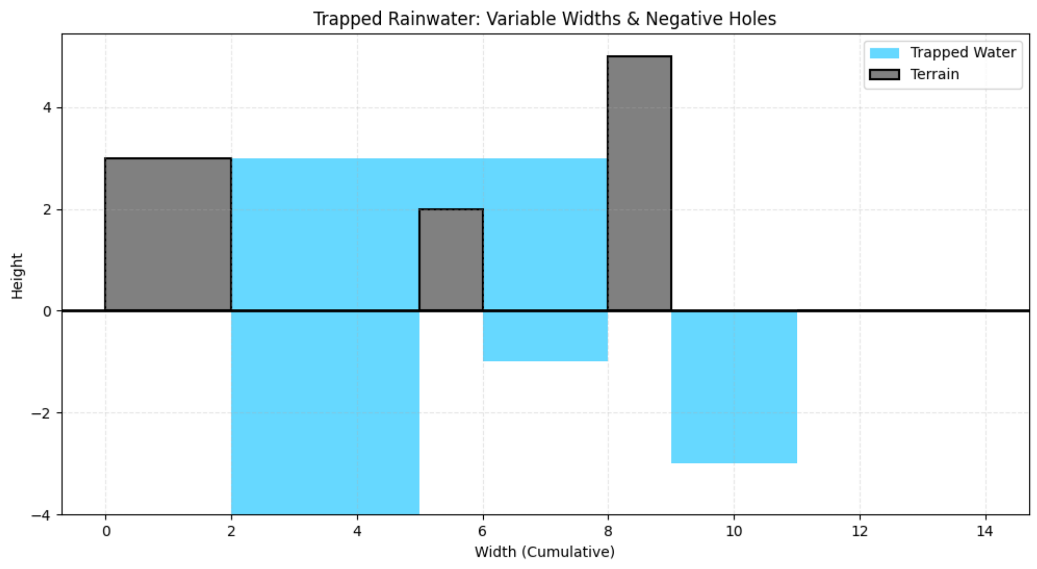
This example demonstrates a scenario where both ends has no boundaries and both ends start with holes(which will leak water to both sides).

- **Heights (H):** [-2, 1, 4, 6, 3, 0, -3]
- **Widths (W):** [3, 2, 2, 1, 2, 3, 2]
- **Trapped Water:** 0



Example 3:

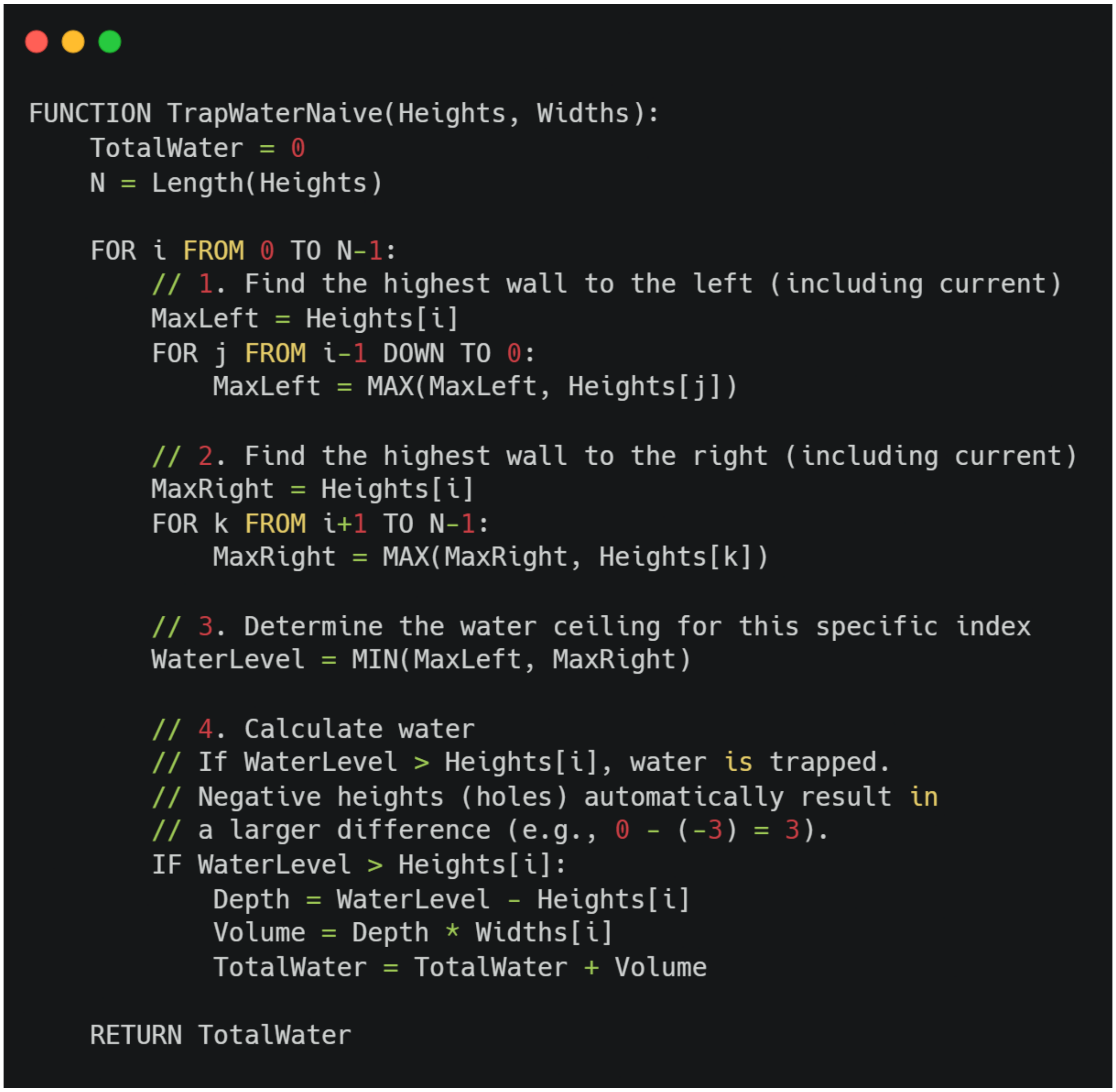
- **Heights (H):** [3, -4, 2, -1, 5, -3, 0]
- **Widths (W):** [2, 3, 1, 2, 1, 2, 3]
- **Trapped water:** 18



3. Pseudocode Solutions

A. Naive Solution (Brute Force)

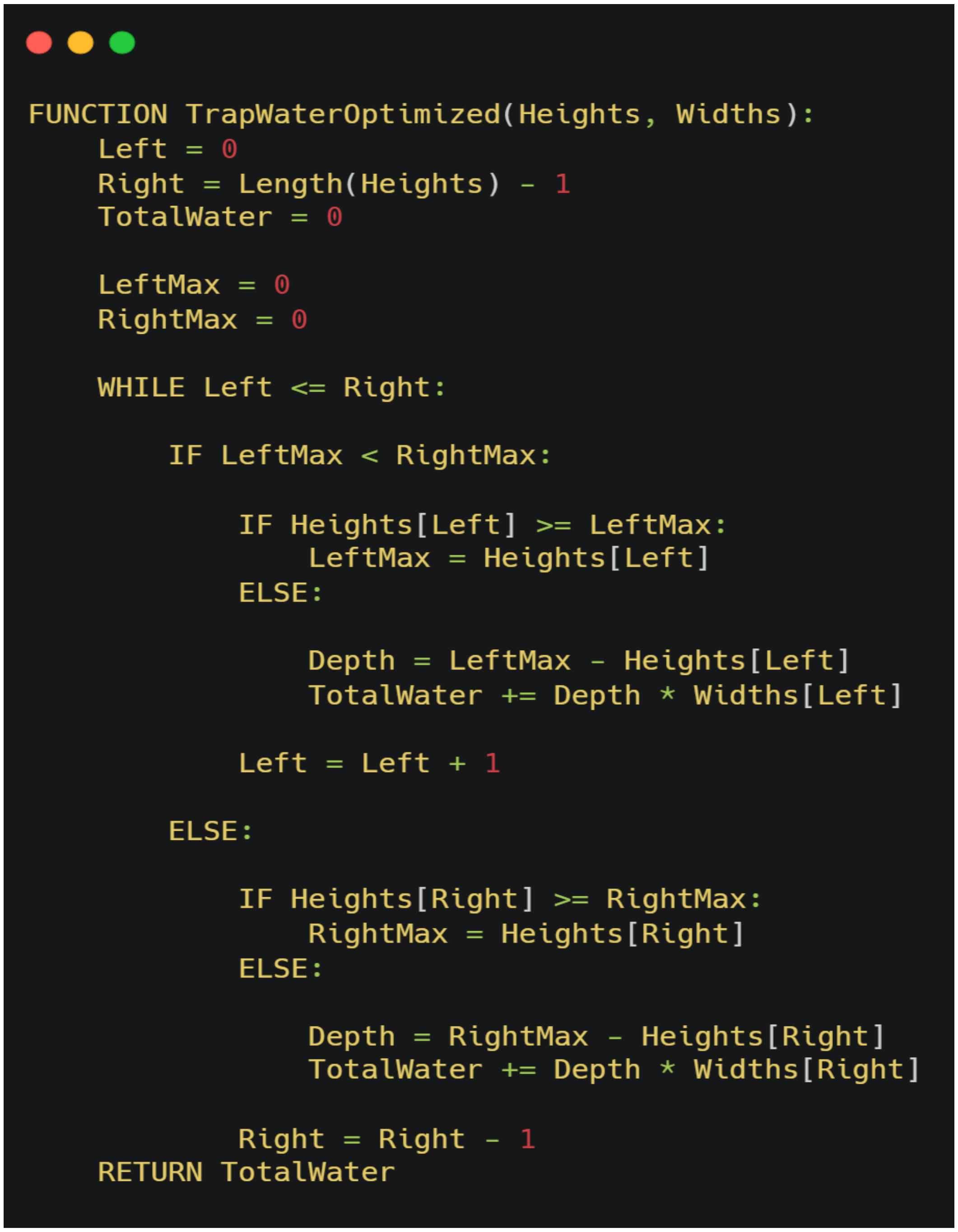
Logic: For every bar in the list, we iterate through the entire array to find the highest wall to its left and the highest wall to its right. The water level is determined by the shorter of these two walls.



```
FUNCTION TrapWaterNaive(Heights, Widths):
    TotalWater = 0
    N = Length(Heights)

    FOR i FROM 0 TO N-1:
        // 1. Find the highest wall to the left (including current)
        MaxLeft = Heights[i]
        FOR j FROM i-1 DOWN TO 0:
            MaxLeft = MAX(MaxLeft, Heights[j])

        // 2. Find the highest wall to the right (including current)
        MaxRight = Heights[i]
        FOR k FROM i+1 TO N-1:
            MaxRight = MAX(MaxRight, Heights[k])

        // 3. Determine the water ceiling for this specific index
        WaterLevel = MIN(MaxLeft, MaxRight)

        // 4. Calculate water
        // If WaterLevel > Heights[i], water is trapped.
        // Negative heights (holes) automatically result in
        // a larger difference (e.g., 0 - (-3) = 3).
        IF WaterLevel > Heights[i]:
            Depth = WaterLevel - Heights[i]
            Volume = Depth * Widths[i]
            TotalWater = TotalWater + Volume

    RETURN TotalWater
```

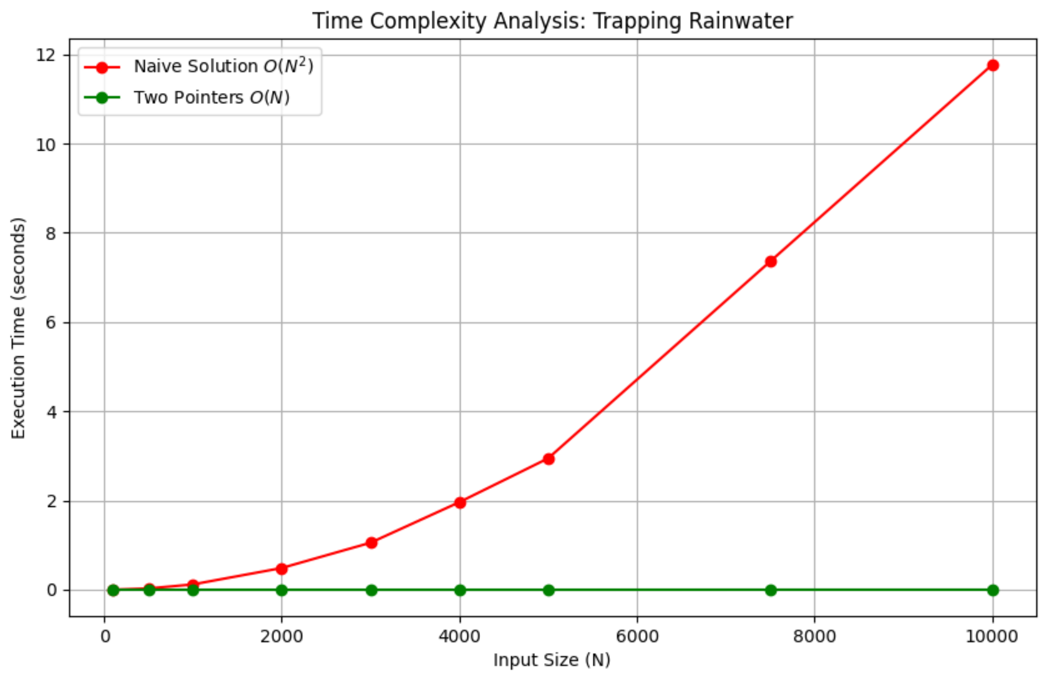
B. Optimized Solution (Two Pointers)

Logic: Instead of rescanning the array for every element, we maintain two pointers (left and right) and track the maximum height seen so far from both ends (LeftMax and RightMax). We always process the side with the smaller maximum height because that side is the limiting factor for water trapping.



```
FUNCTION TrapWaterOptimized(Heights, Widths):
    Left = 0
    Right = Length(Heights) - 1
    TotalWater = 0

    LeftMax = 0
    RightMax = 0

    WHILE Left <= Right:

        IF LeftMax < RightMax:

            IF Heights[Left] >= LeftMax:
                LeftMax = Heights[Left]
            ELSE:

                Depth = LeftMax - Heights[Left]
                TotalWater += Depth * Widths[Left]

            Left = Left + 1

        ELSE:

            IF Heights[Right] >= RightMax:
                RightMax = Heights[Right]
            ELSE:

                Depth = RightMax - Heights[Right]
                TotalWater += Depth * Widths[Right]

            Right = Right - 1
    RETURN TotalWater
```

4. Complexity Analysis



- **Line A (Steep Curve):** "Naive Solution (O(N^2))".
- **Line B (Straight Line):** "Two Pointers (O(N))".

Explanation:

1. Naive Solution (O(N^2)):

For every single element i (of which there are N), we scan the left side and the right side to find boundaries. In the worst case, this nested loop results in roughly N * N operations. As the input size doubles, the time taken quadruples.

2. Two Pointers (O(N)):

We iterate through the list using two pointers that move toward each other. Each element in the Heights and Widths lists is visited exactly once. If the input size doubles, the time taken simply doubles. This is linear time complexity, which is optimal for this problem.

5. Discussion

The core difference between the two solutions lies in **redundancy**.

- **The Naive approach** suffers from heavy redundancy. When calculating the boundaries for index i, we scan the entire array. When we move to i+1, we discard that information and scan the entire array again, oblivious to the fact that the "max height" usually hasn't changed much.

- **The Two Pointers approach** utilizes "memory" (state). By remembering the LeftMax and RightMax as we traverse, we eliminate the need to look back.

Regarding the Problem Variations:

- **Variable Widths:** This adds a simple multiplication step (O(1) arithmetic) to the calculation. It does not affect the algorithmic complexity (Big O) of either solution.

- **Negative Numbers (Holes):** This changes the arithmetic value of the depth, but not the logic. Since Boundary - (-Hole) = Boundary + |Hole|, the standard subtraction formula naturally handles the "deepening" effect of holes without requiring special if-else branches for negative numbers.

Conclusion:

For small datasets, both solutions work. However, for large datasets (e.g., thousands of terrain points), the Two Pointers solution is mandatory as the Naive solution will become exponentially slower.