Lab 4 - Creation of a Labyrinth using Arrays and Recursion

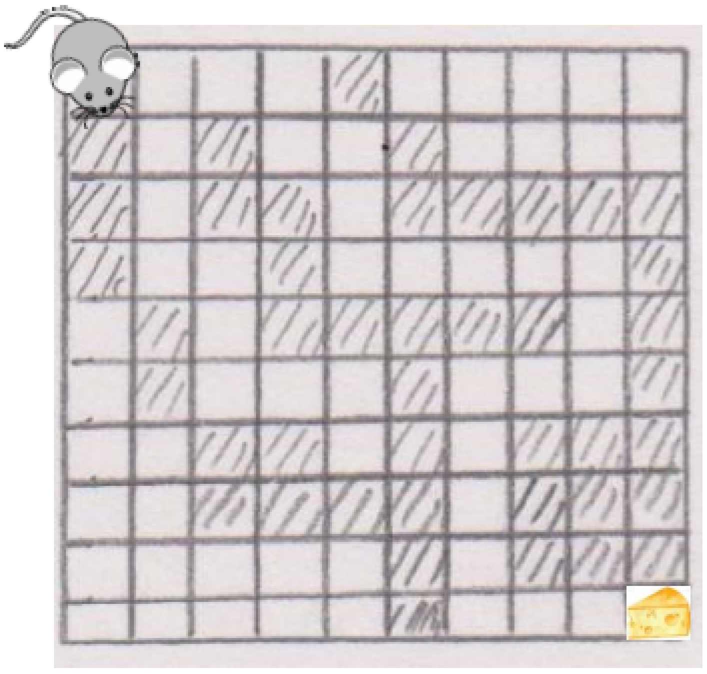
This lab introduces arrays, recursion, and algorithmic thinking through three progressive tasks. Students will learn two-dimensional array manipulation, recursive method design, and backtracking algorithms. The exercises progress from maze pathfinding to Sudoku solving and fractal generation, emphasizing problem decomposition, recursive thinking, and advanced data structure manipulation.

Contents

1. Task 1: Maze Pathfinding with Recursion

Create a recursive pathfinding algorithm to help a mouse navigate through a labyrinth to reach cheese. This task introduces two-dimensional arrays, recursion, and backtracking algorithms.

A poor, hungry mouse sits in the upper left corner of a labyrinth (see sketch) and wants to reach a piece of cheese located in the lower right corner of the labyrinth. It can enter all non-hatched fields, but only via an edge shared by two adjacent fields. Help the mouse reach the cheese. Write a recursive method in Java that shows the mouse a path to the cheese.



Tip

Your method must try for every possible field to find a path to the cheese via each of the four neighboring fields.

Your program should:
1. Represent the labyrinth using a 2D character array
2. Implement a recursive `findPath(int row, int col)` method
3. Use backtracking when paths lead to dead ends
4. Mark visited cells to prevent infinite loops
5. Display the final path through the maze

1.1. Preparation

Represent the labyrinth in a two-dimensional array. Use two more fields than specified by the labyrinth and fill the border fields with walls (corresponding to the hatched elements in the labyrinth). Since the mouse is located in the upper left, there are initially only two meaningful paths. However, from the second step onward, use a method that checks all four directions and calls itself recursively to find the path. Tip: Use a marking character to mark the path that the mouse has taken, as the mouse should not go backwards. (This will be helpful when implementing the algorithm. If the chosen path was not successful, the mouse goes back along this path, deletes the marking again, and starts the search anew.) At the end, the path through the labyrinth should be drawn. If you want, you can recreate the labyrinth with graphical methods and also enter the mouse's path to the cheese. A modification of the labyrinth or a larger number of fields is also possible.

1.2. Assistance

2D Array representation:

```java
public class MazeSolver {
    private char[][] maze = {
        {'#', '#', '#', '#', '#', '#', '#'},
        {'#', ' ', ' ', '#', ' ', ' ', '#'},
        {'#', ' ', '#', '#', ' ', '#', '#'},
        {'#', ' ', ' ', ' ', ' ', '#', '#'},
        {'#', '#', '#', ' ', '#', ' ', '#'},
        {'#', ' ', ' ', ' ', ' ', ' ', '#'},
        {'#', '#', '#', '#', '#', '#', '#'}
    };

    private int startRow = 1, startCol = 1;
    private int endRow = 5, endCol = 5;
}
```

Recursive pathfinding structure:

```java
public boolean findPath(int row, int col) {
    // Base cases
    if (row < 0 || row >= maze.length || col < 0 || col >= maze[0].length) {
        return false; // Out of bounds
    }
    if (maze[row][col] == '#' || maze[row][col] == 'X') {
        return false; // Wall or already visited
    }
    if (row == endRow && col == endCol) {
        maze[row][col] = 'P'; // Mark path
        return true; // Found cheese!
    }

    // Mark current cell as visited
    maze[row][col] = 'X';

    // Try all four directions
    if (findPath(row-1, col) || // North
        findPath(row+1, col) || // South
        findPath(row, col-1) || // West
        findPath(row, col+1)) { // East
        maze[row][col] = 'P'; // Mark as part of solution path
        return true;
    }

    // Backtrack: unmark this cell
    maze[row][col] = ' ';
    return false;
}
```

Display the maze:

```java
public void printMaze() {
    for (int i = 0; i < maze.length; i++) {
        for (int j = 0; j < maze[i].length; j++) {
            System.out.print(maze[i][j] + " ");
        }
        System.out.println();
    }
}
```

2. Task 2: Sudoku Solver with Backtracking (Optional)

Create a Sudoku solver using recursive backtracking. This task builds on the maze pathfinding concepts and applies them to constraint satisfaction problems.

Implement a program that can solve 9x9 Sudoku puzzles using recursive backtracking:

- Read a partially filled Sudoku grid (use 0 for empty cells)
- Find empty cells and try numbers 1-9
- Use recursion to explore all possible solutions
- Backtrack when constraints are violated
- Display the solved puzzle

Your program should:

1. Represent the Sudoku grid as a 2D integer array
2. Implement `isValid(int[][] grid, int row, int col, int num)` to check constraints
3. Create a recursive `solveSudoku(int[][] grid)` method
4. Handle backtracking when no valid numbers can be placed
5. Display the completed puzzle

2.1. Requirements

- Check row, column, and 3x3 box constraints
- Use recursive backtracking algorithm
- Handle cases with no solution
- Display the grid in a readable format

2.2. Assistance

Sudoku grid representation:

```java
public class SudokuSolver {
    private static final int SIZE = 9;
    private static final int EMPTY = 0;

    private int[][] grid = {
        {5, 3, 0, 0, 7, 0, 0, 0, 0},
        {6, 0, 0, 1, 9, 5, 0, 0, 0},
        {0, 9, 8, 0, 0, 0, 0, 6, 0},
        {8, 0, 0, 0, 6, 0, 0, 0, 3},
        {4, 0, 0, 8, 0, 3, 0, 0, 1},
        {7, 0, 0, 0, 2, 0, 0, 0, 6},
        {0, 6, 0, 0, 0, 0, 2, 8, 0},
        {0, 0, 0, 4, 1, 9, 0, 0, 5},
        {0, 0, 0, 0, 8, 0, 0, 7, 9}
    };
}
```

Constraint checking:

```java
private boolean isValid(int[][] grid, int row, int col, int num) {
    // Check row
    for (int x = 0; x < SIZE; x++) {
        if (grid[row][x] == num) return false;
    }

    // Check column
    for (int x = 0; x < SIZE; x++) {
        if (grid[x][col] == num) return false;
    }

    // Check 3x3 box
    int startRow = row - row % 3;
    int startCol = col - col % 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (grid[i + startRow][j + startCol] == num) return false;
        }
    }
    return true;
}
```

Recursive solving:

```java
public boolean solveSudoku(int[][] grid) {
    for (int row = 0; row < SIZE; row++) {
        for (int col = 0; col < SIZE; col++) {
            if (grid[row][col] == EMPTY) {
                for (int num = 1; num <= 9; num++) {
                    if (isValid(grid, row, col, num)) {
                        grid[row][col] = num;

                        if (solveSudoku(grid)) {
                            return true;
                        }

                        grid[row][col] = EMPTY; // Backtrack
                    }
                }
                return false; // No valid number found
            }
        }
    }
    return true; // All cells filled
}
```

3. Lab Execution

If your program is not yet working without issue, we will try to correct this during the course of the lab. With good preparation, this should not be a problem. Every student is required to be able to explain their thought process at the beginning of the lab. By the end of the lab, the task needs to be completed. Of course, we will support you, but your personal commitment must also be clearly recognizable! Julian Moldenhauer, Furkan Yildirim, and Emily Antosch wish you lots of fun and success!