407. Trapping Rain Water II Heap (Priority Quaue) Breadth-First Search Matrix Array Leetcode Link Hard

Problem Description

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.

The challenge is to calculate the volume of water that could be trapped after raining on a 2D elevation map. The map is represented

Intuition

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.

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

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
- 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.

determined by the tallest wall around the cells that we've visited thus far.

visited and push them onto the priority queue since they can't contain any water.

Solution Approach

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

The solution leverages a min-heap (implemented in Python as a priority queue) to efficiently keep track of the current smallest height

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

queue. This cell is part of the current lowest boundary. b. Check all four adjacent cells (using the directional offsets in dirs). For

contains the total volume of trapped water.

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

3. Process Internal Cells: Until the priority queue is empty, do the following: a. Pop the cell with the lowest height from our priority

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

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:

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

○ 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:

in the visited matrix.

Height Map:

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:

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

Get the dimensions of the map

Following the solution approach for this elevation map:

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

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

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

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.

rise of water level and accurately calculate the total volume of trapped water.

def trapRainWater(self, height_map: List[List[int]]) -> int:

Initialize a 2D visited array to keep track of processed cells

Initialize the heap with the boundary cells and mark them as visited

if i == 0 or i == rows - 1 or j == 0 or j == cols - 1:

Check if the neighbor is within bounds and not visited

// Update total water based on the height difference

// Define a tuple for priority queue to store the height and the coordinates

// Priority queue to store the boundary bars' height in ascending order

if (i == 0 || i == rows - 1 || j == 0 || j == cols - 1) {

// Mark the adjacent cell as visited

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

visited[newRow][newCol] = true;

int trapRainWater(vector<vector<int>>>& height_map) {

// Get the dimensions of the height map

using HeightAndCoordinates = tuple<int, int, int>;

int rows = height_map.size(), cols = height_map[0].size();

vector<vector<bool>> visited(rows, vector<bool>(cols, false));

min_heap.emplace(height_map[i][j], i, j);

// Visited matrix to keep track of the visited cells

visited[i][j] = true;

totalWater += Math.max(0, current[0] - heightMap[newRow][newCol]);

// 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});

priority_queue<HeightAndCoordinates, vector<HeightAndCoordinates>, greater<HeightAndCoordinates>> min_heap;

heappush(min_heap, (height_map[i][j], i, j))

Priority Queue (min heap) to process the cells by height

rows, cols = len(height_map), len(height_map[0])

visited = [[False] * cols for _ in range(rows)]

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

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).

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

from heapq import heappop, heappush

visited[i][j] = True 19 20 # Total amount of trapped water 22 trapped_water = 0

```
23
24
           # Directions for neighboring cells
25
           directions = ((-1, 0), (0, 1), (1, 0), (0, -1))
26
27
           # Process the cells until the heap is empty
```

Python Solution

class Solution:

min heap = []

while min_heap:

for i in range(rows):

for j in range(cols):

height, x, y = heappop(min_heap)

nx, ny = x + dx, y + dy

for dx, dy in directions:

6

9

10

11

12

13

14

15

16

17

18

28

29

30

31

32

```
33
                     if 0 <= nx < rows and 0 <= ny < cols and not visited[nx][ny]:</pre>
 34
                         # Calculate the possible water level difference
                         trapped_water += max(0, height - height_map[nx][ny])
 35
 36
 37
                         # Mark the neighbor as visited
                         visited[nx][ny] = True
 38
 39
                         # Push the neighbor cell onto the heap with the max height
 40
                         # to keep track of the 'water surface' level
 41
 42
                         heappush(min_heap, (max(height, height_map[nx][ny]), nx, ny))
 43
 44
             # Return the total accumulated trapped water
 45
             return trapped_water
 46
Java Solution
     import java.util.PriorityQueue;
     public class Solution {
         // Method to calculate the total trapped rainwater in the given height map
  5
         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
 10
 11
             // Initialize the min-heap with the boundary cells and mark them as visited
 12
             for (int i = 0; i < rows; ++i) {
 13
                 for (int j = 0; j < cols; ++j) {
                     if (i == 0 || i == rows - 1 || j == 0 || j == cols - 1) {
 14
 15
                         minHeap.offer(new int[]{heightMap[i][j], i, j});
 16
                         visited[i][j] = true;
 17
 18
 19
 20
 21
             int totalWater = 0; // Variable to store total trapped water
 22
             // Direction vectors (up, right, down, left)
 23
             int[] dirX = \{-1, 0, 1, 0\};
 24
             int[] dirY = {0, 1, 0, -1};
 25
 26
             // Process cells in the priority queue
 27
             while (!minHeap.isEmpty()) {
 28
                 int[] current = minHeap.poll();
 29
                 // Iterate over all four adjacent cells
 30
                 for (int k = 0; k < 4; ++k) {
 31
                     int newRow = current[1] + dirX[k], newCol = current[2] + dirY[k];
 32
 33
                     // Check bounds and visited status of the adjacent cell
 34
                     if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && !visited[newRow][newCol]) {</pre>
```

16 // Mark the boundary cells as visited and add them to the min_heap for (int i = 0; i < rows; ++i) { 17 for (int j = 0; j < cols; ++j) { 18 19 20 21

35

36

37

38

39

40

41

42

43

44

45

46

48

6

8

9

10

11

12

13

14

15

22

23

24

25

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

98

};

};

minHeap.pop();

minHeap.push(element);

let i = minHeap.length - 1;

const rows: number = heightMap.length;

for (let i = 0; i < rows; i++) {

let trappedWater: number = 0;

while (minHeap.length) {

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

for (let j = 0; j < cols; j++) {</pre>

visited[i][j] = true;

const directions: number[] = [-1, 0, 1, 0, -1];

// Check all 4 potential directions

for (let k = 0; k < 4; k++) {

// Process cells until the priority queue is empty

// Visited matrix to keep track of visited cells

// Mark the boundary cells as visited and add them to the minHeap

insertHeap([heightMap[i][j], i, j]);

// Initialize the accumulator for the trapped water to 0

const [currentHeight, row, col] = extractMin()!;

const newRow: number = row + directions[k];

const newCol: number = col + directions[k + 1];

than M+N for non-trivial maps, the overall space complexity is dominated by the vis matrix.

if (i === 0 || i === rows - 1 || j === 0 || j === cols - 1) {

// Directions array to facilitate traversal of adjacent cells (up, right, down, left)

// Helper function to insert elements in the heap

i = Math.floor((i - 1) / 2);

const insertHeap = (element: HeightAndCoordinates) => {

while (i !== 0 && compare(minHeap[Math.floor((i - 1) / 2)], minHeap[i]) > 0) {

const visited: boolean[][] = Array.from(new Array(rows), () => new Array(cols).fill(false));

[minHeap[i], minHeap[Math.floor((i - 1) / 2)]] = [minHeap[Math.floor((i - 1) / 2)], minHeap[i]];

heapify(0);

return min;

47 }

C++ Solution

public:

1 class Solution {

```
26
             // Initialize the water trapped accumulator to 0
 27
             int trapped_water = 0;
 28
 29
             // Directions array to facilitate the traversal of adjacent cells
             // Up, Right, Down, Left
 30
             const int directions [5] = \{-1, 0, 1, 0, -1\};
 31
 32
             // Process cells until there are no more cells in the priority queue
 33
 34
             while (!min_heap.empty()) {
                 auto [current_height, row, col] = min_heap.top();
 35
 36
                 min_heap.pop();
 37
 38
                 // Check all 4 possible directions
                 for (int k = 0; k < 4; ++k) {
 39
                     int new row = row + directions[k];
 40
 41
                     int new_col = col + directions[k + 1];
 42
 43
                     // 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]) {</pre>
 44
 45
                         // Update trapped water if the adjacent cell's height is less than the current cell's height
 46
                         trapped_water += max(0, current_height - height_map[new_row][new_col]);
 47
 48
                         // Mark the cell as visited
 49
                         visited[new_row][new_col] = true;
 50
 51
                         // Push the maximum height of the adjacent cell or current cell into the min_heap
 52
                         min_heap.emplace(max(height_map[new_row][new_col], current_height), new_row, new_col);
 53
 54
 55
 56
             // Return the total trapped water
 57
 58
             return trapped_water;
 59
 60
    };
 61
Typescript Solution
  1 // Define a type for the priority queue to store the height and the coordinates
  2 type HeightAndCoordinates = [number, number, number];
  4 // 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[] = [];
  8
  9
         // Comparator function for Min Heap
         const compare: (a: HeightAndCoordinates, b: HeightAndCoordinates) => number = ([heightA], [heightB]) => heightA - heightB;
 10
 11
 12
         // Helper function to heapify the minHeap
 13
         const heapify = (index: number) => {
 14
             let smallest = index;
 15
             const left = 2 * index + 1;
 16
             const right = 2 * index + 2;
 17
 18
             if (left < minHeap.length && compare(minHeap[left], minHeap[smallest]) < 0) {</pre>
 19
                 smallest = left;
 20
 21
             if (right < minHeap.length && compare(minHeap[right], minHeap[smallest]) < 0) {</pre>
 22
                 smallest = right;
 23
 24
             if (smallest !== index) {
 25
                 [minHeap[index], minHeap[smallest]] = [minHeap[smallest], minHeap[index]];
 26
                 heapify(smallest);
 27
         };
 28
 29
 30
         // Helper function to extract the top element from the heap
 31
         const extractMin = (): HeightAndCoordinates | undefined => {
 32
             if (minHeap.length === 0) return undefined;
 33
             const min = minHeap[0];
 34
             minHeap[0] = minHeap[minHeap.length - 1];
```

93 94 95 // Return total trapped water 96 return trappedWater; 97

Time and Space Complexity

operation and there are M*N cells.

81 // Check if the new cell is within bounds and has not been visited 82 if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && !visited[newRow][newCol]) {</pre> 83 // Update trapped water if the adjacent cell's height is less than the current height 84 trappedWater += Math.max(0, currentHeight - heightMap[newRow][newCol]); 85 86 // Mark the cell as visited 87 visited[newRow][newCol] = true; 88 89 // Add the maximum height of the adjacent or current cell to the minHeap insertHeap([Math.max(heightMap[newRow][newCol], currentHeight), newRow, newCol]); 90 91 92

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

The time complexity of the code is 0(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