Prajwalhonashetti143@gmail.com

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.

This Java program solves the Knight's Tour problem using backtracking. The goal is to find a sequence of moves for a knight on an 8x8 chessboard such that the knight visits every square exactly once.

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

    // Function to print the solution matrix
    private static void printSolution(int[][] board) {
        for (int[] row : board) {
            for (int x : row) {
                System.out.print(x + " ");
            }
            System.out.println();
        }
    }

    // Function to check if a particular move is valid
    private static boolean isValidMove(int x, int y, int[][] board) {
```

```
        return (x >= 0 && y >= 0 && x < N && y < N && board[x][y] == -1);
    }

    // Function to solve the Knight's Tour problem using backtracking
    private static boolean solveKnightsTour(int[][] board, int moveCount, int x, int y, int[]
xMove, int[] yMove) {
        // Base case: If all moves are completed, return true
        if (moveCount == N * N) {
            printSolution(board);
            return true;
        }

        // Try all next moves from the current coordinate x, y
        for (int i = 0; i < N; i++) {
            int nextX = x + xMove[i];
            int nextY = y + yMove[i];
            if (isValidMove(nextX, nextY, board)) {
                board[nextX][nextY] = moveCount;
                if (solveKnightsTour(board, moveCount + 1, nextX, nextY, xMove, yMove)) {
                    return true;
                } else {
                    board[nextX][nextY] = -1; // Backtrack
                }
            }
        }

        return false;
    }

    public static void solveKnightsTour() {
```

```
        int[][] board = new int[N][N];

        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++) {
                board[x][y] = -1;
            }
        }

        // Possible knight moves
        int[] xMove = {2, 1, -1, -2, -2, -1, 1, 2};
        int[] yMove = {1, 2, 2, 1, -1, -2, -2, -1};

        // Start from the top-left corner (0, 0) and moveCount 0
        board[0][0] = 0;

        // Start the recursive backtracking process from position (0, 0)
        if (!solveKnightsTour(board, 1, 0, 0, xMove, yMove)) {
            System.out.println("Solution does not exist");
        }
    }
    public static void main(String[] args) {
        solveKnightsTour();
    }
}
```

Key Methods:

printSolution method:

- This method prints the current state of the chessboard.
- It prints each row of the chessboard on a new line, with each cell containing the move number when the knight visited that cell.

isValidMove method:

- Checks if a move to position (x, y) on the chessboard is valid:
- The position (x, y) must be within the bounds of the chessboard.
- The position must not have been visited before (board[x][y] == -1).

solveKnightsTour method:

- This is the main recursive function that attempts to solve the Knight's Tour problem.
- Base Case:
    - If all cells are visited (moveCount == N * N), it prints the solution and returns true.
- Recursive Case:
- It tries all possible knight moves from the current position (x, y).
- For each valid move, it marks the cell as visited (board[nextX][nextY] = moveCount) and recursively tries to solve the rest of the tour.
- If a recursive call returns true, it means a solution is found, so it returns true.
- If not, it backtracks by marking the current cell as unvisited (board[nextX][nextY] = -1).
- If all moves from the current position fail to find a solution, it returns false.

solveKnightsTour method (initialization and start):

- Initializes the chessboard (board) with all cells marked as unvisited (-1).
- Defines the possible moves of the knight (xMove and yMove arrays).
- Starts the backtracking process from the top-left corner of the board (0, 0) with an initial move count of 1.

main method:

Entry point of the program.

Calls the solveKnightsTour method to start solving the Knight's Tour problem.

Output:

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.

```
public class RatInMaze {
    // Size of the maze
    private static final int N = 6;
    // Function to print the solution matrix
    private static void printSolution(int[][] solution) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                System.out.print(solution[i][j] + " ");
            }
            System.out.println();
        }
    }
    // Function to check if rat can move to position (x, y)
    private static boolean isSafe(int[][] maze, int x, int y) {
        // If (x, y) is within maze boundaries and is a valid path (1)
        return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1);
    }

    // Function to solve the Rat in a Maze problem using backtracking
```

```java
private static boolean solveMaze(int[][] maze, int x, int y, int[][] solution) {
    // If the rat reaches the bottom-right corner of the maze
    if (x == N - 1 && y == N - 1) {
        solution[x][y] = 1;
        printSolution(solution);
        return true;
    }

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

        // Move right
        if (solveMaze(maze, x, y + 1, solution))
            return true;

        // Move down
        if (solveMaze(maze, x + 1, y, solution))
            return true;

        // If no move is possible, backtrack: Unmark x, y
        solution[x][y] = 0;
        return false;
    }

    return false;
}
```

```
    // Main function to solve the Rat in a Maze problem and print the solution
    public static void solveMaze(int[][] maze) {
        int[][] solution = new int[N][N];

        if (!solveMaze(maze, 0, 0, solution)) {
            System.out.println("Solution does not exist");
        }
    }

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

        solveMaze(maze);
    }
}
```

Explanation:

printSolution method:

Prints the solution matrix, which shows the path taken by the rat through the maze.

isSafe method:

- Checks if the rat can move to position (x, y):
- (x, y) should be within the maze boundaries (0 to N-1).
- (x, y) should be a valid path (maze[x][y] == 1).

solveMaze method:

- This is the main recursive function that attempts to solve the Rat in a Maze problem.
- Base Case:
- If the rat reaches the bottom-right corner of the maze (x == N-1 && y == N-1), it prints the solution and returns true.
- Recursive Case:
- It checks if (**x, y)** is a valid position to move:
- Marks (**x, y)** as part of the solution path (solution[x][y] = 1).
- Attempts to move right **(solveMaze(maze, x, y + 1, solution)).**
- Attempts to move down (**solveMaze(maze, x + 1, y, solution)).**
- If both moves fail, it backtracks by unmarking **(x, y) (solution[x][y] = 0) and returns false.**
- If no solution exists, it returns false from the main function.

solveMaze method (initialization and start):

- Initializes the solution matrix solution with all cells marked as 0.
- Calls the solveMaze method to start solving the Rat in a Maze problem from the top-left corner (0, 0).

main method:

- Entry point of the program.
- Defines the maze using a 2D array maze.
- Calls the solveMaze method to find and print the solution to the Rat in a Maze problem.

Output:

1 0 0 0 0 0

1 1 0 0 0 0

0 1 1 1 0 0

0 0 0 1 1 0

0 0 0 0 1 1

0 0 0 0 0 1

Explanation of the Output:

- The output is a 6x6 grid where 1 indicates the path taken by the rat through the maze.
- The rat starts at position (0, 0) (top-left corner) and moves towards the bottom-right corner.
- The path taken by the rat is shown as a sequence of 1s, which represents the valid moves through the maze.
- The rest of the cells remain 0, which indicates the walls and cells that the rat did not visit.

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.

```
using System;

public class NQueens {

  // Size of the chessboard

  private static int N = 8;


  // 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++) {

        Console.Write(board[i, j] + " ");

      }

      Console.WriteLine();

    }

    Console.WriteLine();

  }
```

```
// Function to check if a queen can be placed on board[row, col]
private static bool IsSafe(int[,] board, int row, int col) {
    // Check if there is a queen in the same column up to this row
    for (int i = 0; i < row; i++) {
        if (board[i, col] == 1) {
            return false;
        }
    }

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

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

    return true;
}

// Function to solve the N-Queens problem using backtracking
private static bool SolveNQueens(int[,] board, int col) {
    // If all queens are placed, return true
```

```
    if (col >= N) {
        PrintBoard(board);
        return true;
    }

    // Try placing queen in each row of the current column
    for (int i = 0; i < N; i++) {
        if (IsSafe(board, i, col)) {
            board[i, col] = 1; // Place queen

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

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

    return false;
}

// Main function to solve the N-Queens problem and print the solution
public static void SolveNQueens() {
    int[,] board = new int[N, N];

    if (!SolveNQueens(board, 0)) {
        Console.WriteLine("Solution does not exist");
```

```
      }
   }

   // Driver method
   public static void Main(string[] args) {
      SolveNQueens();
   }
}
```

Explanation:

PrintBoard method:

Prints the current state of the chessboard.

IsSafe method:

Checks if a queen can be placed at position (row, col) on the board:

Checks if there is no queen in the same column (board[i, col]).

Checks if there is no queen in the upper left diagonal (board[i, j] where i and j decrement together).

Checks if there is no queen in the upper right diagonal (board[i, j] where i decrements and j increments).

SolveNQueens method:

- Main recursive function to solve the N-Queens problem.
- Base Case:
- If all queens are placed (col >= N), prints the solution and returns true.
- Recursive Case:
- Tries placing a queen in each row of the current column (col).
- If placing a queen is safe, marks the position on the board (board[i, col] = 1) and recursively tries to place the rest of the queens.
- If placing a queen leads to a solution, returns true.
- If not, backtracks by removing the queen from that position (board[i, col] = 0) and tries the next row.
- If no solution is found for the current configuration, returns false.

SolveNQueens method (initialization and start):

- Initializes the chessboard (board) with all cells marked as 0 (no queen placed).
- Calls SolveNQueens to start solving the N-Queens problem from the first column (col = 0).

Main method:

- Entry point of the program.
- Calls the SolveNQueens method to find and print the solution to the N-Queens problem.

Output:

1 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 1

0 0 0 0 0 1 0 0

0 0 1 0 0 0 0 0

0 0 0 0 0 0 1 0

0 1 0 0 0 0 0 0

0 0 0 1 0 0 0 0