407. Trapping Rain Water II

Hard | Breadth-First Search | Array | Matrix | Heap (Priority Queue)

Problem Description

The challenge is to calculate the volume of water that could be trapped after raining on a 2D elevation map. The map is represented as a matrix where each cell's value indicates the height above some arbitrary baseline (e.g., sea level). The problem is quite analogous to trapping water in 3D space. Imagine each cell of the height map as a block that can trap water depending on the heights of the surrounding blocks. The goal is to determine the total amount of water that can be trapped by the elevation map after it has rained.

Intuition

The intuition behind the solution is to use a priority queue (or a min-heap) to keep track of the cells on the perimeter of the current water being trapped. Those cells effectively form a boundary, or a "wall," which dictates how much water can be contained inside them.

We can think of filling up a container with water; water will fill up from the borders inwards and will rise to the height of the shortest wall around it. In other words, the capacity of water at any point in our map is determined by the smallest height on its boundary.

To implement this:

1. We initially add all the border cells to our priority queue since they cannot hold water and therefore set our initial boundary.
2. Then, we continuously expand our water boundary inwards by considering the shortest wall (the wall with the minimum height) on our current boundary, which we obtain from our priority queue. Every time we choose a wall, we look at its adjacent cells.
3. If any adjacent cell is shorter than the chosen wall, it means it's a candidate for holding water and the difference in height is the water it can contain.
4. We then update the boundaries by adding the surrounding cells of the chosen wall to the priority queue. If the water can be trapped from those cells, it would be trapped at a height not lower than the tallest of walls we've encountered so far.
5. This means each time we visit a new cell from the priority queue; we are assured that its capacity to hold water has been determined by the tallest wall around the cells that we've visited thus far.

Repeat this process until all cells have been visited, which ensures all cells that can trap water have been considered. The sum of all the trapped water gives us the result.

Solution Approach

The solution leverages a min-heap (implemented in Python as a priority queue) to efficiently keep track of the current smallest height on the boundary, which represents a potential "wall" that can hold water.

Here is a step-by-step breakdown of the implementation:

1. **Initialize Structures**: A `visited` matrix (same dimensions as the height map) is initialized to keep track of whether a cell has been processed. A priority queue `pq` is used to keep track of the boundary cells. We store tuples of `(height, i, j)` in the queue, where `height` is the cell's height, and `(i, j)` are the cell's coordinates on the map.

2. **Populate Priority Queue**: Loop through all cells on the map. Mark border cells (the first and last row, the first and last column) as visited and push them onto the priority queue since they can't contain any water.

3. **Process Internal Cells**: Until the priority queue is empty, do the following: a. Pop the cell with the lowest height from our priority queue. This cell is part of the current lowest boundary. b. Check all four adjacent cells (using the directional offsets in `dirs`). For each unvisited neighbor, calculate the potential water it could trap, which is the difference between the height of the popped cell and the neighbor's height, if the neighbor's height is less. c. If water can be trapped, add the volume to the answer (`ans`), and push the neighbor onto the priority queue with an updated height to reflect the maximum boundary height encountered so far. d. Mark the neighbor as visited.

4. **Return Result**: Once the priority queue is empty, all cells that could hold water have been processed, and the `ans` variable contains the total volume of trapped water.

The use of a min-heap ensures that we always expand from the current lowest wall, and because we update visited cells with the maximum height encountered, we assure that any future water trapped will respect previous boundaries. This is similar to a "water level" rising to the height of the lowest enclosing boundary, ensuring that we don't miscalculate the volume by "spilling" over lower boundaries that haven't been considered yet.

Example Walkthrough

Let's consider a small `4x4` elevation map as an example:

```
Height Map:
4 4 4 4
4 1 2 4
4 2 1 4
4 4 4 4
```

Following the solution approach for this elevation map:

1. **Initialize Structures**: Create a `visited` 2D matrix initialized with `False` and a priority queue `pq`.

2. **Populate Priority Queue**: Add all border cells (first and last rows, and first and last columns) to the `pq` and mark them as visited in the `visited` matrix.

   - Initial `pq`: [(4,0,0), (4,0,1), (4,0,2), (4,0,3), (4,1,0), (4,3,0), (4,1,3), (4,2,3), (4,3,1), (4,3,2), (4,3,3)]
3. **Process Internal Cells**: Start popping cells from `pq`. For example:

   a. Pop `(4,0,0)` from `pq`. It's a border cell; no adjacent cells are unvisited or able to trap water.

   b. Continue popping border cells until we reach an internal cell. Let's say we now pop `(4,1,1)`.

   c. Check its neighbors: [(1,1,2), (1,2,1)]. These cells are lower than the current cell, so they can potentially hold water.

   d. Calculate trapped water for each neighbor:

   - For `(1,1,2)` with height `2`, it could trap `4 - 2 = 2` units of water.
   - For `(1,2,1)` with height `2`, it could trap `4 - 2 = 2` units of water.

   e. Add neighbor cells to `pq` with updated heights showing the maximum wall height:

   - Add `(4,1,2)` for `(1,1,2)` and mark as visited.
   - Add `(4,2,1)` for `(1,2,1)` and mark as visited.

   f. Continue this process with the new cells in the `pq` until no more cells can be visited.

4. **Return Result**: After processing all cells, we calculate that there is a total of `4` units of trapped water (2 units above cell [1,2] and 2 units above cell [2,1]) in the map.

By using the priority queue to systematically expand the boundary and track the maximum wall height, we efficiently simulate the rise of water level and accurately calculate the total volume of trapped water.

Python Solution

```python
from heapq import heappop, heappush

class Solution:
    def trapRainWater(self, height_map: List[List[int]]) -> int:
        # Get the dimensions of the map
        rows, cols = len(height_map), len(height_map[0])

        # Initialize a 2D visited array to keep track of processed cells
        visited = [[False] * cols for _ in range(rows)]

        # Priority Queue (min heap) to process the cells by height
        min_heap = []

        # Initialize the heap with the boundary cells and mark them as visited
        for i in range(rows):
            for j in range(cols):
                if i == 0 or i == rows - 1 or j == 0 or j == cols - 1:
                    heappush(min_heap, (height_map[i][j], i, j))
                    visited[i][j] = True

        # Total amount of trapped water
        trapped_water = 0

        # Directions for neighboring cells
        directions = ((-1, 0), (0, 1), (1, 0), (0, -1))

        # Process the cells until the heap is empty
        while min_heap:
            height, x, y = heappop(min_heap)
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                # Check if the neighbor is within bounds and not visited
                if 0 <= nx < rows and 0 <= ny < cols and not visited[nx][ny]:
                    # Calculate the possible water level difference
                    trapped_water += max(0, height - height_map[nx][ny])

                    # Mark the neighbor as visited
                    visited[nx][ny] = True

                    # Push the neighbor cell onto the heap with the max height
                    # to keep track of the 'water surface' level
                    heappush(min_heap, (max(height, height_map[nx][ny]), nx, ny))

        # Return the total accumulated trapped water
        return trapped_water
```

Java Solution

```java
import java.util.PriorityQueue;

public class Solution {

    // Method to calculate the total trapped rainwater in the given height map
    public int trapRainWater(int[][] heightMap) {
        int rows = heightMap.length, cols = heightMap[0].length;
        boolean[][] visited = new boolean[rows][cols]; // Track visited cells
        PriorityQueue<int[]> minHeap = new PriorityQueue<>((a, b) -> a[0] - b[0]); // Min-heap based on height

        // Initialize the min-heap with the boundary cells and mark them as visited
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (i == 0 || i == rows - 1 || j == 0 || j == cols - 1) {
                    minHeap.offer(new int[]{heightMap[i][j], i, j});
                    visited[i][j] = true;
                }
            }
        }

        int totalWater = 0; // Variable to store total trapped water
        // Direction vectors (up, right, down, left)
        int[] dirX = {-1, 0, 1, 0};
        int[] dirY = {0, 1, 0, -1};

        // Process cells in the priority queue
        while (!minHeap.isEmpty()) {
            int[] current = minHeap.poll();
            // Iterate over all four adjacent cells
            for (int k = 0; k < 4; ++k) {
                int newRow = current[1] + dirX[k], newCol = current[2] + dirY[k];

                // Check bounds and visited status of the adjacent cell
                if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && !visited[newRow][newCol]) {
                    // Update total water based on the height difference
                    totalWater += Math.max(0, current[0] - heightMap[newRow][newCol]);
                    // Mark the adjacent cell as visited
                    visited[newRow][newCol] = true;
                    // Add the adjacent cell to the priority queue with the max border height
                    minHeap.offer(new int[]{Math.max(current[0], heightMap[newRow][newCol]), newRow, newCol});
                }
            }
        }

        return totalWater; // Return the total amount of trapped rainwater
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int trapRainWater(vector<vector<int>>& height_map) {
        // Define a tuple for priority queue to store the height and the coordinates
        using HeightAndCoordinates = tuple<int, int, int>;

        // Priority queue to store the boundary bars' height in ascending order
        priority_queue<HeightAndCoordinates, vector<HeightAndCoordinates>, greater<HeightAndCoordinates>> min_heap;

        // Get the dimensions of the height map
        int rows = height_map.size(), cols = height_map[0].size();

        // Visited matrix to keep track of the visited cells
        vector<vector<bool>> visited(rows, vector<bool>(cols, false));

        // Mark the boundary cells as visited and add them to the min_heap
        for (int i = 0; i < rows; ++i) {
            for (int j = 0; j < cols; ++j) {
                if (i == 0 || i == rows - 1 || j == 0 || j == cols - 1) {
                    min_heap.emplace(height_map[i][j], i, j);
                    visited[i][j] = true;
                }
            }
        }

        // Initialize the water trapped accumulator to 0
        int trapped_water = 0;

        // Directions array to facilitate the traversal of adjacent cells
        // Up, Right, Down, Left
        const int directions[5] = {-1, 0, 1, 0, -1};

        // Process cells until there are no more cells in the priority queue
        while (!min_heap.empty()) {
            auto [current_height, row, col] = min_heap.top();
            min_heap.pop();

            // Check all 4 possible directions
            for (int k = 0; k < 4; ++k) {
                int new_row = row + directions[k];
                int new_col = col + directions[k + 1];

                // Check if the new cell is within bounds and not visited
                if (new_row >= 0 && new_row < rows && new_col >= 0 && new_col < cols && !visited[new_row][new_col]) {
                    // Update trapped water if the adjacent cell's height is less than the current cell's height
                    trapped_water += max(0, current_height - height_map[new_row][new_col]);

                    // Mark the cell as visited
                    visited[new_row][new_col] = true;

                    // Push the maximum height of the adjacent cell or current cell into the min_heap
                    min_heap.emplace(max(height_map[new_row][new_col], current_height), new_row, new_col);
                }
            }
        }

        // Return the total trapped water
        return trapped_water;
    }
};
```

Typescript Solution

```typescript
// Define a type for the priority queue to store the height and the coordinates
type HeightAndCoordinates = [number, number, number];

// Function to trap rainwater using a height map
function trapRainWater(heightMap: number[][]): number {
    // Create a Min Heap to store boundary bars' height in ascending order
    const minHeap: HeightAndCoordinates[] = [];

    // Comparator function for Min Heap
    const compare: (a: HeightAndCoordinates, b: HeightAndCoordinates) => number = ([heightA], [heightB]) => heightA - heightB;

    // Helper function to heapify the minHeap
    const heapify = (index: number) => {
        let smallest = index;
        const left = 2 * index + 1;
        const right = 2 * index + 2;

        if (left < minHeap.length && compare(minHeap[left], minHeap[smallest]) < 0) {
            smallest = left;
        }
        if (right < minHeap.length && compare(minHeap[right], minHeap[smallest]) < 0) {
            smallest = right;
        }
        if (smallest !== index) {
            [minHeap[index], minHeap[smallest]] = [minHeap[smallest], minHeap[index]];
            heapify(smallest);
        }
    };

    // Helper function to extract the top element from the heap
    const extractMin = (): HeightAndCoordinates | undefined => {
        if (minHeap.length === 0) return undefined;
        const min = minHeap[0];
        minHeap[0] = minHeap[minHeap.length - 1];
        minHeap.pop();
        heapify(0);
        return min;
    };

    // Helper function to insert elements in the heap
    const insertHeap = (element: HeightAndCoordinates) => {
        minHeap.push(element);
        let i = minHeap.length - 1;
        while (i !== 0 && compare(minHeap[Math.floor((i - 1) / 2)], minHeap[i]) > 0) {
            [minHeap[i], minHeap[Math.floor((i - 1) / 2)]] = [minHeap[Math.floor((i - 1) / 2)], minHeap[i]];
            i = Math.floor((i - 1) / 2);
        }
    };

    const rows: number = heightMap.length;
    const cols: number = heightMap[0].length;

    // Visited matrix to keep track of visited cells
    const visited: boolean[][] = Array.from(new Array(rows), () => new Array(cols).fill(false));

    // Mark the boundary cells as visited and add them to the minHeap
    for (let i = 0; i < rows; i++) {
        for (let j = 0; j < cols; j++) {
            if (i === 0 || i === rows - 1 || j === 0 || j === cols - 1) {
                insertHeap([heightMap[i][j], i, j]);
                visited[i][j] = true;
            }
        }
    }

    // Initialize the accumulator for the trapped water to 0
    let trappedWater: number = 0;

    // Directions array to facilitate traversal of adjacent cells (up, right, down, left)
    const directions: number[] = [-1, 0, 1, 0, -1];

    // Process cells until the priority queue is empty
    while (minHeap.length) {
        const [currentHeight, row, col] = extractMin()!;

        // Check all 4 potential directions
        for (let k = 0; k < 4; k++) {
            const newRow: number = row + directions[k];
            const newCol: number = col + directions[k + 1];

            // Check if the new cell is within bounds and has not been visited
            if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && !visited[newRow][newCol]) {
                // Update trapped water if the adjacent cell's height is less than the current height
                trappedWater += Math.max(0, currentHeight - heightMap[newRow][newCol]);

                // Mark the cell as visited
                visited[newRow][newCol] = true;

                // Add the maximum height of the adjacent or current cell to the minHeap
                insertHeap([Math.max(heightMap[newRow][newCol], currentHeight), newRow, newCol]);
            }
        }
    }

    // Return total trapped water
    return trappedWater;
}
```

Time and Space Complexity

The time complexity of the code is `O(M*N*log(M+N))`. This is because the code uses a min-heap to keep track of the boundary cells of the height map which could potentially store at most `O(M+N)` elements (the perimeter of the map). Each cell is pushed and popped exactly once from the priority queue (since visited cells are marked and not revisited) resulting in `O(log(M+N))` for each push and pop operation and there are `M*N` cells.

The space complexity of the code is `O(M*N)`. This is due to two factors. First is the priority queue, which can grow up to the number of boundary cells, which is `O(M+N)` in the worst case. Second is the visited matrix `vis`, which is of size `M*N`. As `M*N` is typically larger than `M+N` for non-trivial maps, the overall space complexity is dominated by the `vis` matrix.