Name- Amit Kumar Parhi

Email- amitkparhi07@gmail.com

Day 16 and 17:

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.

Solution:

```
package com. wipro;

public class KnightsTourProblem {

    // Size of the chessboard
    static final int N = 8;

    // Function to check if a given position is safe for the knight
    static boolean isSafe(int x, int y, int[][] board) {
        return (x >= 0 && y >= 0 && x < N && y < N && board[x][y] == -1);
    }

    // Function to print the solution
    static void printSolution(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();
        }
    }

    // Function to solve the Knight's Tour problem using backtracking
    static boolean solveKnightsTour(int[][] board, int x, int y, int moveCount, int[]
xMove, int[] yMove) {
        if (moveCount == N * N) {
            printSolution(board);
            return true;
        }

        // Try all next moves from the current position x, y
        for (int i = 0; i < 8; i++) {
            int nextX = x + xMove[i];
            int nextY = y + yMove[i];
            if (isSafe(nextX, nextY, board)) {
```

```
                board[nextX][nextY] = moveCount;
                if (solveKnightsTour(board, nextX, nextY, moveCount + 1, xMove,
yMove)) {
                    return true;
                } else {
                    board[nextX][nextY] = -1; // Backtrack
                }
            }
        }

        return false;
    }

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

        // Initialize the chessboard with -1
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                board[i][j] = -1;
            }
        }

        // Possible moves for the knight
        int[] xMove = {2, 1, -1, -2, -2, -1, 1, 2};
        int[] yMove = {1, 2, 2, 1, -1, -2, -2, -1};

        // Starting position of the knight
        int startX = 0, startY = 0;

        // Mark the starting position as visited
        board[startX][startY] = 0;

        if (!solveKnightsTour(board, startX, startY, 1, xMove, yMove)) {
            System.out.println("No solution exists.");
        }
    }
}
```

Output:

```
0  59 38 33 30 17  8 63
37 34 31 60 9  62 29 16
58 1  36 39 32 27 18 7
35 48 41 26 61 10 15 28
42 57 2  49 40 23  6 19
47 50 45 54 25 20 11 14
56 43 52 3  22 13 24 5
51 46 55 44 53  4 21 12
```

Task 2: Rat in a Maze

mplement 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.

Solution:

```
package com.wipro;

public class RatInAMaze {

    // Size of the maze
    static final int N = 6;

    // Function to check if a given position is safe to move
    static boolean isSafe(int[][] maze, int x, int y) {
        return (x >= 0 && y >= 0 && x < N && y < N && maze[x][y] == 1);
    }

    // Function to print the solution path
    static 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();
        }
    }

    // Function to solve the Rat in a Maze problem using backtracking
    static boolean solveMazeUtil(int[][] maze, int x, int y, int[][] sol) {
        if (x == N - 1 && y == N - 1) {
            sol[x][y] = 1; // Reached the destination
            return true;
        }

        if (isSafe(maze, x, y)) {
            sol[x][y] = 1; // Mark the current cell as part of solution path

            // Move right
            if (solveMazeUtil(maze, x, y + 1, sol)) {
                return true;
            }

            // Move down
            if (solveMazeUtil(maze, x + 1, y, sol)) {
                return true;
            }

            // If neither right nor down leads to the solution, backtrack
            sol[x][y] = 0;
```

```
            return false;
        }

        return false;
    }

    // Function to solve the Rat in a Maze problem
    static boolean solveMaze(int[][] maze) {
        int[][] sol = new int[N][N];

        if (!solveMazeUtil(maze, 0, 0, sol)) {
            System.out.println("No solution exists.");
            return false;
        }

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

    public static void main(String[] args) {
        int[][] maze = {
            {1, 0, 0, 0, 0, 0},
            {1, 1, 1, 1, 1, 1},
            {0, 1, 0, 0, 0, 1},
            {1, 1, 0, 1, 1, 1},
            {1, 1, 0, 0, 0, 1},
            {1, 1, 1, 1, 1, 1}
        };

        solveMaze(maze);
    }
}
```

Output:

```
Solution path:
1 0 0 0 0 0
1 1 1 1 1 1
0 0 0 0 0 1
0 0 0 0 0 1
0 0 0 0 0 1
0 0 0 0 0 1
```

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.

Solution:

```java
package com.wipro;

public class NQueenProblem {

    // Size of the chessboard
    static final int N = 8;

    // Function to check if a queen can be placed safely at position (row, col)
    static boolean isSafe(int[][] board, int row, int col) {
        // Check the column on the left side
        for (int i = 0; i < col; i++) {
            if (board[row][i] == 1) {
                return false;
            }
        }

        // Check upper diagonal on the left side
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        // Check lower diagonal on the left side
        for (int i = row, j = col; i < N && j >= 0; i++, j--) {
            if (board[i][j] == 1) {
                return false;
            }
        }

        return true;
    }

    // Function to solve the N-Queen problem using backtracking
    static boolean solveNQueensUtil(int[][] board, int col) {
        // All queens are placed successfully
        if (col >= N) {
            return true;
        }

        // Try placing queen in each row of the current column
        for (int i = 0; i < N; i++) {
            if (isSafe(board, i, col)) {
                // Place queen at position (i, col)
                board[i][col] = 1;

                // Recur to place the rest of the queens
                if (solveNQueensUtil(board, col + 1)) {
                    return true;
                }

                // If placing queen at (i, col) doesn't lead to a solution, backtrack
                board[i][col] = 0;
```

```
            }
        }

        // If queen cannot be placed in any row of the current column, return false
        return false;
    }

    // Function to solve the N-Queen problem and print the solution
    static boolean solveNQueens() {
        int[][] board = new int[N][N];

        // Initialize the board with 0s
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                board[i][j] = 0;
            }
        }

        if (!solveNQueensUtil(board, 0)) {
            System.out.println("No solution exists.");
            return false;
        }

        // Print the solution
        printSolution(board);
        return true;
    }

    // Function to print the solution
    static void printSolution(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) {
        solveNQueens();
    }
}
```

Output:

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```