MEDIUM

Path With Minimum Effort

Intuition

2D array consisting of the heights ,we can move in 4 directions and we have to reach our destination using minimum effort.

Eg.

1	2	2
3	8	2
5	3	**5**

0,0 - source
2,2 - Destination

We can move in 4 directions only

So we have multiple ways to reach :

Eg.
1->2->2->2->5 : effort = max(1|0|0|3) = 3
1->3->8->2->5 : effort = max(2|5|6|3) = 6
1->3->5->3->5 : effort = max(2|2|2|2) = 2

Hence we return 0

Dijkstra can be used to solve this.

We will be assuming priority queue and distance vector

Distance :

0	inf	inf
inf	inf	inf
inf	inf	inf

Queue : [0,[0,0]]

After Traversal :

0	1	1
2	5	1
2	inf	2

Priority Queue :

[0,[0,0]]

[2,[1,0]]
[1,[0,1]]

[6,[1,1]]
[2,[1,0]]
[1,[0,2]] // we will take new effort + original effort

[6,[1,1]]
[2,[1,0]]
[1,[1,2]]

[5,[1,1]]
[6,[1,0]]
[3,[2,2]]
[2,[1,0]]

[5,[1,1]]
[6,[1,0]]
[3,[2,2]]
[2,[2,2]]

….
….
….

[3,[2,2]]
[2,[2,2]]

The answer is therefore 2

Approach

- Create a min-heap ordered priority queue
- Create a distance vector that initially contains infinity
- Insert the source element into the priority queue
- Mark the source node to have a distance 0
- Traverse until the queue becomes empty :
    - Extract the first element of the queue
    - Check if the row and the column of first element are the destination :
        - Return distance to the destination
    - Traverse for the adjacent elements :
        - Calculate the new indexes
        - Check for validity of new indexes :
            - Calculate the new effort as new effort is maximum of the absolute value of the difference between the heights or the current node distance to reach
            - Check if the new effort is less as compared to the original reaching distance :
                - Update the distance
                - Push the new effort distance and new row and new col into the queue
- Return 0 // no need to include this

Function Code

```
int MinimumEffort(vector<vector<int>>& grid) {
        // Creating a min-heap ordered priority queue

priority_queue<pair<int,pair<int,int>>,vector<pair<int,pair<int,int>>>,grea
ter<pair<int,pair<int,int>>>>pq;
        // calculating the dimensions of the grid
        int n = grid.size();
        int m = grid[0].size();
        // creating a distance vector that is initialized with all as
infinity
        vector<vector<int>> distance(n,vector<int>(m,1e9));
        // marking the source as distance 0 and pushing it to the queue
        distance[0][0] = 0;
        pq.push({0,{0,0}});
        // traversing until the queue becomes empty
        while(!pq.empty())
        {
```

```
            // extracting the first element from the queue
            auto it = pq.top();
            int difference = it.first;
            int row = it.second.first;
            int col = it.second.second;
            // popping the first element
            pq.pop();

            // checking for the condition if we are able to reach to the
last element
            if(row==n-1 && col==m-1)
            {
                return difference;
            }
            // traversing for the adjacent elements
            int dr[] = {-1,0,1,0};
            int dc[] = {0,1,0,-1};
            for(int i=0;i<4;i++)
            {
                // calculating new dimensions
                int nrow = row+dr[i];
                int ncol = col+dc[i];


                // checking for the validity of dimensions
                if(nrow<n && ncol<m && nrow>=0 && ncol>=0)
                {
                    // checking if we are able to reach them with a better
distance
                    int newEffort =
max(abs(grid[row][col]-grid[nrow][ncol]),difference);
                    if(newEffort<distance[nrow][ncol])
                    {
                        // updating the distance
                        distance[nrow][ncol] = newEffort;
                        // pushing updated distance to queue
                        pq.push({newEffort,{nrow,ncol}});
                    }
                }
            }
        }
        return 0;
    }
```

Time Complexity

O(n*m*4*log(n*m))