Trapping Rain Water II

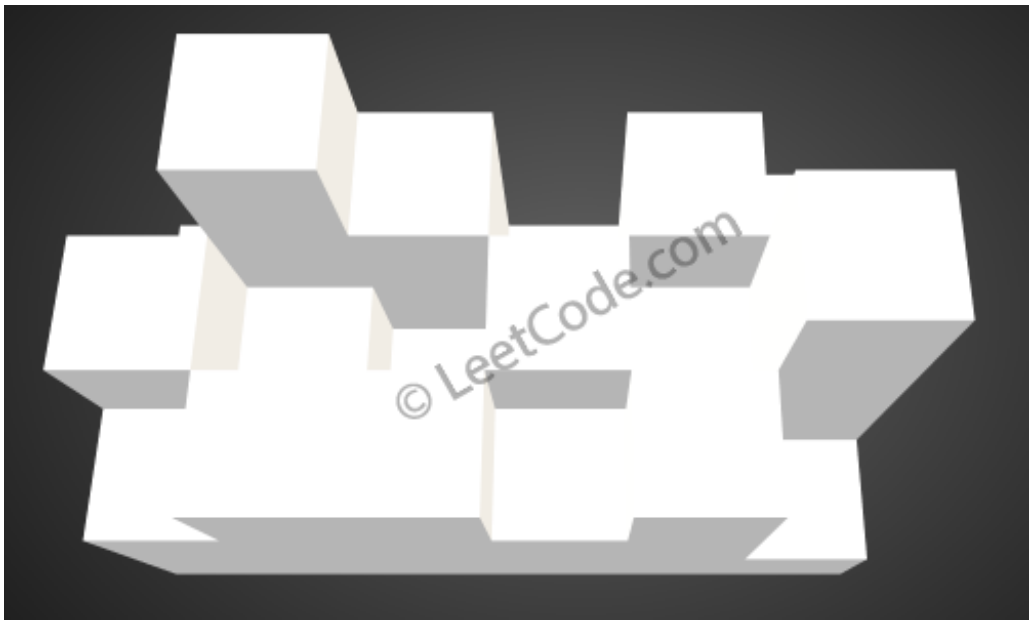
Given an `m x n` matrix of positive integers representing the height of each unit cell in a 2D elevation map, compute the volume of water it is able to trap after raining.

Note:
Both *m* and *n* are less than 110. The height of each unit cell is greater than 0 and is less than 20,000.

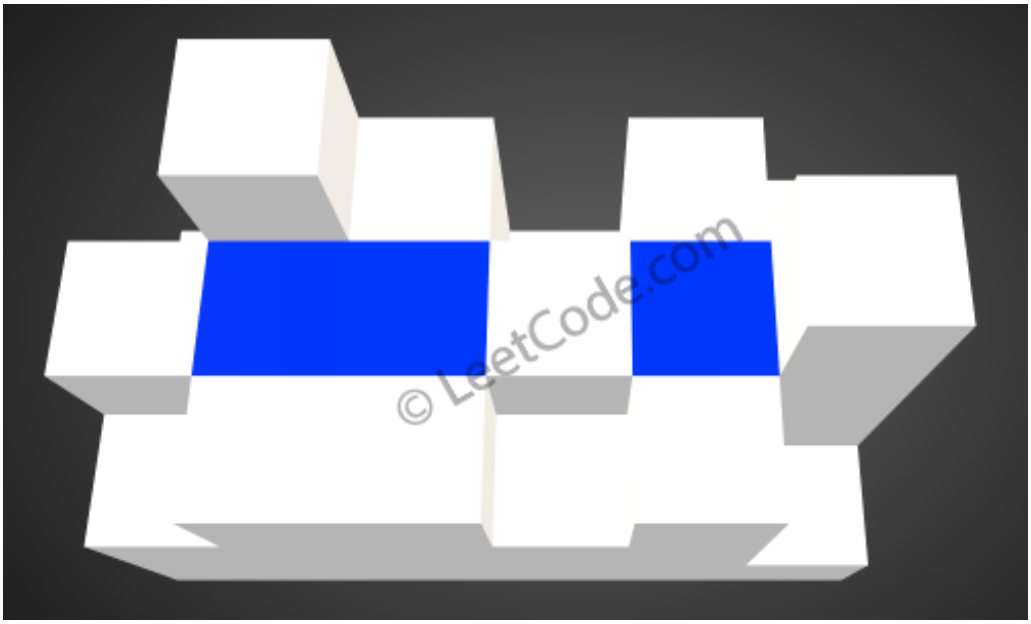
Example:

```
Given the following 3x6 height map:
[
  [1,4,3,1,3,2],
  [3,2,1,3,2,4],
  [2,3,3,2,3,1]
]

Return 4.
```



The above image represents the elevation map `[[1,4,3,1,3,2],[3,2,1,3,2,4],[2,3,3,2,3,1]]` before the rain.



After the rain, water are trapped between the blocks. The total volume of water trapped is 4.

Subscribe to see which companies asked this question

Solution 1

Source code from:
https://github.com/shawnfan/LintCode/blob/master/Java/Trapping Rain Water II.java

```
public class Solution {

    public class Cell {
        int row;
        int col;
        int height;
        public Cell(int row, int col, int height) {
            this.row = row;
            this.col = col;
            this.height = height;
        }
    }

    public int trapRainWater(int[][] heights) {
        if (heights == null || heights.length == 0 || heights[0].length == 0)
            return 0;

        PriorityQueue<Cell> queue = new PriorityQueue<>(1, new Comparator<Cell>()
{
            public int compare(Cell a, Cell b) {
                return a.height - b.height;
            }
        });

        int m = heights.length;
        int n = heights[0].length;
        boolean[][] visited = new boolean[m][n];

        // Initially, add all the Cells which are on borders to the queue.
        for (int i = 0; i < m; i++) {
            visited[i][0] = true;
            visited[i][n - 1] = true;
            queue.offer(new Cell(i, 0, heights[i][0]));
            queue.offer(new Cell(i, n - 1, heights[i][n - 1]));
        }

        for (int i = 0; i < n; i++) {
            visited[0][i] = true;
            visited[m - 1][i] = true;
            queue.offer(new Cell(0, i, heights[0][i]));
            queue.offer(new Cell(m - 1, i, heights[m - 1][i]));
        }

        // from the borders, pick the shortest cell visited and check its neighbo
rs:
        // if the neighbor is shorter, collect the water it can trap and update it
s height as its height plus the water trapped
       // add all its neighbors to the queue.
        int[][] dirs = new int[][]{{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
```

```
        int res = 0;
        while (!queue.isEmpty()) {
            Cell cell = queue.poll();
            for (int[] dir : dirs) {
                int row = cell.row + dir[0];
                int col = cell.col + dir[1];
                if (row >= 0 && row < m && col >= 0 && col < n && !visited[row][c
ol]) {
                    visited[row][col] = true;
                    res += Math.max(0, cell.height - heights[row][col]);
                    queue.offer(new Cell(row, col, Math.max(heights[row][col], ce
ll.height)));
                }
            }
        }

        return res;
    }
}
```

written by yuhaowang001 original link here

Solution 2

This problem can also be solved in a more general approach way using Dijkstra.

Construct a graph `G = (V, E)` as follows:
V = all cells plus a dummy vertex, `v`, corresponding to the outside region.
If `cell(i, j)` is adjacent to `cell(i', j')`, then add an direct edge from `(i, j)` to `(i', j')` with weight `height(i', j')`.
Add an edge with zero weight from any boundary cell to the dummy vertex `v`.

The weight of a path is defined as the weight of the heaviest edge along it. Then, for any cell `(i, j)`, the height of water it can save is equal to the weight, denoted by `dist(i, j)`, of the shortest path from `(i, j)` to `v`. (If the weight is less than or equal to `height(i, j)`, no water can be accumulated at that particular position.)

We want to compute the `dist(i, j)` for all pairs of `(i, j)`. Here, we have multiple sources and one destination, but this problem essentially can be solved using one pass of Dijkstra algorithm if we **reverse** the directions of all edges. The graph is sparse, i.e., there are O(rc) edges, resulting an `O(rc log(rc)) = O(rc max(log r, log c))` runtime and using `O(rc)` space.

Java Code:

```
public class Solution {

    int[] dx = {0, 0, 1, -1};
    int[] dy = {1, -1, 0, 0};

    List<int[]>[] g;
    int start;

    private int[] dijkstra() {
        int[] dist = new int[g.length];
        Arrays.fill(dist, Integer.MAX_VALUE / 2);
        dist[start] = 0;
        TreeSet<int[]> tree = new TreeSet<>((u, v) -> u[1] == v[1] ? u[0] - v[0]
: u[1] - v[1]);
        tree.add(new int[]{start, 0});
        while (!tree.isEmpty()) {
            int u = tree.first()[0], d = tree.pollFirst()[1];
            for (int[] e : g[u]) {
                int v = e[0], w = e[1];
                if (Math.max(d, w) < dist[v]) {
                    tree.remove(new int[]{v, dist[v]});
                    dist[v] = Math.max(d, w);
                    tree.add(new int[]{v, dist[v]});
                }
            }
        }
        return dist;
    }

    public int trapRainWater(int[][] a) {
        if (a == null || a.length == 0 || a[0].length == 0) return 0;
```

```
        int r = a.length, c = a[0].length;

        start = r * c;
        g = new List[r * c + 1];
        for (int i = 0; i < g.length; i++) g[i] = new ArrayList<>();
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++) {
                if (i == 0 || i == r - 1 || j == 0 || j == c - 1) g[start].add(ne
w int[]{i * c + j, 0});
                for (int k = 0; k < 4; k++) {
                    int x = i + dx[k], y = j + dy[k];
                    if (x >= 0 && x < r && y >= 0 && y < c) g[i * c + j].add(new
int[]{x * c + y, a[i][j]});
                }
            }

        int ans = 0;
        int[] dist = dijkstra();
        for (int i = 0; i < r; i++)
            for (int j = 0; j < c; j++) {
                int cb = dist[i * c + j];
                if (cb > a[i][j]) ans += cb - a[i][j];
            }

        return ans;
    }
}
```

written by lixx2100 original link here

Solution 3

```
/*
Basic physics:
Unlike bricks, water flows to wherever it could.
i.e we can't have the follwoing config made with water, but can do it with bricks
000
010
000
In the case above, if the "1" is built with water, that water can't stay. It needs to be spilled!

2 steps Algorithm:
1. Since we know how to trap rain water in 1d, we can just transfor this 2D problem into 2 1D problems
    we go row by row, to calculate each spot's water
    we go column by column, to calculate each spot's water

2. Then, here comes the meat,
    For every spot that gets wet, from either row or column calculation, the water can possibly spill.
    We need to check the water height aganist it's 4 neighbors.
        If the water height is taller than any one of its 4 neightbors, we need to spill the extra water.
        If we spill any water from any slot, then its 4 neightbors needs to check themselves again.
            For example, if we spill some water in the current slot b/c its bottm neighbor's height, current slot's top neighbor's height might need to be updated again.
        we keep checking until there is no water to be spilled.
*/


public class Solution {
    public int trapRainWater(int[][] heightMap) {
        /*FIRST STEP*/
        if(heightMap.length == 0) return 0;
        int[][] wetMap = new int[heightMap.length][heightMap[0].length];
        int sum = 0;
        /*row by row*/
        for(int i = 1; i < wetMap.length - 1; i++){
            wetMap[i] = calculate(heightMap[i]);
        }
        /*column by column*/
        for(int i = 1; i < heightMap[0].length - 1; i++){
            int[] col = new int[heightMap.length];
            for(int j = 0; j < heightMap.length; j++){
                col[j] = heightMap[j][i];
            }
            int[] colResult = calculate(col);
            /*update the wetMap to be the bigger value between row and col, later we can spill, don't worry*/
            for(int j = 0; j < heightMap.length; j++){
                wetMap[j][i] = Math.max(colResult[j], wetMap[j][i]);
                sum += wetMap[j][i];
            }
```

```
        }
        /*SECOND STEP*/
        boolean spillWater = true;
        int[] rowOffset = {-1,1,0,0};
        int[] colOffset = {0,0,1,-1};
        while(spillWater){
            spillWater = false;
            for(int i = 1; i < heightMap.length - 1; i++){
                for(int j = 1; j < heightMap[0].length - 1; j++){
                    /*If this slot has ever gotten wet, exammine its 4 neightbors
*/
                    if(wetMap[i][j] != 0){
                        for(int m = 0; m < 4; m++){
                            int neighborRow = i + rowOffset[m];
                            int neighborCol = j + colOffset[m];
                            int currentHeight = wetMap[i][j] + heightMap[i][j];
                            int neighborHeight = wetMap[neighborRow][neighborCol]
+
                                                                heightMap[neighbor
Row][neighborCol];
                            if(currentHeight > neighborHeight){
                                int spilledWater = currentHeight - Math.max(neigh
borHeight, heightMap[i][j]);
                                wetMap[i][j] = Math.max(0, wetMap[i][j] - spilled
Water);
                                sum -= spilledWater;
                                spillWater = true;
                            }
                        }
                    }
                }
            }
        }
        return sum;
    }

    /*Nothing interesting here, the same function for trapping water 1*/
    private int[] calculate (int[] height){
        int[] result = new int[height.length];
        Stack<Integer> s = new Stack<Integer>();
        int index = 0;
        while(index < height.length){
            if(s.isEmpty() || height[index] <= height[s.peek()]){
                s.push(index++);
            }else{
                int bottom = s.pop();
                if(s.size() != 0){
                    for(int i = s.peek() + 1; i < index; i++){
                        result[i] += (Math.min(height[s.peek()], height[index]) -
height[bottom]);
                    }
                }
            }
        }
        return result;
    }
```

```
}
```

written by DonaldTrump original link here

From Leetcoder.