Day 25 Assignment

Name: Mehul Anjikhane Email: mehulanjikhane13@gmail.com

Task 1: The Knight's Tour Problem

Create a function bool SolveKnightsTour(int[,] board, int moveX, int moveY, int moveCount, int[] xMove, int[] yMove) that attempts to solve the Knight's Tour problem using backtracking. The function should return true if a solution exists and false otherwise. The board represents the chessboard, moveX and moveY are the current coordinates of the knight, moveCount is the current move count, and xMove[], yMove[] are the possible next moves for the knight. Fill the chessboard such that the knight visits every square exactly once. Keep the chessboard size to 8x8.

```
package algorithms;

public class KnightsTourProblem {

	static final int N = 8; // Chessboard size (8x8)

	public static boolean isSafe(int board[][], int row, int col)
{
			return (row >= 0 && row < N && col >= 0 && col
< N && board[row][col] == 0);
		}

		public static boolean solveKTUtil(int board[][], int
moveX, int moveY, int moveCount, int[] xMove, int[] yMove) {
			if (moveCount == N * N) {
				return true; // All squares visited
			}

			// Try all possible moves
			for (int i = 0; i < 8; i++) {
				int nextX = moveX + xMove[i];
				int nextY = moveY + yMove[i];

				if (isSafe(board, nextX, nextY)) {
					board[moveX][moveY] =
moveCount + 1; // Mark current square visited
					if (solveKTUtil(board,
nextX, nextY, moveCount + 1, xMove, yMove)) {
						return true;
					} else {

	board[moveX][moveY] = 0; // Backtrack: Unmark if path doesn't
lead to solution
					}
```

```
                        }
                    }

                    return false; // No valid move found
            }

            public static void solveKnightsTour() {
                    int board[][] = new int[N][N];

                // Possible knight moves (8 possible directions)
                    int[] xMove = { 2, 1, -1, -2, -2, -1, 1, 2 };
                    int[] yMove = { 1, 2, 2, 1, -1, -2, -2, -1 };

                    // Start from any corner square
                    board[0][0] = 1; // Mark starting position

                    if (solveKTUtil(board, 0, 0, 1, xMove, yMove))
                    {
                        System.out.println("Solution exists:");
                            printBoard(board);
                    } else {
                            System.out.println("Solution does
not exist");
                    }
            }

            public static void printBoard(int board[][]) {
                    for (int i = 0; i < N; i++) {
                            for (int j = 0; j < N; j++) {
                                    System.out.print(board[i][j]
+ " ");
                            }
                            System.out.println();
                    }
            }
            public static void main(String[] args) {
                    solveKnightsTour();
            }
}
```

Output:

```
Solution exists:
2 61 40 35 32 19 10 0
39 36 33 62 11 64 31 18
60 3 38 41 34 29 20 9
37 50 43 28 63 12 17 30
44 59 4 51 42 25 8 21
49 52 47 56 27 22 13 16
58 45 54 5 24 15 26 7
53 48 57 46 55 6 23 14
```

Task 2: Rat in a Maze

Implement a function bool SolveMaze(int[,] maze) that uses backtracking to find a path from the top left corner to the bottom right corner of a maze. The maze is represented by a 2D array where 1s are paths and 0s are walls. Find a rat's path through the maze. The maze size is 6x6.

```
package algorithms;

import java.util.Arrays;

public class RatInMaze {

	private static final int N = 6;

	public static boolean SolveMaze(int[][] maze) {
		int[][] sol = new int[N][N];

		if (solveMazeUtil(maze, 0, 0, sol) == false) {
			System.out.println("Solution doesn't exist");
				return false;
		}

		System.out.println("Solution:");
		printSolution(sol);
		return true;
	}

	private static boolean solveMazeUtil(int[][] maze, int x,
int y, int[][] sol)
{
		if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {
				sol[x][y] = 1;
				return true;
			}

			if (isSafe(maze, x, y)) {
				sol[x][y] = 1;

			if (solveMazeUtil(maze, x + 1, y, sol))
					return true;

			if (solveMazeUtil(maze, x, y + 1, sol))
					return true;

				sol[x][y] = 0;
				return false;
			}
```

```
                    return false;
          }

          private static boolean isSafe(int[][] maze, int x, int y)
{
               return (x >= 0 && x < N && y >= 0 && y < N &&
maze[x][y] == 1);
          }

          private static void printSolution(int[][] sol) {
               for (int i = 0; i < N; i++) {
                    for (int j = 0; j < N; j++) {
                         if (sol[i][j] == 1) {
                              System.out.print("R ");
                              // R for right move
                         }
                         else {
                              System.out.print("- ");
                         }
                    }
                    System.out.println();
               }
          }

          public static void main(String[] args) {
               int[][] maze = { { 1, 1, 0, 0, 0, 0 }, { 1, 1, 0, 1,
0, 0 },{ 0, 1, 1,1, 0, 0 }, { 0, 0, 0, 1, 1, 0 },{ 0, 0, 0, 1, 1, 1
}, { 0, 0, 0, 0, 1, 1 } };
                    System.out.println("Given Maze:");

                    for (int i = 0; i < maze.length; i++) {

     System.out.println(Arrays.toString(maze[i]));
                    }

                    SolveMaze(maze);
          }
}
```

Output:

```
Given Maze:
[1, 1, 0, 0, 0, 0]
[1, 1, 0, 1, 0, 0]
[0, 1, 1, 1, 0, 0]
[0, 0, 0, 1, 1, 0]
[0, 0, 0, 1, 1, 1]
[0, 0, 0, 0, 1, 1]
```

```
Solution:
R - - - - -
R R - - - -
- R R R - -
- - - R - -
- - - R R -
- - - - R R
```

Task 3: N Queen Problem

Write a function bool SolveNQueen(int[,] board, int col) in C# that places N queens on an N x N chessboard so that no two queens attack each other using backtracking. Place N queens on the board such that no two queens can attack each other. Use a standard 8x8 chessboard.

```
package algorithms;

public class NQueenProblem {

	private static final int N = 8;

	// Function to solve the N Queens problem
	public static boolean SolveNQueen(int[][] board, int col) {

		// Base case: If all queens are placed, return true
		if (col >= N) {
			return true;
		}

		// Try placing a queen in each row of the current column
		for (int i = 0; i < N; i++) {
			if (isSafe(board, i, col)) {
				// Place the queen and recursively
solve for the next column
				board[i][col] = 1;
				if (SolveNQueen(board, col + 1)) {
					return true;
				// If a solution is found, return true
				} else {
					board[i][col] = 0;
		// Backtrack: Remove the queen if no solution is found
				}
			}
		}
		return false; // If no queen can be placed in this
column, return false
	}
```

```
	// Function to check if it's safe to place a queen at
board[row][col]
	private static boolean isSafe(int[][] board, int row, int col)
	{
		int i, j;

		// Check if there is a queen in the same row
		for (i = 0; i < col; i++) {
			if (board[row][i] == 1) {
				return false;
			}
		}

	// Check if there is a queen in the upper left diagonal
		for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
			if (board[i][j] == 1) {
				return false;
			}
		}

	// Check if there is a queen in the lower left diagonal
		for (i = row, j = col; j >= 0 && i < N; i++, j--) {
			if (board[i][j] == 1) {
				return false;
			}
		}
		return true; // If no conflicts, it's safe to place
a queen at board[row][col]
	}

// Function to print the board (0 for empty square, 1 for queen)
	private static void printSolution(int board[][]) {
		for (int i = 0; i < N; i++) {
			for (int j = 0; j < N; j++) {
				if (board[i][j] == 1)
					System.out.print("Q ");
				else
					System.out.print(". ");
			}
			System.out.println();
		}
	}

	public static void main(String[] args) {
		int[][] board = new int[N][N];

	// Initialize the board with zeros (no queens placed)
		for (int i = 0; i < N; i++) {
			for (int j = 0; j < N; j++) {
```

```
                                        board[i][j] = 0;
                                }
                        }
        System.out.println("Standard Board Size: " + N + " X " + N);

                        // Solve the N Queens problem starting from the
first column
                        if (SolveNQueen(board, 0)) {
                                System.out.println("Solution:");
                                printSolution(board); // Print the
solution if it exists
                        } else {
                                System.out.println("No solution exists");
                        }
        }
}
```

Output:

```
Standard Board Size: 8 X 8
Solution:
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```