<https://leetcode.com/problems/find-the-safest-path-in-a-grid/description/>

You are given a 0-indexed 2D matrix grid of size n x n, where (r, c) represents:

A cell containing a thief if grid[r][c] = 1

An empty cell if grid[r][c] = 0

You are initially positioned at cell (0, 0). In one move, you can move to any adjacent cell in the grid, including cells containing thieves.

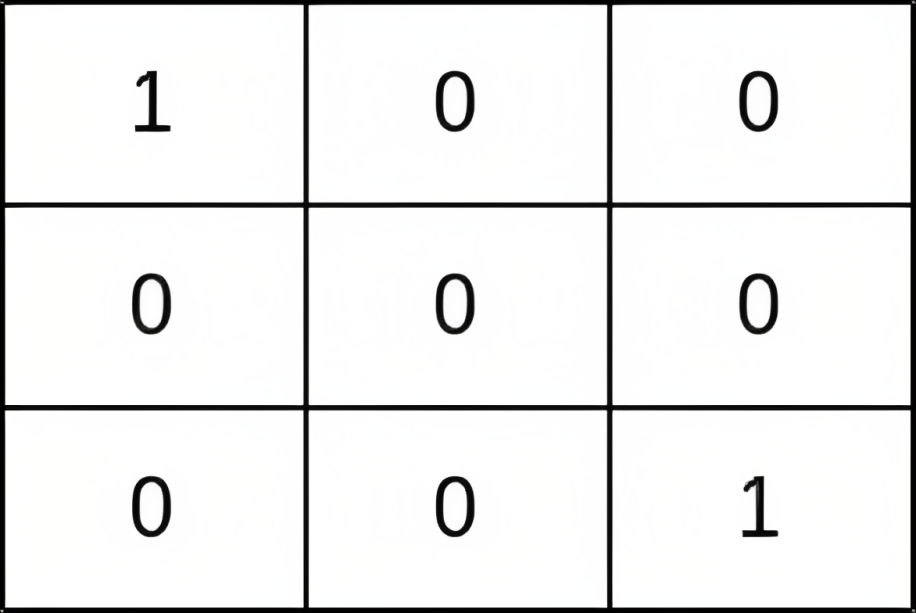
The **safeness factor** of a path on the grid is defined as the minimum manhattan distance from any cell in the path to any thief in the grid.

Return the **maximum safeness factor** of all paths leading to cell (n - 1, n - 1).

An adjacent cell of cell (r, c), is one of the cells (r, c + 1), (r, c - 1), (r + 1, c) and (r - 1, c) if it exists.

The **Manhattan distance** between two cells (a, b) and (x, y) is equal to |a - x| + |b - y|, where |val| denotes the absolute value of val.

**Example 1:**

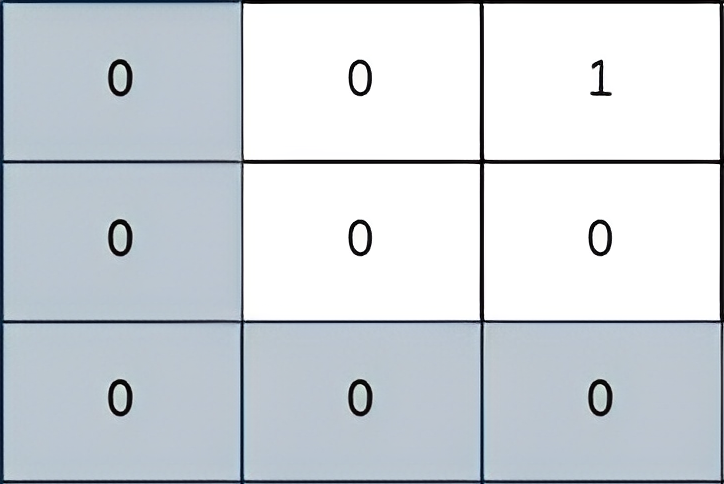


**Input:** grid = [[1,0,0],[0,0,0],[0,0,1]]

**Output:** 0

**Explanation:** All paths from (0, 0) to (n - 1, n - 1) go through the thieves in cells (0, 0) and (n - 1, n - 1).

**Example 2:**

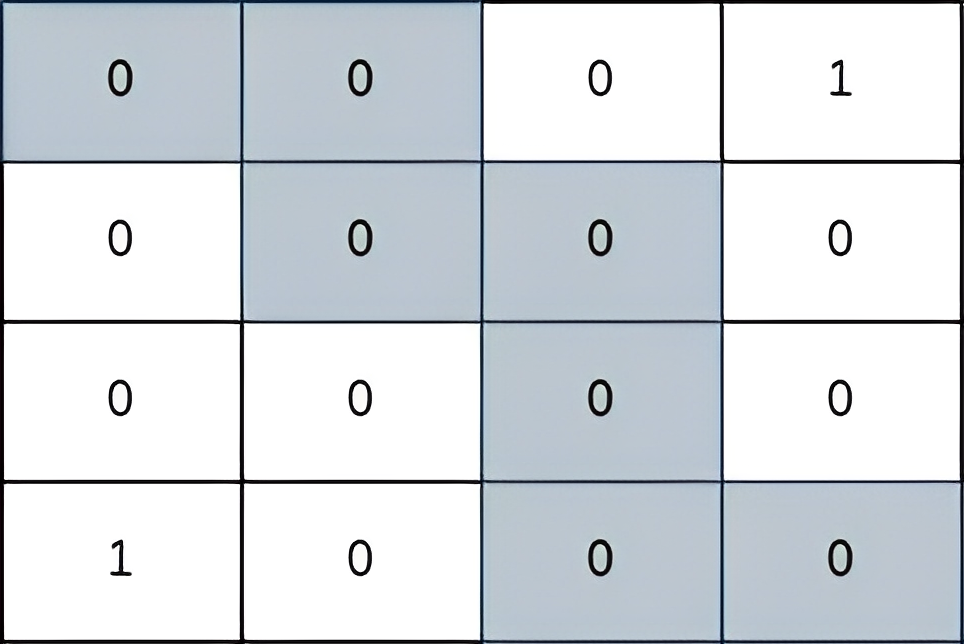


**Input:** grid = [[0,0,1],[0,0,0],[0,0,0]]

**Output:** 2

**Explanation:** The path depicted in the picture above has a safeness factor of 2 since:- The closest cell of the path to the thief at cell (0, 2) is cell (0, 0). The distance between them is | 0 - 0 | + | 0 - 2 | = 2.It can be shown that there are no other paths with a higher safeness factor.

**Example 3:**



**Input:** grid = [[0,0,0,1],[0,0,0,0],[0,0,0,0],[1,0,0,0]]

**Output:** 2

**Explanation:** The path depicted in the picture above has a safeness factor of 2 since:- The closest cell of the path to the thief at cell (0, 3) is cell (1, 2). The distance between them is | 0 - 1 | + | 3 - 2 | = 2.- The closest cell of the path to the thief at cell (3, 0) is cell (3, 2). The distance between them is | 3 - 3 | + | 0 - 2 | = 2.It can be shown that there are no other paths with a higher safeness factor.

**Constraints:**

1 <= grid.length == n <= 400

grid[i].length == n

grid[i][j] is either 0 or 1.

There is at least one thief in the grid.

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

**Solution 1: Multi-source BFS + Reversed Djikstra (180 min)**

class Solution {

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

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

    public int maximumSafenessFactor(List<List<Integer>> grid) {

        int n = grid.size();

        int[][] distToThief = findDistToThief(grid, n);

        // Find minimum safeness factor on maximum safeness factor prioritized

        // path by using reversed Dijkstra(maxPQ)

        // The key idea here is to use Dijkstra's single source shortest path

        // algorithm to find the optimal path from the source cell [0, 0] to

        // the destination cell [n-1, n-1]. However, since each cell in the

        // grid already contains its safeness factor, we need to modify

        // Dijkstra's algorithm to find the path with the maximum safeness

        // factor. In our modified Dijkstra's algorithm, we can greedily

        // prioritize cells with a higher safeness factor to append to our

        // path. The safeness factor of the path would be the minimum of the

        // safeness values encountered in that path so far. Once we reach the

        // destination cell, the safeness factor of the path would represent

        // the required maximum safeness factor

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

        // Reversed Dijkstra algorithm initialize with all cells safeness

        // factors as min value as -1, except the start cell [0, 0] has

        // to reset as distToThief[0][0]

        // Note: we have to set as -1 to guarantee all cells has safeness factor

        // as 0 on 'distToThief' matrix reflect properly, test out by:

        // Input grid = [[0,1],[1,1]], Output = 1, Expected = 0

        int[][] safenessFactors = new int[n][n];

        for(int i = 0; i < n; i++) {

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

        }

        safenessFactors[0][0] = distToThief[0][0];

        // {x, y, maximum\_safeness\_till\_now}

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

        // Initialize result as potential maximum manhattan distance + 1

        // as 2 \* n - 1 + 1 = 2 \* n

        int result = 2 \* n;

        while(!maxPQ.isEmpty()) {

            int[] cur = maxPQ.poll();

            int x = cur[0];

            int y = cur[1];

            int safenessFactor = cur[2];

            // Find minimum safeness factor on maximum safeness factor prioritized path

            result = Math.min(result, safenessFactor);

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

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

                return result;

            }

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

            if(safenessFactor < safenessFactors[x][y]) {

                continue;

            }

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

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

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

                    int new\_safenessFactor = distToThief[new\_x][new\_y];

                    // Dijkstra algorithm only update the path if a larger

                    // safeness factor is found till current cell

                    if(new\_safenessFactor > safenessFactors[new\_x][new\_y]) {

                        safenessFactors[new\_x][new\_y] = new\_safenessFactor;

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

                    }

                }

            }

        }

        return result;

    }

    // Multi-source BFS to compute the safeness factor for each cell in

    // the grid, use multi-source because the grid may contains more

    // than one thief cell, and we have to calculate the safeness factor

    // as level by level expansion for each thief cell

    // And we don't need explicitly declare 'visited' array, instead we

    // update non-thief cell's value from -1 to actual manhattan distance

    // after visiting, and use condition 'if cell value is -1' to filter

    // out non-visited cells

    private int[][] findDistToThief(List<List<Integer>> grid, int n) {

        int[][] distances = new int[n][n];

        Queue<int[]> multiSourceQueue = new LinkedList<>();

        for(int i = 0; i < n; i++) {

            for(int j = 0; j < n; j++) {

                if(grid.get(i).get(j) == 1) {

                    // Change thief cells value from 1 to 0, then treat thief

                    // cell's new value 0 as initial actual manhattan distance

                    // during BFS level by level expansion

                    distances[i][j] = 0;

                    // Push all thief cells onto BFS queue as multiple beginning source

                    multiSourceQueue.offer(new int[] {i, j});

                } else {

                    // Change non-thief cells value from 0 to -1 to help identify

                    // non-visited non-thief cells without using 'visited' array

                    distances[i][j] = -1;

                }

            }

        }

        // If reverse the initial setup for non-thief cells as -1

        // and thief cells as 0 to non-thief cells as 0 and thief

        // cells as -1, we will need separate variable 'distance'

        // to track each cell's actual manhattan distance, because

        // if thief cells as BFS beginning cell, if assign its value

        // as 0, it can be treated as beginning distance to other

        // non-thief cells, but if reverse the setup, attribute gone

        while(!multiSourceQueue.isEmpty()) {

            int size = multiSourceQueue.size();

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

                int[] cur = multiSourceQueue.poll();

                int x = cur[0];

                int y = cur[1];

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

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

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

                    // If a cell value is -1 means its a non-thief cell and not visited before

                        distances[new\_x][new\_y] = distances[x][y] + 1;

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

                    }

                }

            }

        }

        return distances;

    }

}

Time Complexity: O(n^2 \* logn)

Space Complexity: O(n^2)

**Solution 2: Multi-source BFS + Binary Search (180 min)**

class Solution {

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

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

    public int maximumSafenessFactor(List<List<Integer>> grid) {

        int n = grid.size();

        int[][] distToThief = findDistToThief(grid, n);

        int lo = 0;

        int hi = 0;

        for(int i = 0; i < n; i++) {

            for(int j = 0; j < n; j++) {

                hi = Math.max(hi, distToThief[i][j]);

            }

        }

        // Early terminate should happen for check on if [0, 0] or

        // [n - 1, n - 1] reachable, if no such check, test case will

        // fail as below:

        // grid = [[1,1,1],[0,1,1],[0,0,0]], Output = 1, Expected = 0

        // Pre-check: If start or end cell is unsafe, return 0

        //if(distToThief[0][0] == 0 || distToThief[n - 1][n - 1] == 0) {

        //    return 0;

        //}

        // Find upper boundary

        while(lo <= hi) {

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

            // To find the maximum safeness factor of all paths

            // leading to cell (n - 1, n - 1), if current safeness

            // factor 'mid' can reach, we move 'lo' forward to

            // 'mid + 1' to attempt larger safeness factor, otherwise

            // we move 'hi' backward to 'mid - 1' to attempt smaller

            // safeness factor

            if(canReach(distToThief, n, mid)) {

                lo = mid + 1;

            } else {

                hi = mid - 1;

            }

        }

        return lo - 1;

    }

    // Check if a path exists with given minimum safeness value

    // The return 'true' condition is using BFS traversal able to find a path

    // from [0, 0] to [n - 1, n - 1], during process only cell has no smaller

    // safeness factor than given 'minSafenessFactor' can consider as part of

    // the path, and if run out of cells(all cells visited) then not able to

    // reach with given 'minSafenessFactor', then return false

    private boolean canReach(int[][] distToThief, int n, int minSafenessFactor) {

        // Check if the source and destination cells satisfy minimum safeness factor,

        // if no such check, test case will fail as below:

        // grid = [[1,1,1],[0,1,1],[0,0,0]], Output = 1, Expected = 0

        if(distToThief[0][0] < minSafenessFactor

            || distToThief[n - 1][n - 1] < minSafenessFactor) {

            return false;

        }

        // We have to separately use 'visited' matrix instead of

        // modify original 'distToThief' matrix since if the value

        // changes on matrix, it will impact inequality for further

        // processing, e.g distToThief[new\_x][new\_y] >= minSafenessFactor

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

        Queue<int[]> queue = new LinkedList<>();

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

        visited[0][0] = true;

        while(!queue.isEmpty()) {

            int[] cur = queue.poll();

            int x = cur[0];

            int y = cur[1];

            // Valid path found

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

                return true;

            }

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

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

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

                // Only in boundary, not visited and has no smaller

                // safeness factor than given 'minSafenessFactor' can

                // consider as part of the path

                    && !visited[new\_x][new\_y]

                    && distToThief[new\_x][new\_y] >= minSafenessFactor) {

                    visited[new\_x][new\_y] = true;

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

                }

            }

        }

        // Exhaust cells but not able to reach [n - 1, n - 1]

        return false;

    }

    // Multi-source BFS to compute the safeness factor for each cell in

    // the grid, use multi-source because the grid may contains more

    // than one thief cell, and we have to calculate the safeness factor

    // as level by level expansion for each thief cell

    // And we don't need explicitly declare 'visited' array, instead we

    // update non-thief cell's value from -1 to actual manhattan distance

    // after visiting, and use condition 'if cell value is -1' to filter

    // out non-visited cells

    private int[][] findDistToThief(List<List<Integer>> grid, int n) {

        int[][] distances = new int[n][n];

        Queue<int[]> multiSourceQueue = new LinkedList<>();

        for(int i = 0; i < n; i++) {

            for(int j = 0; j < n; j++) {

                if(grid.get(i).get(j) == 1) {

                    // Change thief cells value from 1 to 0, then treat thief

                    // cell's new value 0 as initial actual manhattan distance

                    // during BFS level by level expansion

                    distances[i][j] = 0;

                    // Push all thief cells onto BFS queue as multiple beginning source

                    multiSourceQueue.offer(new int[] {i, j});

                } else {

                    // Change non-thief cells value from 0 to -1 to help identify

                    // non-visited non-thief cells without using 'visited' array

                    distances[i][j] = -1;

                }

            }

        }

        // If reverse the initial setup for non-thief cells as -1

        // and thief cells as 0 to non-thief cells as 0 and thief

        // cells as -1, we will need separate variable 'distance'

        // to track each cell's actual manhattan distance, because

        // if thief cells as BFS beginning cell, if assign its value

        // as 0, it can be treated as beginning distance to other

        // non-thief cells, but if reverse the setup, attribute gone

        while(!multiSourceQueue.isEmpty()) {

            int size = multiSourceQueue.size();

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

                int[] cur = multiSourceQueue.poll();

                int x = cur[0];

                int y = cur[1];

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

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

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

                    // If a cell value is -1 means its a non-thief cell and not visited before

                        && distances[new\_x][new\_y] == -1) {

                        distances[new\_x][new\_y] = distances[x][y] + 1;

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

                    }

                }

            }

        }

        return distances;

    }

}

Time Complexity: O(n^2 \* logn)

Space Complexity: O(n^2)

**When calculate each cell's safeness factor, it uses multi-source BFS, we understand it because we may have multiple thief cells which consider as multiple beginning cells for BFS, but how we guarantee no conflict calculation of safeness factor for one cell if run multiple times BFS from different beginning cell ?**

**Refer to chatGPT**

The BFS guarantees that safeness factors are calculated in the correct order (shortest distance first) and that each cell is processed exactly once. This ensures no conflicts in the safeness factor calculation, even with multiple thief cells.

If a thief at a different location could theoretically result in a different safeness factor for the same cell, that update will never occur because the BFS guarantees the shortest distance (minimum safeness factor) is processed first.

For below example, we have two thief cells as two sources for multi-source BFS, one at [0, 4], one at [4, 0], if we want to calculate the safe factor (minimum manhattan distance) for 'x' at [1, 3], we won't have conflict, and the safe factor result for 'x' at [1, 3] must come from thief cell [0, 4] and not from [4, 0], because even in multi-source BFS, we still strictly follow level by level traversal, for level 1, [0, 4] can reach [0, 3] and [1, 4], for level 2, [0, 4] can reach [0, 2], [1, 3] and [2, 4] which includes 'x' at [1, 3], but on the other hand about [4, 0], for level 1, [4, 0] can only reach [3, 0] and [4, 1], for level 2, [4, 0] can reach [1, 0], [2, 1] and [3, 2], ... etc. now multi-source BFS done for level 2 for two thief cells, actually, [4, 0] can only reach [1, 3] in level 5, which is surely getting a larger safe factor, but it won't allowed since in level 2 when [0, 4] reach 'x' at [1, 3], we will mark it as visited, and it will guarantee that's the smallest safe factor as definition from the problem

0 0 0 0 1

0 0 0 x 0

0 0 0 0 0

1 0 0 0 0

**Refumulated by chatGPT as below**

For the given example, where two thief cells are sources for the multi-source BFS—one at [0, 4] and the other at [4, 0]—if we want to calculate the safe factor (minimum Manhattan distance) for the cell 'x' at [1, 3], there will be **no conflict**. This is because multi-source BFS processes all cells level by level, ensuring that a cell's safe factor is determined the first time it is visited, which is guaranteed to correspond to the smallest Manhattan distance.

**Step-by-Step Explanation:**

Thief **[0, 4]:**

Level 1: This thief reaches [0, 3] and [1, 4].

Level 2: This thief reaches [0, 2], [1, 3] (which is 'x'), and [2, 4].

At this point (Level 2), the safe factor of 'x' is finalized as 2 (the Manhattan distance from [0, 4]).

Thief **[4, 0]:**

Level 1: This thief reaches [3, 0] and [4, 1].

Level 2: This thief reaches [2, 1], [3, 2], and [4, 2].

Level 3: This thief expands further but still cannot reach [1, 3] (which is 'x').

Level 5: Only at this point does thief [4, 0] theoretically reach [1, 3], but by this time, 'x' has already been processed and marked as visited in Level 2 (when thief [0, 4] reached it).

Marking Cells as Visited:

In multi-source BFS, cells are marked as visited as soon as they are processed for the first time. This guarantees that when [1, 3] is visited in Level 2 by the thief at [0, 4], it is finalized with a safe factor of 2.

When thief [4, 0] eventually reaches [1, 3] at Level 5, it cannot update the safe factor because the cell is already visited.

Conclusion:

The safe factor for 'x' at [1, 3] is determined by the nearest thief, which is [0, 4].

Even though the thief at [4, 0] could theoretically reach 'x' later, its contribution is irrelevant because BFS ensures that each cell is processed **exactly once**, based on the shortest distance.

For the example grid, where thief cells are located at [0, 4] and [4, 0], the safe factor (minimum Manhattan distance) for the cell 'x' at [1, 3] must come from the thief cell at [0, 4]. This is because multi-source BFS processes all cells strictly level by level, ensuring that a cell is finalized with the smallest possible safe factor (shortest Manhattan distance) the first time it is visited. In this case, thief [0, 4] reaches [1, 3] at Level 2, while thief [4, 0] only reaches [1, 3] at Level 5. Since [1, 3] is marked as visited after Level 2, it is not updated further, guaranteeing the correct safe factor is computed.

**Refer to**

<https://leetcode.com/problems/find-the-safest-path-in-a-grid/editorial/>

**Overview**

We are given a grid representing a city layout where some cells contain thieves and others are empty, and we need to find the maximum safeness factor of all paths from the top-left corner to the bottom-right corner. The safeness factor of a path is defined as the minimum Manhattan distance from any cell in the path to any thief in the grid.

**Key Observations:**

Manhattan distance between two cells is the sum of the absolute differences of their row and column indices.

All the cells in the grid contain either 0 or 1, representing empty cells and cells containing thieves respectively.

You start from the top-left corner (0, 0) and can move to adjacent cells in any of the four directions.

The maximum level of safety one can achieve while traversing from the starting point to the destination is by ensuring the least proximity to any cell containing a thief.

**Approach 1: Breadth-First Search + Binary Search**

**Intuition**

Since we need to find the safeness factor of a path from the source to the destination, the initial intuition to solve this problem is that we should first find the safeness factors of the cells in the path. The path can span across the entire grid, so we need to find the safeness factors for all the cells in the grid.

One approach to find the safeness factors of the cells would be to iterate over each cell in the grid and find its distance from all the thieves in the grid. We can then pick the smallest distance as the safeness factor for that cell.

However, this brute force approach would have a time complexity of O(n^4), which would not satisfy the constraints of the problem. Therefore, a more optimized approach is needed.

To optimize the solution, we can leverage the properties of a multi-source breadth-first Search (BFS). Instead of finding the distance of each cell from all the thieves, we can do the opposite: find the distance of all the thieves from each cell.

Note: A multi-source breadth-first search is a BFS where multiple starting nodes are explored simultaneously. This is an efficient method to find the shortest distances from any of the starting nodes to all reachable nodes in the graph. You can refer to this excellent problem to gain some practice on multi-source BFS.

The intuition for this can be,

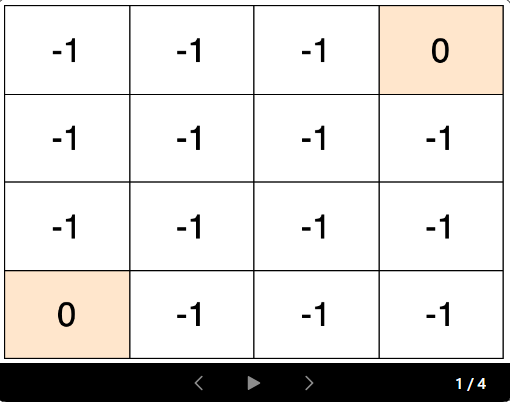
We start by adding all the thief coordinates to a queue as the initial points of exploration.

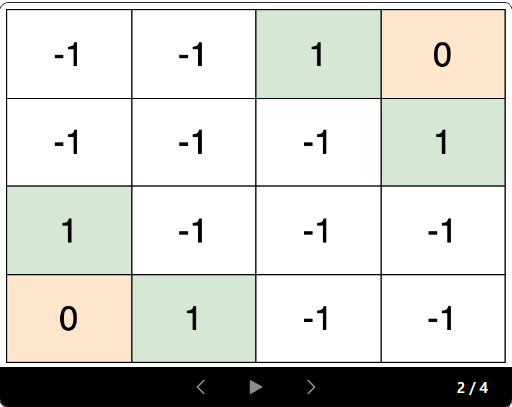
We then explore the neighboring cells (up, down, left, and right) from all the thieves in one iteration, like ripples spreading outwards from each thief.

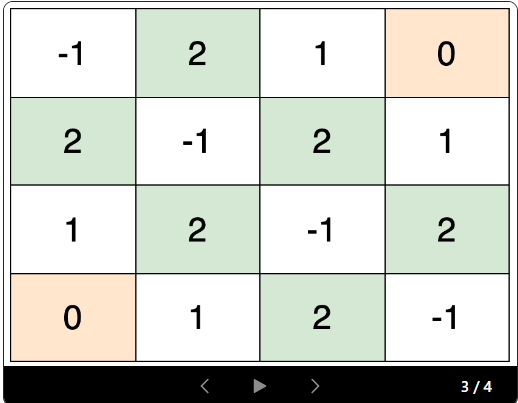
As we visit each cell, we mark it with the minimum distance from the nearest thief. This is because the first time a cell is visited, it means that the current thief is the closest one to that cell.

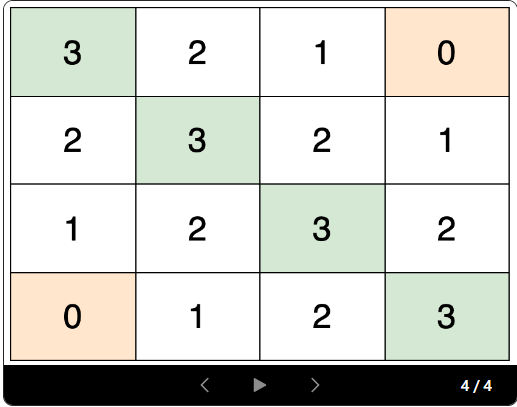
We continue the BFS traversal until all the cells in the grid are marked with their corresponding safeness values.

The following slideshow demonstrates how the BFS gradually populates the grid with its minimum distances from a thief.



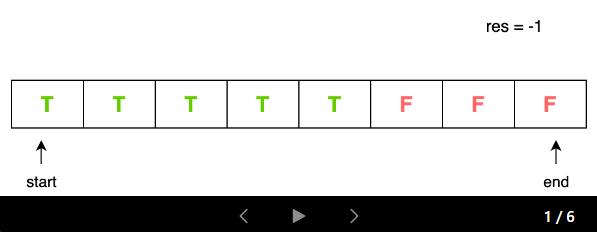


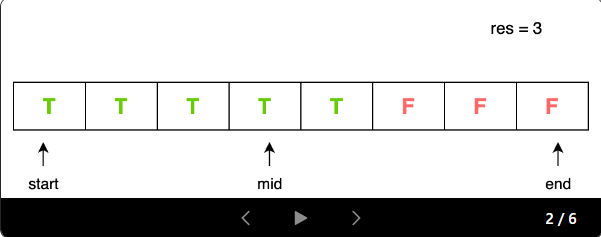


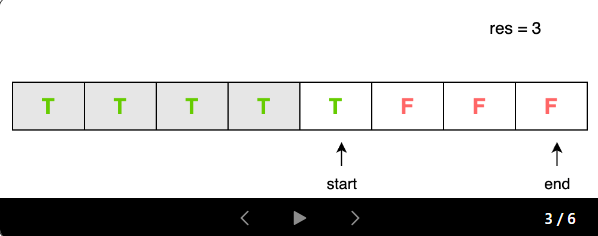


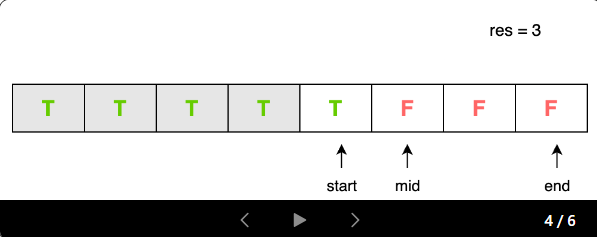
Now that we have the safeness factor of each cell, we need to find the maximum safeness factor for which a path exists from the source cell to the destination cell. This implies that for all safeness values greater than it, no path exists, and at least one path exists for all values less than it. We can visualize these safeness factors as a monotonic sequence on a number line. The values that satisfy the constraints of the problem will be a contiguous series. These will be followed by a series of values that do not satisfy the constraints. We will name this breakpoint the inflection point.

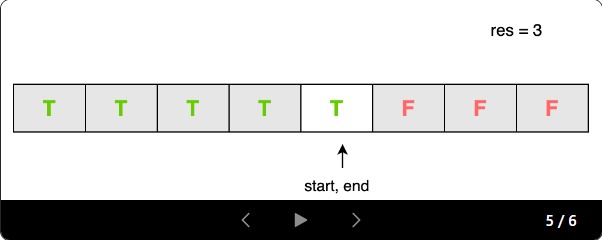
The following slideshow visualizes how we iteratively converge to the location of the inflection point using binary search.

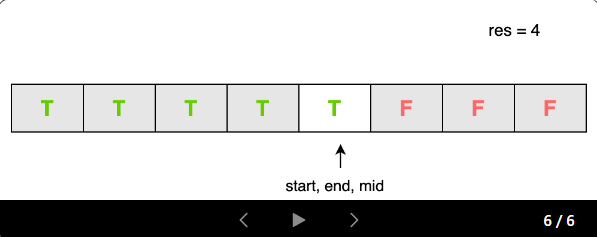












During the binary search, to determine if a safeness value meets the problem constraints, we employ another breadth-first search (BFS) traversal on the grid. The traversal attempts to find a path where every cell in the path satisfies this minimum safeness value. If such a path is found, it indicates that the given safeness value is a valid solution to the problem.

Thus, to find the maximum safeness factor, we can use binary search to efficiently locate the inflection point in this monotonic sequence. The last "True" value at the inflection point will be the maximum safeness factor for which a path exists.

In summary, the final solution involves two key steps:

Perform a breadth-first search to compute the safeness factor for each cell, leveraging the fact that the first time a cell is visited, it represents the minimum distance from the nearest thief.

Apply binary search to find the maximum safeness factor for which a path exists from the source to the destination cell.

This approach is more efficient than the initial brute-force solution, as it avoids the need to calculate the distance of each cell from all the thieves. Instead, it focuses on finding the distance of each cell from all the thieves, which can be done more optimally manner using BFS.

**Algorithm**

Initialize dir to store directions for moving to neighboring cells: right, left, down, up.

Define isValidCell method to check if a given cell is valid within the grid.

Define isValidSafeness method to check if a path exists with a minimum safeness value.

**Implementation**

class Solution {

// Directions for moving to neighboring cells: right, left, down, up

final int[][] dir = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

public int maximumSafenessFactor(List<List<Integer>> grid) {

int n = grid.size();

int[][] mat = new int[n][n];

Queue<int[]> multiSourceQueue = new LinkedList<>();

// To make modifications and navigation easier, the grid is converted into a 2-d array.

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

for (int j = 0; j < n; j++) {

if (grid.get(i).get(j) == 1) {

// Push thief coordinates to the queue

multiSourceQueue.add(new int[]{i, j});

// Mark thief cell with 0

mat[i][j] = 0;

} else {

// Mark empty cell with -1

mat[i][j] = -1;

}

}

}

// Calculate safeness factor for each cell using BFS

while (!multiSourceQueue.isEmpty()) {

int size = multiSourceQueue.size();

while (size-- > 0) {

int[] curr = multiSourceQueue.poll();

// Check neighboring cells

for (int[] d : dir) {

int di = curr[0] + d[0];

int dj = curr[1] + d[1];

int val = mat[curr[0]][curr[1]];

// Check if the neighboring cell is valid and unvisited

if (isValidCell(mat, di, dj) && mat[di][dj] == -1) {

// Update safeness factor and push to the queue

mat[di][dj] = val + 1;

multiSourceQueue.add(new int[]{di, dj});

}

}

}

}

// Binary search for maximum safeness factor

int start = 0;

int end = 0;

int res = -1;

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

for (int j = 0; j < n; j++) {

// Set end as the maximum safeness factor possible

end = Math.max(end, mat[i][j]);

}

}

while (start <= end) {

int mid = start + (end - start) / 2;

if (isValidSafeness(mat, mid)) {

// Store valid safeness and search for larger ones

res = mid;

start = mid + 1;

} else {

end = mid - 1;

}

}

return res;

}

// Check if a path exists with given minimum safeness value

private boolean isValidSafeness(int[][] grid, int minSafeness) {

int n = grid.length;

// Check if the source and destination cells satisfy minimum safeness

if (grid[0][0] < minSafeness || grid[n - 1][n - 1] < minSafeness) {

return false;

}

Queue<int[]> traversalQueue = new LinkedList<>();

traversalQueue.add(new int[]{0, 0});

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

visited[0][0] = true;

// Breadth-first search to find a valid path

while (!traversalQueue.isEmpty()) {

int[] curr = traversalQueue.poll();

if (curr[0] == n - 1 && curr[1] == n - 1) {

return true; // Valid path found

}

// Check neighboring cells

for (int[] d : dir) {

int di = curr[0] + d[0];

int dj = curr[1] + d[1];

// Check if the neighboring cell is valid, unvisited and satisfying minimum safeness

if (isValidCell(grid, di, dj) && !visited[di][dj] && grid[di][dj] >= minSafeness) {

visited[di][dj] = true;

traversalQueue.add(new int[]{di, dj});

}

}

}

return false; // No valid path found

}

// Check if a given cell lies within the grid

private boolean isValidCell(int[][] mat, int i, int j) {

int n = mat.length;

}

}

**Complexity Analysis**

Let *n*⋅*n* be the size of the matrix.

Time complexity: *O*(*n^*2⋅log*n*).

The time complexity for the initial BFS is *O*(*n^*2), as each cell in the *n*⋅*n* grid is visited once during the traversal.

The binary search occurs in the range [0, maximum safeness factor possible], where the maximum safeness factor possible is 2⋅*n*. The time complexity of the binary search is *O*(log(2⋅*n*)), which is equivalent to *O*(log*n*).

For each iteration of the binary search, a breadth-first Search is conducted to verify validity, which has a time complexity of *O*(*n^*2). Thus, the total time complexity of the binary search portion is *O*(*n^*2⋅log*n*).

The total time complexity is the sum of the time complexities of the two parts: *O*(*n^*2)+*O*(*n^*2⋅log*n*). This can be simplified to *O*(*n^*2⋅log*n*).

Space complexity: *O*(*n^*2).

The data structure used in the algorithm is a queue, which takes linear space. Since the total number of cells in the grid is *n*2, the space complexity is *O*(*n^*2).

**Approach 2: BFS + Greedy**

**Intuition**

In the previous approach, we used a binary search strategy to find the maximum safeness factor for which a path exists from the source to the destination. While this was an efficient solution, the intuition behind this approach is to directly find the optimal path from the source to the destination by leveraging Dijkstra's algorithm.

Similar to the previous approach, we first need to populate the grid with the safeness values for each cell. The algorithm to achieve this is the same as before, using the breadth-first Search (BFS) technique to compute the distance of each cell from the nearest thief.

The key idea here is to use Dijkstra's single source shortest path algorithm to find the optimal path from the source cell [0, 0] to the destination cell [n-1, n-1]. However, since each cell in the grid already contains its safeness factor, we need to modify Dijkstra's algorithm to find the path with the maximum safeness factor. In our modified Dijkstra's algorithm, we can greedily prioritize cells with a higher safeness factor to append to our path. The safeness factor of the path would be the minimum of the safeness values encountered in that path so far. Once we reach the destination cell, the safeness factor of the path would represent the required maximum safeness factor.

The modified Dijkstra's algorithm works as follows:

We start with the source cell [0, 0] in a priority queue, where the priority is based on the highest safeness factor encountered in the path so far.

For efficiency, cells we've explored are marked as -1 in the grid itself.

If the current cell is the destination [n-1, n-1], the traversal is over, and we return the maximum safeness factor encountered so far.

If the current cell is not the destination, we explore the valid adjacent cells. A cell is considered valid if it is within the grid boundaries and not visited yet (not -1).

For each valid neighbor, we calculate the potential safeness factor considering the current path's safeness and the new cell's distance to thieves. The minimum of these two values becomes the new safeness for the path with the addition of the neighbor.

We add the valid neighbors to the priority queue, prioritizing them based on their safeness factor.

We continue the exploration until we reach the destination cell.

The key advantage of this approach is that it directly finds the optimal path from the source to the destination instead of relying on a binary search to find the maximum safeness factor. By using Dijkstra's algorithm, we can ensure that we find the path with the maximum safeness factor, without the need to perform a separate binary search.

Additionally, this approach may be more intuitive for some users, as it closely resembles the problem of finding the shortest path with the maximum weight (safeness factor) on a weighted graph.

**Algorithm**

Initialize dir to store directions for moving to neighboring cells: right, left, down, up.

Define the isValidCell method to check if a given cell is valid within the grid.

Initialize variables:

n as the size of the grid.

q as a queue of coordinates to perform the breadth-first search (BFS).

Mark thieves as 0 and empty cells as -1 in the

 grid. Push thieves' coordinates to the queue.

Perform BFS to calculate the safeness factor for each cell:

While the queue is not empty:

Retrieve the front element curr from the queue.

Explore neighboring cells:

If the neighboring cell is valid and unvisited (safeness factor = -1):

Update its safeness factor and push it to the queue.

Initialize a priority queue

 pq to prioritize cells with a higher safeness factor. Push the starting cell to pq.

Perform BFS to find the path with the maximum safeness factor:

While the priority queue pq is not empty:

Retrieve the top element curr from pq.

If the destination is reached, return the safeness factor of the path.

Explore neighboring cells:

If the neighboring cell is valid and not marked as visited:

Update the safeness factor for the path and mark the cell as visited.

If no path is found, return -1.

Note: In the C++ implementation, the elements in the priority queue are stored as [safeness, row, col] to leverage C++'s default comparison capabilities.

**Implementation**

class Solution {

// Directions for moving to neighboring cells: right, left, down, up

final int[][] dir = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};

public int maximumSafenessFactor(List<List<Integer>> grid) {

int n = grid.size();

int[][] mat = new int[n][n];

Queue<int[]> multiSourceQueue = new LinkedList<>();

// To make modifications and navigation easier, the grid is converted into a 2-d array

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

for (int j = 0; j < n; j++) {

if (grid.get(i).get(j) == 1) {

// Push thief coordinates to the queue

multiSourceQueue.add(new int[] {i, j});

// Mark thief cell with 0

mat[i][j] = 0;

} else {

// Mark empty cell with -1

mat[i][j] = -1;

}

}

}

// Calculate safeness factor for each cell using BFS

while (!multiSourceQueue.isEmpty()) {

int size = multiSourceQueue.size();

while (size-- > 0) {

int[] curr = multiSourceQueue.poll();

// Check neighboring cells

for (int[] d : dir) {

int di = curr[0] + d[0];

int dj = curr[1] + d[1];

int val = mat[curr[0]][curr[1]];

// Check if the neighboring cell is valid and unvisited

if (isValidCell(mat, di, dj) && mat[di][dj] == -1) {

// Update safeness factor and push to the queue

mat[di][dj] = val + 1;

multiSourceQueue.add(new int[] {di, dj});

}

}

}

}

// Priority queue to prioritize cells with higher safeness factor

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

// Push starting cell to the priority queue

pq.add(new int[] {0, 0, mat[0][0]}); // [x-coordinate, y-coordinate, maximum\_safeness\_till\_now]

mat[0][0] = -1; // Mark the source cell as visited

// BFS to find the path with maximum safeness factor

while (!pq.isEmpty()) {

int[] curr = pq.poll();

// If reached the destination, return safeness factor

if (curr[0] == n - 1 && curr[1] == n - 1) {

return curr[2];

}

// Explore neighboring cells

for (int[] d : dir) {

int di = d[0] + curr[0];

int dj = d[1] + curr[1];

if (isValidCell(mat, di, dj) && mat[di][dj] != -1) {

// Update safeness factor for the path and mark the cell as visited

pq.add(new int[] {di, dj, Math.min(curr[2], mat[di][dj])});

mat[di][dj] = -1;

}

}

}

return -1; // No valid path found

}

// Check if a given cell lies within the grid

private boolean isValidCell(int[][] mat, int i, int j) {

int n = mat.length;

}

}

**Complexity Analysis**

Let *n*⋅*n* be the size of the matrix.

Time Complexity: *O*(*n^*2⋅log(*n*))

Similar to Approach 1, the time complexity of the initial BFS is *O*(*n^*2).

To find the optimal path, we use Dijkstra's single source shortest path algorithm, which has a time complexity of *O*(*n^*2⋅log(*n*)) when implemented in a grid of size *n*⋅*n*.

The total time complexity is the sum of the time complexities of the two parts: *O*(*n^*2)+*O*(*n^*2⋅log(*n*)). This can be simplified to *O*(*n^*2⋅log(*n*)).

Space Complexity: *O*(*n^*2)

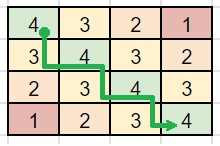
The two data structures used in this approach are the queue and the priority queue, both of which have a linear space complexity. Since the maximum number of elements that can be present in the queues is *n*⋅*n*, the space complexity is *O*(*n^*2).

**Refer to**

<https://leetcode.com/problems/find-the-safest-path-in-a-grid/solutions/3870053/bfs-dijkstra/>

We can use BFS to compute the safeness factor for each cell in the grid.

This is how the resulting grid looks for the third example:



Then, we can run Dijkstra from [0][0], finding the path to the target with the maximum safeness.

**C++**

We re-use the input grid to populate safeness (BFS), and mark visited nodes (Dijkstra).

int maximumSafenessFactor(vector<vector<int>>& g) {

queue<array<int, 2>> q;

int dir[5] = {1, 0, -1, 0, 1}, n = g.size();

for (int i = 0; i < n; ++i)

for (int j = 0; j < n; ++j)

if (g[i][j])

q.push({i, j});

while (!q.empty()) {

auto [i, j] = q.front(); q.pop();

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

int x = i + dir[d], y = j + dir[d + 1];

if (min(x, y) >= 0 && max(x, y) < n && g[x][y] == 0) {

g[x][y] = g[i][j] + 1;

q.push({x, y});

}

}

}

priority\_queue<array<int, 3>> pq;

pq.push({g[0][0], 0, 0});

while (pq.top()[1] < n - 1 || pq.top()[2] < n - 1) {

auto [sf, i, j] = pq.top(); pq.pop();

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

int x = i + dir[d], y = j + dir[d + 1];

if (min(x, y) >= 0 && max(x, y) < n && g[x][y] > 0) {

pq.push({min(sf, g[x][y]), x, y});

g[x][y] \*= -1;

}

}

}

return pq.top()[0] - 1;

}

**Refer to**

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

[L1102.Path With Maximum Minimum Value (Ref.L1368,L2812)](note://WEBf4a6c0c7ead8e64acb71fe2f9814cb8f)

[L1631.Path With Minimum Effort (Ref.L778)](note://WEBdd02629027772608e8c1a486e531fde2)