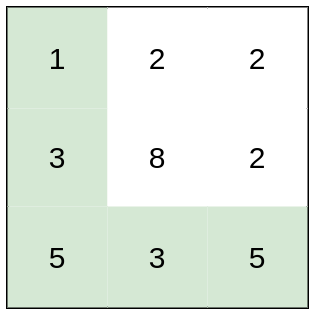
<https://leetcode.com/problems/path-with-minimum-effort/description/>

You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size rows x columns, where heights[row][col] represents the height of cell (row, col). You are situated in the top-left cell, (0, 0), and you hope to travel to the bottom-right cell, (rows-1, columns-1) (i.e., 0-indexed). You can move up, down, left, or right, and you wish to find a route that requires the minimum effort.

A route's effort is the maximum absolute difference in heights between two consecutive cells of the route.

Return the minimum effort required to travel from the top-left cell to the bottom-right cell.

**Example 1:**

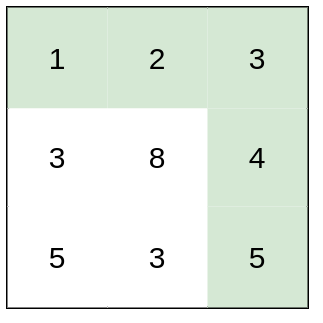


**Input:** heights = [[1,2,2],[3,8,2],[5,3,5]]

**Output:** 2

**Explanation:** The route of [1,3,5,3,5] has a maximum absolute difference of 2 in consecutive cells.This is better than the route of [1,2,2,2,5], where the maximum absolute difference is 3.

**Example 2:**

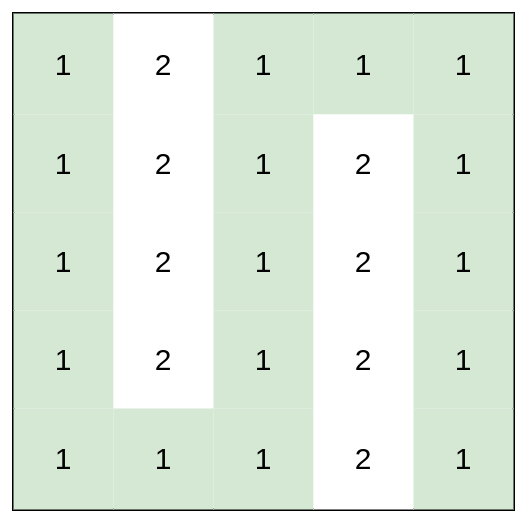


**Input:** heights = [[1,2,3],[3,8,4],[5,3,5]]

**Output:** 1

**Explanation:** The route of [1,2,3,4,5] has a maximum absolute difference of 1 in consecutive cells, which is better than route [1,3,5,3,5].

**Example 3:**



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

**Output:** 0

**Explanation:** This route does not require any effort.

**Constraints:**

rows == heights.length

columns == heights[i].length

1 <= rows, columns <= 100

1 <= heights[i][j] <= 10^6

**Attempt 1: 2024-12-18**

**Solution 1: Dijkstra (10 min, similar to** [L778.Swim in Rising Water (Ref.L1368,L1631)](note://WEB2176826d0a2ce65cdc47c7317edd5a19)**)**

**A little different than classic Dijkstra which used to find minimum accumulated distance (here is the cost):**

**Total path cost is defined as maximum absolute difference in heights between two consecutive cells of the path**

class Solution {

    public int minimumEffortPath(int[][] heights) {

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

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

        int m = heights.length;

        int n = heights[0].length;

        // Dijkstra algorithm initialize with all cells effort as max value

        // as 10^6 (because 1 <= heights[i][j] <= 10^6), maximum absolute

        // difference is 10^6 - 1, set as 10^6 at start moment, except the

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

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

        for(int i = 0; i < m; i++) {

            // Since 1 <= heights[i][j] <= 10^6, maximum

            // absolute difference is 10^6 - 1, set as 10^6

            Arrays.fill(efforts[i], 1000000);

        }

        // Start with effort = 0 (absolute difference on itself is 0)

        efforts[0][0] = 0;

        int result = 0;

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

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

        while(!minPQ.isEmpty()) {

            int[] cur = minPQ.poll();

            int x = cur[0];

            int y = cur[1];

            int effort = cur[2];

            // Refer to L778

            // Djikstra is surely going to find the most optimal minimum effort route,

            // and the route presents on 'efforts' matrix is absolutely the final route

            // start from [0, 0] and end to [m - 1, n - 1], each value on this route

            // used to store corresponding cell's minimum effort when route reach to

            // this cell, and the 'effort' define as maximum absolute difference in

            // heights between two consecutive cells of the route, now since we find

            // the minimum effort route, the maximum effort on this route (get by repeatly

            // compare each cell's effort on this path) will be treated as global minimum

            // effort required to travel from the top-left cell to the bottom-right cell,

            // since that's the maximum effort on the final designated minimum effort route

            result = Math.max(result, effort);

            // If we've reached the bottom-right cell, return the effort

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

                return result;

            }

            // Skip if encounter same cell again and cell's effort is outdated

            if(effort > efforts[x][y]) {

                continue;

            }

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

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

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

                    int new\_effort = Math.abs(heights[new\_x][new\_y] - heights[x][y]);

                    // Dijkstra algorithm only update the route if

                    // a smaller effort is found till current cell

                    if(new\_effort < efforts[new\_x][new\_y]) {

                        efforts[new\_x][new\_y] = new\_effort;

                        minPQ.offer(new int[] {new\_x, new\_y, new\_effort});

                    }

                }

            }

        }

        return result;

    }

}

Time Complexity: O(m\*n\*log⁡(m\*n)), where m and n are the number of rows and columns, respectively.

This comes from processing each cell and pushing it into the priority queue.

Space Complexity: O(m\*n) for the minEffort array and the priority queue.

**Solution 2: Binary Search + DFS (10 min, similar to** [L778.Swim in Rising Water (Ref.L1368,L1631)](note://WEB2176826d0a2ce65cdc47c7317edd5a19)**)**

class Solution {

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

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

    public int minimumEffortPath(int[][] heights) {

        int m = heights.length;

        int n = heights[0].length;

        // 1 <= heights[i][j] <= 10^6

        int lo = 0;

        int hi = 1000000;

        // Find lower boundary

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            // If current 'mid'(effort) able to reach,

            // we try to move 'hi' backward to 'mid - 1'

            // to attempt if smaller 'mid' able to reach,

            // otherwise try to move 'lo' forward to

            // 'mid + 1' to attempt if larger 'mid' able

            // to reach

            if(canReach(heights, 0, 0, m, n, mid)) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

    private boolean canReach(int[][] heights, int x, int y, int m, int n, int minEffort) {

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

        return helper(heights, x, y, x, y, m, n, minEffort, visited);

    }

    // DFS to check if we can reach (m - 1, n - 1)

    private boolean helper(int[][] heights, int x, int y, int old\_x, int old\_y, int m, int n, int minEffort, boolean[][] visited) {

        // Out of bounds, already visited, or exceeds minEffort

        if(x < 0 || x >= m || y < 0 || y >= n || visited[x][y]) {

            return false;

        }

        if(Math.abs(heights[x][y] - heights[old\_x][old\_y]) > minEffort) {

            return false;

        }

        // Check if we reached the bottom-right cell

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

            return true;

        }

        // Mark current cell as visited

        visited[x][y] = true;

        for(int k = 0; k < 4; k++) {

            int new\_x = x + dx[k];

            int new\_y = y + dy[k];

            if(helper(heights, new\_x, new\_y, x, y, m, n, minEffort, visited)) {

                return true;

            }

        }

        // No path found

        return false;

    }

}

Time: O(M \* N \* log(MAX\_HEIGHT)), where MAX\_HEIGHT = 10^6, M <= 100 is the number of rows and N <= 100 is the number of columns in the matrix.

Space: O(M \* N)

**Refer to chatGPT**

LeetCode 1631: **Path With Minimum Effort** is a graph-based problem where you are tasked with finding a path between two corners of a grid such that the maximum absolute difference in heights between adjacent cells on the path is minimized.

This problem can be solved efficiently using algorithms like **Dijkstra's algorithm** or **Binary Search with BFS/DFS**. Here's a Java solution using **Dijkstra's algorithm**:

**Solution Using Dijkstra's Algorithm**

public class Solution {

public int minimumEffortPath(int[][] heights) {

int rows = heights.length;

int cols = heights[0].length;

// Directions for traversing up, down, left, and right

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

// Priority queue to get the cell with the smallest effort

PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> Integer.compare(a[2], b[2]));

pq.offer(new int[]{0, 0, 0}); // {row, col, effort}

// To track the minimum effort to reach each cell

int[][] minEffort = new int[rows][cols];

for (int[] row : minEffort) {

Arrays.fill(row, Integer.MAX\_VALUE);

}

minEffort[0][0] = 0;

while (!pq.isEmpty()) {

int[] current = pq.poll();

int row = current[0], col = current[1], effort = current[2];

// If we reach the bottom-right corner, return the effort

if (row == rows - 1 && col == cols - 1) {

return effort;

}

// Explore the neighboring cells

for (int[] direction : directions) {

int newRow = row + direction[0];

int newCol = col + direction[1];

// Calculate the new effort for this neighboring cell

int newEffort = Math.max(effort, Math.abs(heights[newRow][newCol] - heights[row][col]));

// If this path is better, update and add to the queue

if (newEffort < minEffort[newRow][newCol]) {

minEffort[newRow][newCol] = newEffort;

pq.offer(new int[]{newRow, newCol, newEffort});

}

}

}

}

// If no path exists, return 0 (but this case doesn't occur as per the problem)

return 0;

}

}

**Explanation:**

Data Structures:

A priority queue (pq) ensures we always process the cell with the smallest effort.

A minEffort array keeps track of the minimum effort to reach each cell.

Algorithm:

Start from the top-left corner of the grid and use a priority queue to process cells.

For each cell, calculate the effort required to move to its neighbors.

**If the new effort is smaller than the previously recorded effort for that cell, update it and add the cell to the priority queue.**

The process continues until the bottom-right corner is reached.

Complexity:

Time Complexity: O((m⋅n)⋅log⁡(m⋅n)), where mmm and nnn are the number of rows and columns, respectively. This comes from processing each cell and pushing it into the priority queue.

Space Complexity: O(m⋅n) for the minEffort array and the priority queue.

This approach ensures an efficient and correct solution to the problem.

**Refer to**

<https://leetcode.com/problems/path-with-minimum-effort/solutions/909017/java-python-dijikstra-binary-search-clean-concise/>

**✔️ Solution 1: Dijikstra**

If we observe, this problem is to find the shortest path from a source cell (0, 0) to a destination cell (m-1, n-1). **Here, total path cost is defined as maximum absolute difference in heights between two consecutive cells of the path.**

Thus, we could use Dijikstra's algorithm which is used to find the shortest path in a weighted graph with a slight modification of criteria for the shortest path, which costs O(E log V), where E is number of edges E = 4\*M\*N, V is number of veritices V = M\*N

class Solution {

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

public int minimumEffortPath(int[][] heights) {

int m = heights.length, n = heights[0].length;

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

for (int i = 0; i < m; i++) Arrays.fill(dist[i], Integer.MAX\_VALUE);

PriorityQueue<int[]> minHeap = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));

minHeap.offer(new int[]{0, 0, 0}); // distance, row, col

dist[0][0] = 0;

while (!minHeap.isEmpty()) {

int[] top = minHeap.poll();

int d = top[0], r = top[1], c = top[2];

if (d > dist[r][c]) continue; // this is an outdated version -> skip it

if (r == m - 1 && c == n - 1) return d; // Reach to bottom right

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

int nr = r + DIR[i], nc = c + DIR[i + 1];

int newDist = Math.max(d, Math.abs(heights[nr][nc] - heights[r][c]));

if (dist[nr][nc] > newDist) {

dist[nr][nc] = newDist;

minHeap.offer(new int[]{dist[nr][nc], nr, nc});

}

}

}

}

return 0; // Unreachable code, Java require to return interger value.

}

}

**Complexity**

Time: O(ElogV) = O(M\*N log M\*N), where M <= 100 is the number of rows and N <= 100 is the number of columns in the matrix.

Space: O(M\*N)

**✔️ Solution 2: Binary Search + DFS**

Using binary search to choose a minimum threadshold so that we can found a route which absolute difference in heights between two consecutive cells of the route always less or equal to threadshold.

class Solution(object):

def minimumEffortPath(self, heights):

m, n = len(heights), len(heights[0])

DIR = [0, 1, 0, -1, 0]

def dfs(r, c, visited, threadshold):

if r == m-1 and c == n-1: return True # Reach destination

visited[r][c] = True

for i in range(4):

nr, nc = r+DIR[i], c+DIR[i+1]

if nr < 0 or nr == m or nc < 0 or nc == n or visited[nr][nc]: continue

if abs(heights[nr][nc]-heights[r][c]) <= threadshold and dfs(nr, nc, visited, threadshold):

return True

return False

def canReachDestination(threadshold):

visited = [[False] \* n for \_ in range(m)]

return dfs(0, 0, visited, threadshold)

left = 0

ans = right = 10\*\*6

while left <= right:

mid = left + (right-left) // 2

if canReachDestination(mid):

right = mid - 1 # Try to find better result on the left side

ans = mid

else:

left = mid + 1

return ans

**Complexity**

Time: O(M \* N \* log(MAX\_HEIGHT)), where MAX\_HEIGHT = 10^6, M <= 100 is the number of rows and N <= 100 is the number of columns in the matrix.

Space: O(M \* N)

**Refer to**

[L778.Swim in Rising Water (Ref.L1368,L1631)](note://WEB2176826d0a2ce65cdc47c7317edd5a19)

[L1102.Path With Maximum Minimum Value (Ref.L1368)](note://WEBf4a6c0c7ead8e64acb71fe2f9814cb8f)

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