

Backtracking Algorithms - Assignment

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.

Code:

```
package knightstour;

public class KnightsTourProblem {
    static int N = 8;
    static int[] xMove = { 2, 1, -1, -2, -2, -1, 1, 2 };
    static int[] yMove = { 1, 2, 2, 1, -1, -2, -2, -1 };

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

    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] + " ");
            }
            System.out.println();
        }
    }
}
```

```

static boolean solveKnightsTour(int[][] board, int moveX, int moveY, int
moveCount) {
    int nextX, nextY;
    if (moveCount ==  $N * N$ ) {
        return true;
    }
    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)) {
                return true;
            } else {
                board[nextX][nextY] = -1;
            }
        }
    }
    return false;
}

```

```

static boolean solveKT() {
    int[][] board = new int[ $N$ ][ $N$ ];
    for (int x = 0; x <  $N$ ; x++) {
        for (int y = 0; y <  $N$ ; y++) {
            board[x][y] = -1;
        }
    }
    int startX = 0;
    int startY = 0;
    board[startX][startY] = 0;
    if (!solveKnightsTour(board, startX, startY, 1)) {
        System.out.println("Solution does not exist");
    }
    return false;
}

```

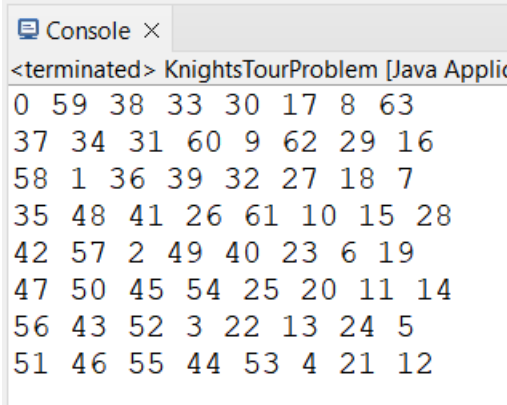
```

    } else {
        printSolution(board);
    }
    return true;
}

public static void main(String[] args) {
    solveKT();
}
}

```

Output:



```

Console ×
<terminated> KnightsTourProblem [Java Applic
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

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.

Code:

```

package ratinaMaze;

public class RatInAMaze {
    static final int N = 6;

    static void printSolution(int[][] solution) {

```

```

for (int[] row : solution) {
for (int cell : row) {
System.out.print(cell + " ");
}
System.out.println();
}
}

```

```

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

```

```

static boolean solveMaze(int[][] maze) {
int[][] solution = new int[N][N];

```

```

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

```

```

    printSolution(solution);
return true;
}

```

```

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

```

```

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

```

```

if (solveMazeUtil(maze, x + 1, y, solution)) {

```

```

    return true;
}
if (solveMazeUtil(maze, x, y + 1, solution)) {
    return true;
}
solution[x][y] = 0;
return false;
}
return false;
}

```

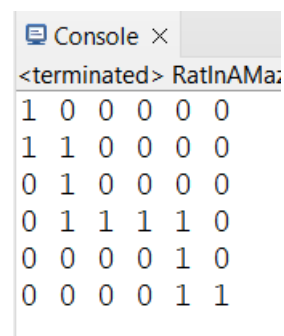
```

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:



```

<terminated> RatInAMa:
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 Java that places N queens on an $N \times N$ 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 8×8 chessboard.

Code:

```
package nqueenproblem;
```

```
public class NQueenProblem {  
    static final int  $N$  = 8;
```

```
    static void printSolution(int[][] board) {  
        for (int[] row : board) {  
            for (int cell : row) {  
                System.out.print(cell + " ");  
            }  
            System.out.println();  
        }  
    }
```

```
    static boolean isSafe(int[][] board, int row, int col) {  
        for (int i = 0; i < col; i++)  
            if (board[row][i] == 1)  
                return false;  
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)  
            if (board[i][j] == 1)  
                return false;  
        for (int i = row, j = col; j >= 0 && i <  $N$ ; i++, j--)  
            if (board[i][j] == 1)  
                return false;  
        return true;  
    }
```

```

static boolean solveNQueen(int[][] board, int col) {
    if (col >= N)
        return true;

    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQueen(board, col + 1))
                return true;
            board[i][col] = 0;
        }
    }
    return false;
}

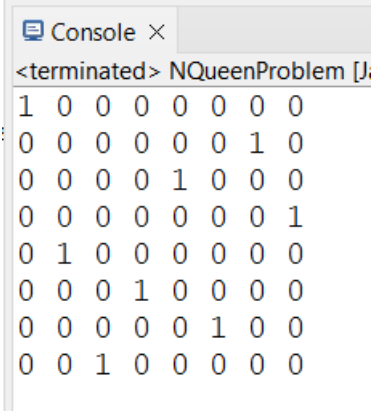
```

```

public static void main(String[] args) {
    int[][] board = new int[N][N];
    if (!solveNQueen(board, 0)) {
        System.out.println("Solution does not exist");
    } else {
        printSolution(board);
    }
}

```

Output:



```

Console ×
<terminated> NQueenProblem [J:
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

```