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.

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 Os are walls. Find a rat's path through the maze he maze size is 6x6.

- **Initialization**:
  - N is the size of the maze (6x6).
  - solution is a 2D array to store the path from the start to the end of the maze.
- **solveMazeUtil**:
  - This is a recursive utility function that uses backtracking to solve the maze.
  - It marks the current cell (x, y) as part of the solution path.
  - It tries to move in four possible directions (right, down, left, and up) and recursively checks if moving in that direction leads to a solution.
  - If none of the directions lead to a solution, it backtracks by unmarking the cell (x, y).
- **isSafe**:
  - This function checks if the cell (x, y) is within the maze bounds and is a valid path (i.e., maze[x][y] == 1).
- **printSolution**:
  - This function prints the solution matrix.
- **main**:
  - It defines the maze and calls the SolveMaze function to find and print the path from the start to the end of the maze.

Task 3: N Queen Problem

Write a function bool SolveNQueen(int[,] board, int col) in java 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.

To solve the N-Queens problem using backtracking in Java, we need to ensure that no two queens attack each other. This means that no two queens can be in the same row, column, or diagonal. The standard 8x8 chessboard will be used in this example.

1. **Initialization**:
   - N is the size of the chessboard (8x8 in this case).
2. **SolveNQueen**:
   - This is the main function that tries to solve the N-Queens problem.
   - It takes the chessboard and the current column as input.
   - If all queens are placed (i.e., col >= N), it returns true.
   - It tries to place a queen in each row of the current column. If placing the queen leads to a solution, it returns true. If not, it backtracks by removing the queen.
3. **isSafe**:
   - This function checks if it is safe to place a queen at board[row][col].
   - It checks for queens in the same row to the left, in the upper diagonal to the left, and in the lower diagonal to the left.
4. **printSolution**:
   - This function prints the board with the placed queens.
5. **main**:
   - It initializes the chessboard and calls SolveNQueen to solve the problem.
   - If a solution is found, it prints the board; otherwise, it prints "No solution found".

This implementation ensures that the queens are placed in such a way that no two queens attack each other, and it uses backtracking to explore all possible configurations.

```java
public class NQueens {
    private static final int N = 8;

    // Method 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
                board[i][col] = 1;

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

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

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

    // Method 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 this row on the left side
        for (i = 0; i < col; i++) {
            if (board[row][i] == 1) {
                return false;
            }
        }

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

    return true;
}

// Method to print the solution board
private 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) {
    int[][] board = new int[N][N];
```

```
        if (!SolveNQueen(board, 0)) {
            System.out.println("No solution found");
        } else {
            printSolution(board);
        }
    }
}
```