Data Structure and Operations: **Day-16-17(The Knight's Tour Problem, Rat in a Maze, N Queen Problem)**:

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 wipro.com.assignment12;

public class KnightsTour {
    // Size of the chessboard
    private static final int N = 8;

    // Function to solve the Knight's Tour problem using backtracking
    public static boolean SolveKnightsTour(int[][] board, int moveX, int moveY, int
moveCount, int[] xMove, int[] yMove) {
        // Base case: if moveCount equals N*N, we have solved the problem
        if (moveCount == N * N) {
            return true;
        }

        // Try all next moves from the current coordinate moveX, moveY
        for (int i = 0; i < N; i++) {
            int nextX = moveX + xMove[i];
            int nextY = moveY + yMove[i];

            // Check if the next move is within bounds and not visited yet
            if (isValidMove(nextX, nextY, board)) {
                // Mark the current move
                board[nextX][nextY] = moveCount + 1;

                // Recursively try the next move
                if (SolveKnightsTour(board, nextX, nextY, moveCount + 1, xMove,
yMove)) {
                    return true;
                } else {
                    // Backtrack: if the recursive call fails, undo the current move
                    board[nextX][nextY] = 0;
                }
            }
        }

        // If no move works from the current position, return false
```

```
        return false;
    }

    // Function to check if the knight's move is valid
    private static boolean isValidMove(int x, int y, int[][] board) {
        return (x >= 0 && x < N && y >= 0 && y < N && board[x][y] == 0);
    }

    public static void main(String[] args) {
        int[][] board = new int[N][N];
        int[] xMove = {2, 1, -1, -2, -2, -1, 1, 2};
        int[] yMove = {1, 2, 2, 1, -1, -2, -2, -1};

        // Starting position for the knight (can be any valid position)
        int startX = 0;
        int startY = 0;

        // Place the knight at the starting position
        board[startX][startY] = 1;

        // Try to solve the Knight's Tour problem from the starting position
        if (SolveKnightsTour(board, startX, startY, 1, xMove, yMove)) {
            // Print the solution
            System.out.println("Solution exists. The knight's tour path:");
            printBoard(board);
        } else {
            System.out.println("Solution does not exist.");
        }
    }

    // Function to print the chessboard
    private 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] + "\t");
            }
            System.out.println();
        }
    }
}
```

Output:

```
Solution exists. The knight's tour path:
1       60      39      34      31      18      9       64
38      35      32      61      10      63      30      17
59      2       37      40      33      28      19      8
36      49      42      27      62      11      16      29
43      58      3       50      41      24      7       20
48      51      46      55      26      21      12      15
57      44      53      4       23      14      25      6
52      47      56      45      54      5       22      13
```

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.

```
package wipro.com.assignment12;

public class RatInMaze {
    // Size of the maze
    private static final int N = 6;

    // Function to solve the Rat in a Maze problem using backtracking
    public static boolean SolveMaze(int[][] maze) {
        // Create a solution matrix to store the path
        int[][] sol = new int[N][N];

        // Initialize the solution matrix
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                sol[i][j] = 0;
            }
        }

        // Start from the top-left corner (0, 0)
        if (solveMazeUtil(maze, 0, 0, sol) == false) {
            System.out.println("Solution does not exist");
            return false;
        }

        // Print the solution path
        printSolution(sol);
        return true;
    }

    // Recursive function to solve the Rat in a Maze problem
    private static boolean solveMazeUtil(int[][] maze, int x, int y, int[][] sol) {
        // If (x, y) is the bottom-right corner, we have solved the maze
        if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {
            sol[x][y] = 1;
            return true;
        }

        // Check if maze[x][y] is a valid position
        if (isSafe(maze, x, y)) {
            // Mark (x, y) as part of the solution path
            sol[x][y] = 1;

            // Move right
```

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

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

            // If no move works, backtrack: unmark (x, y) as part of the solution
path
            sol[x][y] = 0;
            return false;
        }

        return false;
    }

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

    // Function to print the solution matrix
    private static void printSolution(int[][] sol) {
        System.out.println("Solution path:");
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                System.out.print(sol[i][j] + " ");
            }
            System.out.println();
        }
    }

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

        // Attempt to solve the maze from the top-left corner (0, 0)
        if (!SolveMaze(maze)) {
            System.out.println("No solution exists");
        }
    }
}
```

Output:

```
Solution does not exist
No solution exists
```

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 wipro.com.assignment12;
public class NQueenProblem {
    // Size of the chessboard
    private static final int N = 8;

    // Function to solve the N Queen problem using backtracking
    public static boolean SolveNQueen(int[][] board, int col) {
        // Base case: if all queens are placed, return true
        if (col >= N) {
            return true;
        }

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

                // Recursively place queens in the next columns
                if (SolveNQueen(board, col + 1)) {
                    return true;
                }

                // If placing queen in board[i][col] doesn't lead to a solution,
backtrack
                board[i][col] = 0;
            }
        }

        // If no queen can be placed in this column `col`, return false
        return false;
    }

    // Function to check if a queen can be placed on board[row][col]
    private static boolean isSafe(int[][] board, int row, int col) {
        // Check if there's a queen in the same row to the left
        for (int i = 0; i < col; i++) {
            if (board[row][i] == 1) {
                return false;
            }
```

```
        }

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

        // If no conflicts, it's safe to place a queen at board[row][col]
        return true;
    }

    // Function to print the board
    private 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) {
        int[][] board = new int[N][N];

        // Attempt to solve the N Queen problem starting from the first column (0)
        if (SolveNQueen(board, 0)) {
            System.out.println("Solution exists. The board configuration:");
            printBoard(board);
        } else {
            System.out.println("Solution does not exist.");
        }
    }
}
```

Output:

Solution exists. The board configuration:

```
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
```