Day 21

Assignment 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.

A)

The Knight's Tour problem using backtracking in Java. The function SolveKnightsTour will attempt to find a path that visits every square on an 8x8 chessboard exactly once using the knight's move.

Java:

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

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

  private 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();
    }
  }

  public static boolean solveKnightsTour() {
    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 next value of x coordinate
    // yMove[] is for 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;

    // Start from 0,0 and explore all tours using solveKTUtil()
    if (!solveKTUtil(0, 0, 1, board, xMove, yMove)) {
      System.out.println("Solution does not exist");
      return false;
    } else {
      printSolution(board);
    }

    return true;
  }

  private static boolean solveKTUtil(int x, int y, int moveCount, int[][] board, int[] xMove, int[] yMove) {
    int nextX, nextY;
    if (moveCount == N * N) {
      return true;
    }

    for (int k = 0; k < 8; k++) {
      nextX = x + xMove[k];
      nextY = y + yMove[k];
      if (isSafe(nextX, nextY, board)) {
        board[nextX][nextY] = moveCount;
        if (solveKTUtil(nextX, nextY, moveCount + 1, board, xMove, yMove)) {
          return true;
        } else {
          board[nextX][nextY] = -1; // backtracking
        }
      }
    }

    return false;
  }

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

Explanation:

1. Initialization:

- The chessboard (board) is initialized to -1 indicating unvisited squares.

- The knight starts at the first block (0,0), so board[0][0] is set to 0 indicating the first move.

2. Movement Arrays:

- xMove and yMove arrays represent the possible moves a knight can make in terms of x and y coordinates respectively.

3. Recursive Backtracking:

- solveKTUtil is a recursive function that attempts to solve the problem.

*- If moveCount equals N * N, all squares are visited, and the function returns true.*

- For each possible move, the function checks if the next position is valid (within bounds and unvisited). If it is, the knight moves to that position and the function calls itself recursively with the updated move count.

- If the move leads to a solution, the function returns true. Otherwise, it backtracks by marking the square as unvisited and tries the next move.

4. Solution Output:

- If a solution exists, it is printed out using the printSolution method. If not, a message stating that no solution exists is printed.