medium.com

Solving Trapping Rain Water II — 2D Elevation map problem

Akanksha

6-8 minutes

Everyone is familiar with trapping water problem. Trapping rainwater problem 2 is the advanced version of the above mentioned problem. A 2D elevation map is given as a mxn matrix which represents the height of each cell. Task is to compute the volume of water that will be trapped after raining.

How to approach this problem?

Consider a 3x6 matrix :

```
[
  [12,13,1,12],
  [13,4,13,12],
  [13,8,10,12],
  [12,13,12,12],
  [13,13,13,13]]
```

1. Visualize the elevated blocks as illustrated in the figure (above : visualization of trapped water).
2. Consider cell positioned at A*(1,2)* with height 4. It is surrounded by cells with height *(Top: 13,Left: 13,Right: 13,Bottom: 8)*.Now you it can be seen it can hold 8(being the minimum of all four heights)–4

i.e. 4 volume of water in between. But visualize the 2 D elevation map and take a closer look at cell positioned at B*(2,2).* It is surrounded by cells with height *(Top: 4 ,Left: 13,Right: 10,Bottom: 13)*. Here the 4 will not be considered minimum cause its level can be increased based on the water being trapped there. And its level further depend on the lower edge(i.e cell 2,2). So, the possible minimum level for B(2,2) is 10. Which implies that cell B(2,2) can trap 10–8 i.e 2 volume of water and further the minimum for A(1,2) changes to 10 and now A(1,2) can trap 10–4 i.e. 6 volume of water. **Is this the final solution? No**, we didn't take closer look to adjacent cells of B(2,2). Visualize again. Now consider the cell (2,3) which is adjacent to B(2,2). It is surrounded with heights*(Top: 13,Left: 8,Right: 12,Bottom: 12).*Since cell with height 8 depends on the adjacent cell C(2,3) itself hence it cannot be the minimum for C. The possible minimum for cell C(2,3) is 12. Hence C can trap 12–10 i.e. 2 volume of water. Now, the minimum for B now becomes 12 as B depends on C for its minimum boundary. (Visualize, trust me. It becomes easier when you visualize).Now B(2,2) can trap 12–8 i.e. 4 volume of water. Cell A positioned at(1,2) also depends on B for its minimum boundary.Since the minimum boundary of B(2,2) changed to 12. Hence, A(1,2) can now trap 12–4 ie. 8 volumes of water.

3. Now start from the top left cell (0,0) and observe its adjacent cells. Will it be able to hold water? Of course not, because it is the boundary cell and water will simply fall off it. No matter of what height it is, the water will not be trapped as its 2 edges (I repeat, visualize)are not surrounded by any other cell .
4. Moving on to cell positioned at (0,1) , its 3 edges are covered but not the upper edge, so this cell also won’t be able to trap the

water. Now, it can be clearly observed that boundary cells won't be trapping any water. Hence, boundary cells won't be considered for trapping water. But plays an important role in deciding the minimum boundary.

It can be observed from point 2 that the minimum boundary for a cell keeps on changing based on the minimum boundary of adjacent cells. The minimum boundary cells can be tracked by starting with cells whose height is minimum. Hence , to maintain the increasing order of height of cell, priority queue is best suited for this.

Priority Queue will keep track of cell positioned at [i,j] with height h and will maintain the increasing order of heights of cells.

Now, we need a DS to store i ,j and height of cells altogether in priority queue hence priority queue of type int[] will be declared.

```
PriorityQueue<int[]> pq=new PriorityQueue((a,b)->a[0]-b[0]);
```

Why a[0]-b[0]? Because an increasing order needs to be maintained so cell with minimum height can be fetched.

Why 0? Because I'll be storing height of cell in index 0 of int[].You can store it in 1 or 2 as well.

Now, how will I be storing cell values(height, i,j) in pq?

```
//Consider storing values of cell(i,j) with height h
 pq.add(new int[] {h,i,j);
```

At every step, priority queue will poll out the cell with minimum height. For that polled out cell, all the adjacent cells will be visited .So this might lead to re-visiting of some previously visited cells, which will be superfluous. Hence, it will be best to keep track of all

the visited cells so such cells won't be visited again.

A boolean array will keep track of all the cell's visit. All the cells will be initially set to not visited and as the cells are added into priority queue, those cells will be marked visited.

```
 boolean[][] visited=new boolean[m][n];
//where m x n is order of the matrix of heights of cell
```

Let's implement the above discussed approach.

Given: MxN matrix heightMap[][]

```
public int trapRainWater(int[][] heightMap) {
        int row=heightMap.length;
        if(row==0)
                return 0;
        int col=heightMap[0].length;
                boolean[][] visited=new boolean[row][col];
        PriorityQueue<int[]> minHeap=new PriorityQueue<>
        ((a,b)->a[0]-b[0]);
/*
a[0]-b[0] for asc order of priority and 0 is bcz I'll be storing height in
index 0
and i in index1 and j in index 2
input all border cells first*/
        for(int i=0;i<row;i++)
        {
           minHeap.add(new int[] {heightMap[i][0],i,0});
           minHeap.add(new int[] {heightMap[i][col-1],i,col-1});
           visited[i][0]=true;
           visited[i][col-1]=true;
        }
```

```
            for(int j=1;j<col-1;j++)
        {
            minHeap.add(new int[] { heightMap[0][j],0,j});
            minHeap.add(new int[] { heightMap[row-1][j],row-1,j});
            visited[0][j]=true;
            visited[row-1][j]=true;
        }
        /*
now visiting all the non-visited cells
starting with the minimum boundary
why minimum?  Because cell1 depends on its adjacent cells
for min boundary
and those adjacent cells further depend on their adjacent
cells for min boundary..
and hence starting with min boundary and covering all possible
adjcaent cells
to the min boundary and keep increasing the value of min
boundary
make sure all the adjacent (L,R,T,B) cells are visited
*/
        int[][] dirs={{-1,0},{1,0},{0,-1},{0,1}};
        int curMax=0,res=0;
        while(!minHeap.isEmpty())
        {
            int[] curCell=minHeap.poll();
            curMax=Math.max(curMax,curCell[0]);
//height is stored in 0 index
            for(int[] dir: dirs)
            {
```

```
        int i=curCell[1]+dir[0];
        int j=curCell[2]+dir[1];
        if(i>=0 && i<row && j>=0 && j<col &&
        visited[i][j]==false)
        {
            if(curMax> heightMap[i][j])
                res+=curMax-heightMap[i][j];
            minHeap.add(new int[] {heightMap[i][j],i,j});
            visited[i][j]=true;
        }
                    }
}
return res;
    }
```