A maze is given as an n\*n binary matrix of blocks. The rat begins in the upper left most block (i.e maze[0][0]), and its destination block is the lower rightmost block i.e maze[n-1, n-1]. A rat starts from the source and has to reach the destination. The rat can only move forward and down.

In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination.

Ex:

{1, 0, 0, 0}

{1, 1, 0, 1} -> Maze has a path

{0, 1, 0, 0}

{1, 1, 1, 1}

{1, 0, 0, 1}

{0, 0, 1, 1} -> Maze does not have a path

{1, 1, 1, 1}

{1, 1, 1, 1}

Create a method that will return true if there is a valid path, and false if there is not. If a valid solution exists, print out the path it will take (hint: pass in an auxillary int[][] into the helper method to use to track the solution path).

a.) What are some edge cases to check for up front?

If maze[0][0] == 0 or maze [n–1][n-1] == 0, or if maze is null or of length 0.

b.) What errors should you handle when you recurse?

If the recursion would go out of bounds

c.) When should you stop recursing?

When the recursion would go out of bounds, the element we land on is 0, or we get to the end of the maze.

Please implement your solution to the problem below:

// Size of the maze

static int N;

/\* A utility function to print solution matrix

sol[N][N] \*/

void printSolution(int sol[][]) {

     for (int i = 0; i < N; i++) {

         for (int j = 0; j < N; j++)

             System.out.print(" " + sol[i][j] + " ");

         System.out.println();

     }

 }

/\* A utility function to check if x, y is valid

index for N\*N maze \*/

boolean isSafe(int maze[][], int x, int y) {

    // if (x, y outside maze) return false

    return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1);

}

public boolean solveMaze(int[][] maze){

        int sol[][] = new int[N][N];

        if (solveMazeUtil(maze, 0, 0, sol) == false) {

            System.out.print("Solution doesn't exist");

            return false;

        }

        printSolution(sol);

        return true;

}

/\* A recursive utility function to solve Maze

    problem \*/

    boolean solveMazeUtil(int maze[][], int x, int y,

                          int sol[][])

    {

        // if (x, y is goal) return true

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

            sol[x][y] = 1;

            return true;

        }

        // Check if maze[x][y] is valid

        if (isSafe(maze, x, y) == true) {

            // mark x, y as part of solution path

            sol[x][y] = 1;

            /\* Move forward in x direction \*/

            if (solveMazeUtil(maze, x + 1, y, sol))

                return true;

            /\* If moving in x direction doesn't give

            solution then Move down in y direction \*/

            if (solveMazeUtil(maze, x, y + 1, sol))

                return true;

            /\* If none of the above movements works then

            BACKTRACK: unmark x, y as part of solution

            path \*/

            sol[x][y] = 0;

            return false;

        }

        return false;

    }