<https://leetcode.com/problems/shortest-path-in-binary-matrix/description/>

Given an n x n binary matrix grid, return the length of the shortest clear path in the matrix. If there is no clear path, return -1.

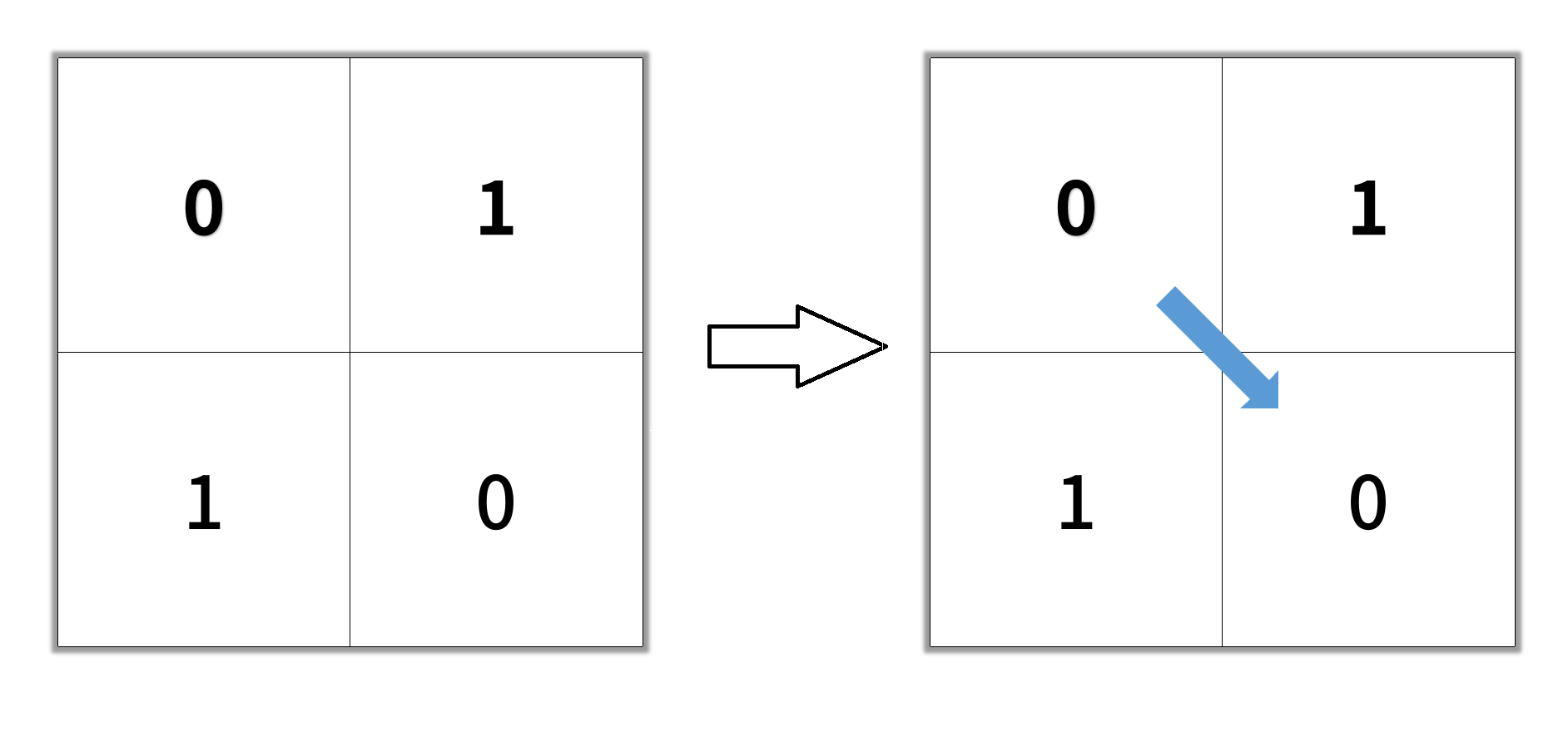
A clear path in a binary matrix is a path from the top-left cell (i.e., (0, 0)) to the bottom-right cell (i.e., (n - 1, n - 1)) such that:

All the visited cells of the path are 0.

All the adjacent cells of the path are 8-directionally connected (i.e., they are different and they share an edge or a corner).

The length of a clear path is the number of visited cells of this path.

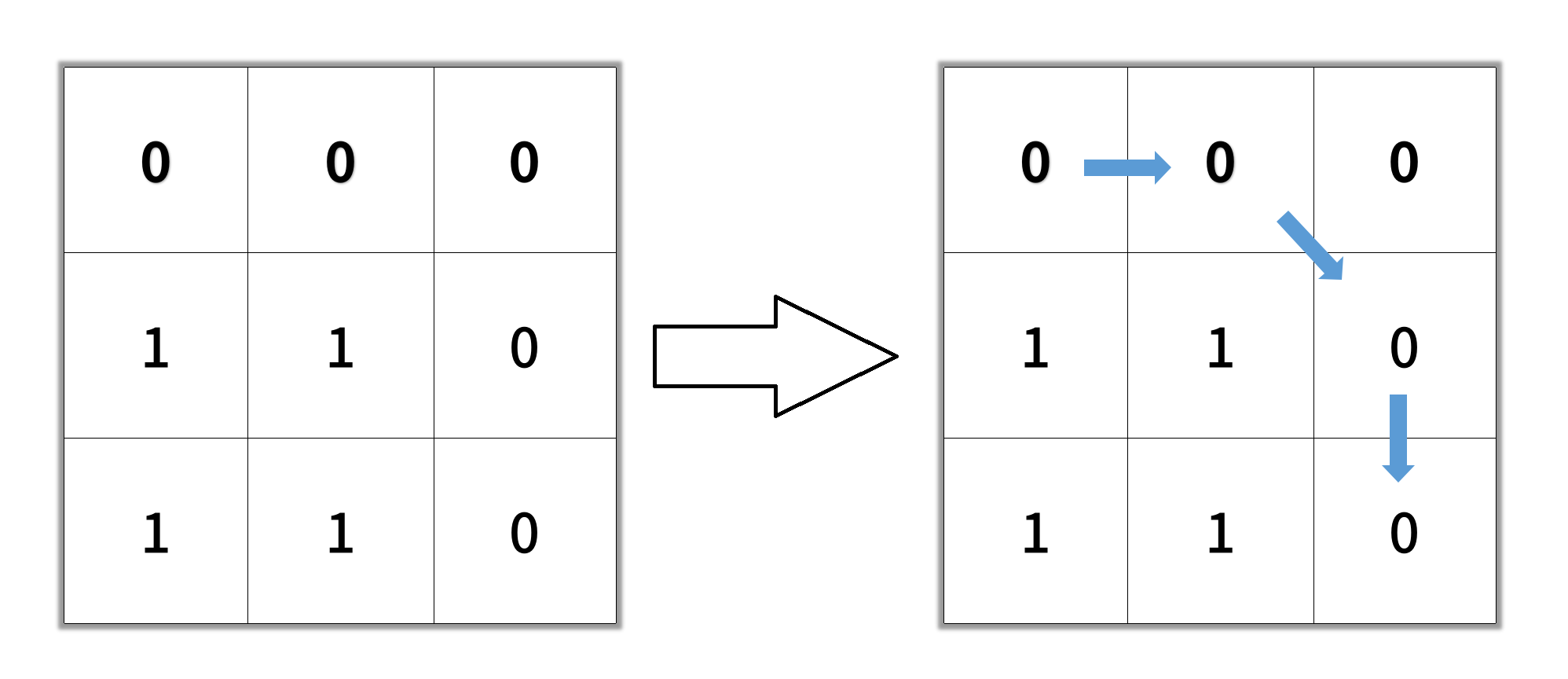
**Example 1:**



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

**Output:** 2

**Example 2:**



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

**Output:** 4

**Example 3:**

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

**Output:** -1

**Constraints:**

n == grid.length

n == grid[i].length

1 <= n <= 100

grid[i][j] is 0 or 1

**Attempt 1: 2024-11-30**

**Solution 1: BFS + Level Order Traversal (10min)**

class Solution {

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

        // Test out: [[1,0,0],[1,1,0],[1,1,0]]

        if(grid[0][0] == 1) {

            return -1;

        }

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

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

        int n = grid.length;

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

        q.offer(new int[]{0, 0});

        grid[0][0] = 1;

        int step = 1;

        while(!q.isEmpty()) {

            int size = q.size();

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

                int[] cur = q.poll();

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

                    return step;

                }

                for(int k = 0; k < 8; k++) {

                    int new\_x = cur[0] + dx[k];

                    int new\_y = cur[1] + dy[k];

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

                        grid[new\_x][new\_y] = 1;

                    }

                }

            }

            step++;

        }

        return -1;

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n^2)

**Refer to Deepseek**

This problem requires finding the shortest path from the top-left corner to the bottom-right corner in a binary matrix, where you can move in 8 directions (up, down, left, right, and diagonally). Here's a BFS solution in Java:

import java.util.Queue;

import java.util.LinkedList;

class Solution {

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

int n = grid.length;

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

return -1;

}

// Directions for 8 possible movements

int[][] directions = {

{-1, -1}, {-1, 0}, {-1, 1},

{0, -1}, {0, 1},

{1, -1}, {1, 0}, {1, 1}

};

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

queue.offer(new int[]{0, 0, 1}); // {row, col, distance}

grid[0][0] = 1; // Mark as visited

while (!queue.isEmpty()) {

int[] current = queue.poll();

int row = current[0];

int col = current[1];

int distance = current[2];

// Check if we've reached the destination

if (row == n - 1 && col == n - 1) {

return distance;

}

// Explore all 8 directions

for (int[] dir : directions) {

int newRow = row + dir[0];

int newCol = col + dir[1];

// Check boundaries and if cell is accessible

&& grid[newRow][newCol] == 0) {

queue.offer(new int[]{newRow, newCol, distance + 1});

grid[newRow][newCol] = 1; // Mark as visited

}

}

}

return -1; // No path found

}

}

**Explanation:**

**Initial Check**: First, we check if the start or end cell is blocked (contains 1), in which case we immediately return -1.

**BFS Setup**: We use a queue to implement BFS, which is ideal for finding shortest paths in unweighted grids. Each queue element stores the current position and the distance traveled so far.

**Directions**: We define all 8 possible movement directions (including diagonals).

**BFS Execution**:

Dequeue the front element and check if it's the destination.

For each of the 8 directions, check if the new position is within bounds and accessible (value 0).

If valid, enqueue the new position with incremented distance and mark it as visited by setting to 1.

**Termination**: If we reach the destination, return the distance. If the queue is exhausted without finding the destination, return -1.

This solution efficiently finds the shortest path using BFS with O(N²) time complexity (where N is the grid size) and O(N²) space complexity for the queue in the worst case.