4. Rat In Maze Puzzle Document

High-Level Description:

To solve Rat in a Maze problem we approached using
recursive backtracking by exploring all possible paths from the starting position
(0, 0) to the destination (n-1, n-1) in a grid. At each cell, the program checks
if moving right or down is valid (for example, the cell is open and within bounds).
If a move is valid, the rat proceeds to the next cell recursively. If it reaches
the destination, it adds the path to a list of solutions. If a path doesn't lead
to the destination, it backtracks and tries the next direction. The process continues
until all paths have been explored.

Model:

- The state passed between recursive calls consists of:
  - The current position of the rat in the grid (row and column indices).
  - The current path taken as a string or a list of coordinates.
  - The maze itself, represented as a 2D array of boolean values (true for open cells, false for cells the rat can not go).

Java Source Code (Recursive Backtracking Implementation)

```
/**

* Recursive method to find all paths from the current position in the maze.

*

* maze The maze represented as a 2D array (1 for open, 0 for blocked).

* x The current row index of the rat.

* y The current column index of the rat.

* path The string representing the path taken so far (composed of 'R' and
'D').

* paths The list that collects all valid paths from start to destination.

*/
```

```
static void findPaths(int[][] maze, int x, int y, String path, List<String>
paths) {

// Check if the current position is the destination

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

paths.add(path); // Add the current path to the list of paths

return; // Exit the method since we found a valid path

}

// Check if the current cell is valid (within bounds and not blocked)

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

// Mark the cell as visited by changing its value to 0

maze[x][y] = 0;



// Move right (to the next column)

findPaths(maze, x, y + 1, path + "R", paths);

// Move down (to the next row)

findPaths(maze, x + 1, y, path + "D", paths);

// Backtrack: unmark the cell to allow other paths to use it

maze[x][y] = 1; // Mark the cell as unvisited

}

}



/**

* Check if a cell in the maze is safe to move to.
```

```
*

* maze The maze represented as a 2D array.

* x The row index to check.

* y The column index to check.

* true if the cell is within bounds and open, false otherwise.

*/

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

// Return true if the cell is within bounds and open (1)

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

}
```

Testing Program

```
import java.util.ArrayList;
import java.util.List;

static final int N = 4;
public static void main(String[] args) {

// Define the maze: 1 is an open cell, 0 is a blocked cell

int[][] maze = {

{1, 0, 0, 0},

{1, 1, 0, 1},

{0, 1, 0, 0},

{0, 1, 1, 1}

};

// List to store all found paths

List<String> paths = new ArrayList<>();
```

```
// Start finding paths from the top-left corner (0, 0)

findPaths(maze, 0, 0, "", paths);

// Output all possible paths

System.out.println("All possible paths:");

for (String path : paths) {

System.out.println(path);

}

}
```

Sample Input and Output:

Inputs:

int[][] maze = {
{1, 0, 0, 0},
{1, 1, 0, 1},
{0, 1, 0, 0},
{1, 1, 1, 1}
};

Expected Outputs:
All Possible paths:
RRDD
RDRD
RDDR

Note: For the output letter "D" refers to move down and the letter "R" refers to move to the right.