<https://leetcode.ca/2018-12-06-1102-Path-With-Maximum-Minimum-Value/>

Given an m x n integer matrix grid, return the maximum score of a path starting at (0, 0) and ending at (m - 1, n - 1) moving in the 4 cardinal directions.

The score of a path is the minimum value in that path.

For example, the score of the path 8 → 4 → 5 → 9 is 4.

**Example 1:**

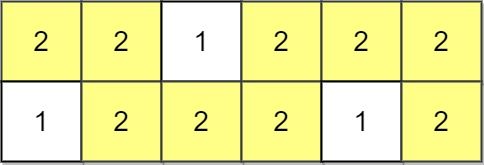


**Input:** grid = [[5,4,5],[1,2,6],[7,4,6]]

**Output:** 4

**Explanation:** The path with the maximum score is highlighted in yellow.

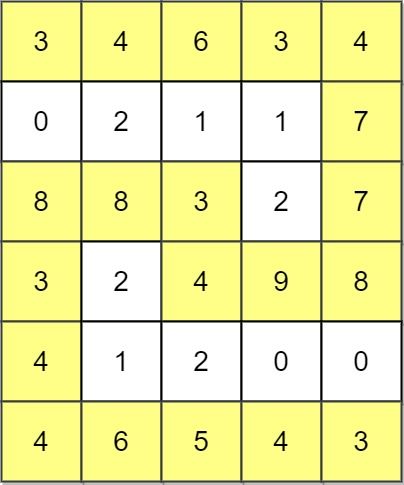
**Example 2:**



**Input:** grid = [[2,2,1,2,2,2],[1,2,2,2,1,2]]

**Output:** 2

**Example 3:**



**Input:** grid = [[3,4,6,3,4],[0,2,1,1,7],[8,8,3,2,7],[3,2,4,9,8],[4,1,2,0,0],[4,6,5,4,3]]

**Output:** 3

**Constraints:**

m == grid.length

n == grid[i].length

1 <= m, n <= 100

0 <= grid[i][j] <= 10^9

**Attempt 1: 2024-12-17**

**Solution 1: Reversed Dijkstra (10 min, similar to** [L2812.Find the Safest Path in a Grid (Ref.L778,L1102,L1631)](note://WEB787275be571505150312dacd1e3e5609)**, have to use maxPQ for Dijkstra instead of classic minPQ)**

**从 [0 , 0] 到 [m - 1, n - 1] 可以通过不同的path到达，不同的 path 有不同的分数 (当前 path 上的最小值)，而我们要找到这所有 path 的分数中最大的那一个**

**Style 1: Exactly same as L1368 and L2812**

class Solution {

    public int maximumMinimumPath(int[][] grid) {

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

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

        int m = grid.length;

        int n = grid[0].length;

        int[][] scores = new int[m][n];

        // Reversed Dijkstra algorithm initialize with all cells score

        // as min value as -1, because 0 <= grid[i][j] <= 10^9, except

        // the start cell [0, 0] has to reset as grid[0][0],

        for(int i = 0; i < m; i++) {

            Arrays.fill(scores[i], -1);

        }

        scores[0][0] = grid[0][0];

// Priority queue to process cells in descending order of path minimum value

        PriorityQueue<int[]> maxPQ = new PriorityQueue<>((a, b) -> b[2] - a[2]);

        maxPQ.offer(new int[] {0, 0, scores[0][0]});

        int result = (int) (1e9 + 1);

        while(!maxPQ.isEmpty()) {

            int[] cur = maxPQ.poll();

            int x = cur[0];

            int y = cur[1];

            int score = cur[2];

            result = Math.min(result, score);

// If we've reached the bottom-right corner, return the result

            if(x == m - 1 && y == n - 1) {

                return result;

            }

            if(score < scores[x][y]) {

                continue;

            }

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

                int new\_x = x + dx[k];

                int new\_y = y + dy[k];

                    int new\_score = grid[new\_x][new\_y];

                    // Dijkstra algorithm only update the path if a larger

                    // score is found till current cell

                    if(new\_score > scores[new\_x][new\_y]) {

                        scores[new\_x][new\_y] = new\_score;

                        maxPQ.offer(new int[] {new\_x, new\_y, new\_score});

                    }

                }

            }

        }

        return result;

    }

}

Time Complexity: O(m \* n \* log(m \* n))

Space Complexity: O(m \* n)

**Style 2: More concise**

class Solution {

public int maximumMinimumPath(int[][] grid) {

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

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

int m = grid.length;

int n = grid[0].length;

// Priority queue to process cells in descending order of path minimum value

PriorityQueue<int[]> maxPQ = new PriorityQueue<>((a, b) -> b[2] - a[2]);

// Add the starting cell (0, 0) to the priority queue

maxPQ.offer(new int[] {0, 0, grid[0][0]});

// Visited array to keep track of visited cells

boolean[][] visited = new boolean[m][n];

visited[0][0] = true;

while (!maxPQ.isEmpty()) {

int[] cur = maxPQ.poll();

int x = cur[0];

int y = cur[1];

int score = cur[2];

// If we've reached the bottom-right corner, return the result

if (x == m - 1 && y == n - 1) {

return score;

}

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

int newX = x + dx[k];

int newY = y + dy[k];

visited[newX][newY] = true;

// Push the neighbor with updated minimum value in the path

maxPQ.offer(new int[] {newX, newY, Math.min(score, grid[newX][newY])});

}

}

}

// This point should never be reached for a valid input

return -1;

}

}

Time Complexity: O(m \* n \* log(m \* n))

Space Complexity: O(m \* n)

**Refer to**

LeetCode 1102, **"Path With Maximum Minimum Value"**, requires finding a path from the top-left corner of a grid to the bottom-right corner such that the minimum value in the path is maximized. Below is a **Java solution** using a **Priority Queue (Max-Heap)** to implement a modified Dijkstra's algorithm.

**Solution Explanation**

Algorithm:

Use a max-heap to prioritize exploring paths with larger minimum values first.

Keep track of visited cells to avoid revisiting them.

Start from the top-left cell and explore all neighbors.

For each cell, calculate the minimum value encountered in the path so far, push it into the heap, and continue exploring.

Stop when reaching the bottom-right cell.

Key Insights:

The max-heap ensures that paths with larger minimum values are processed first.

Once we reach the bottom-right cell, the value is guaranteed to be the maximum possible minimum value for any valid path.

Time Complexity:

O(m⋅n⋅log⁡(m⋅n)), where mmm and nnn are the grid dimensions.

Each cell is processed once, and heap operations take O(log⁡(m⋅n)).

Space Complexity:

O(m⋅n) for the heap and visited array.

**Refer to**

[L1368.Minimum Cost to Make at Least One Valid Path in a Grid (Ref.L743,L2290)](note://WEB4b3aa9003dbf5915537540decf09b434)

[L2812.Find the Safest Path in a Grid (Ref.L778,L1102,L1631)](note://WEB787275be571505150312dacd1e3e5609)