Himanshu Kamane
kamanehimanshu76@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.

Solution:

```
package com.app;
public class KnightsTour {
   static int N = 8; // Size of the chessboard
   // Utility function to check if the move is valid
   static boolean isSafe(int x, int y, int[][] board) {
       return (x >= 0 && x < N && y >= 0 && y < N && board[x][y] == -1);
   }
   // Utility function to print the solution board
   static void printSolution(int[][] board) {
       for (int x = 0; x < N; x++) {
           for (int y = 0; y < N; y++) {
               System.out.print(board[x][y] + "\t");
           }
           System.out.println();
       }
   }
   // This function solves the Knight's Tour problem using Backtracking.
   // This function returns false if no complete tour is possible, otherwise
   // return true and prints the tour.
   static boolean solveKnightsTour(int[][] board, int moveX, int moveY, int
moveCount, int[] xMove, int[] yMove) {
       int nextX, nextY;
       if (moveCount == N * N) {
           return true;
       }
       // Try all next moves from the current coordinate moveX, moveY
       for (int k = 0; k < 8; k++) {
           nextX = moveX + xMove[k];
```

```
            nextY = moveY + yMove[k];
            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; // Backtracking
                }
            }
        }
        return false;
    }
    public static void main(String[] args) {
        int[][] board = new int[N][N];
        // Initialization of the solution matrix
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++) {
                board[x][y] = -1;
            }
        }
        // xMove[] and yMove[] define the next move of Knight.
        // xMove[] is for the next value of x coordinate
        // yMove[] is for the next value of y coordinate
        int[] xMove = { 2, 1, -1, -2, -2, -1, 1, 2 };
        int[] yMove = { 1, 2, 2, 1, -1, -2, -2, -1 };
        // Since the Knight is initially at the first block
        board[0][0] = 0;
        System.out.println("Starting the Knight's Tour problem...");
        // Start from 0,0 and explore all tours using solveKnightsTour()
        if (!solveKnightsTour(board, 0, 0, 1, xMove, yMove)) {
            System.out.println("Solution does not exist");
        } else {
            System.out.println("Solution found:");
            printSolution(board);
        }
    }
}
```

Output:

```
Starting the Knight's Tour problem...
Solution found:
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:

```java
package com.app;
public class RatInMaze {
   static int N = 6; // Size of the maze
   // Utility function to check if x, y is valid index for N x N maze
   static boolean isSafe(int x, int y, int[][] maze) {
       // Returns true if x, y is a valid index for N x N maze
       return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1);
   }
   // Utility function to print the solution path
   static void printSolution(int[][] solution) {
       for (int x = 0; x < N; x++) {
           for (int y = 0; y < N; y++) {
               System.out.print(solution[x][y] + " ");
           }
           System.out.println();
       }
   }
   // A utility function to solve the Maze problem using Backtracking.
   // It returns false if no path is possible, otherwise return true and prints
the path
   static boolean solveMazeUtil(int[][] maze, int x, int y, int[][] solution) {
       // If (x, y) is the bottom-right corner, return true
       if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {
           solution[x][y] = 1;
```

```
            return true;
        }
        // Check if maze[x][y] is valid
        if (isSafe(x, y, maze)) {
            // Mark x, y as part of the solution path
            solution[x][y] = 1;
            System.out.println("Move to (" + x + ", " + y + ")");
            // Move forward in x direction
            if (solveMazeUtil(maze, x + 1, y, solution)) {
                return true;
            }
            // If moving in x direction doesn't give solution then move down in
y direction
            if (solveMazeUtil(maze, x, y + 1, solution)) {
                return true;
            }
            // If none of the above movements work then backtrack
            System.out.println("Backtrack from (" + x + ", " + y + ")");
            solution[x][y] = 0;
            return false;
        }
        return false;
    }
    // This function solves the Maze problem using Backtracking. It mainly uses
solveMazeUtil()
    // to solve the problem. It returns false if no path is possible, otherwise
return true and
    // prints the path in the form of 1s.
    static boolean solveMaze(int[][] maze) {
        int[][] solution = new int[N][N];
        System.out.println("Starting to solve the maze...");
        if (!solveMazeUtil(maze, 0, 0, solution)) {
            System.out.println("No solution exists");
            return false;
        }
        System.out.println("Solution found:");
        printSolution(solution);
        return true;
    }
    public static void main(String[] args) {
        int[][] maze = {
            {1, 0, 0, 0, 0, 0},
            {1, 1, 0, 1, 1, 0},
            {0, 1, 0, 0, 1, 0},
            {0, 1, 1, 1, 1, 0},
            {0, 0, 0, 0, 1, 0},
            {0, 0, 0, 0, 1, 1}
        };
        solveMaze(maze);
```

```
    }
}
```

Output:

```
Starting to solve the maze...
Move to (0, 0)
Move to (1, 0)
Move to (1, 1)
Move to (2, 1)
Move to (3, 1)
Move to (3, 2)
Move to (3, 3)
Move to (3, 4)
Move to (4, 4)
Move to (5, 4)
Solution found:
1 0 0 0 0 0
1 1 0 0 0 0
0 1 0 0 0 0
0 1 1 1 1 0
0 0 0 0 1 0
0 0 0 0 1 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:

```
package com.app;
public class NQueens {
    final int N = 8;
    /* A utility function to print solution */
    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();
        }
    }
    /* A utility function to check if a queen can be placed on board[row][col].
```

```
   Note that this function is called when "col" queens are already placed in
columns from 0 to col -1.
   So we need to check only left side for attacking queens */
   boolean isSafe(int board[][], int row, int col) {
       int i, j;
       /* Check this row on left side */
       for (i = 0; i < col; i++)
           if (board[row][i] == 1)
               return false;
       /* Check upper diagonal on left side */
       for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
           if (board[i][j] == 1)
               return false;
       /* Check lower diagonal on left side */
       for (i = row, j = col; j >= 0 && i < N; i++, j--)
           if (board[i][j] == 1)
               return false;
       return true;
   }
   /* A recursive utility function to solve N Queen problem */
   boolean solveNQUtil(int board[][], int col) {
       /* base case: If all queens are placed then return true */
       if (col >= N)
           return true;
       /* Consider this column and try placing this queen in all rows one by
one */
       for (int i = 0; i < N; i++) {
           /* Check if the queen can be placed on board[i][col] */
           if (isSafe(board, i, col)) {
               /* Place this queen in board[i][col] */
               board[i][col] = 1;
               /* recur to place rest of the queens */
               if (solveNQUtil(board, col + 1))
                   return true;
               /* If placing queen in board[i][col] doesn't lead to a solution
then remove queen from board[i][col] */
               board[i][col] = 0; // BACKTRACK
           }
       }
       /* If the queen can not be placed in any row in this column col, then
return false */
       return false;
   }
   /* This function solves the N Queen problem using Backtracking. It mainly
uses solveNQUtil() to solve the problem.
   It returns false if queens cannot be placed, otherwise, it returns true and
prints the placement of queens in the form of a 2D matrix.
   Please note that there may be more than one solutions, this function prints
one of the feasible solutions.*/
```

```
    boolean solveNQ() {
        int board[][] = new int[N][N];
        if (!solveNQUtil(board, 0)) {
            System.out.print("Solution does not exist");
            return false;
        }
        printSolution(board);
        return true;
    }
    // Driver code
    public static void main(String args[]) {
        NQueens Queen = new NQueens();
        Queen.solveNQ();
    }
}
```

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
```