Day 16 and 17: Manisha 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.

Explanation:

1. KnightsTour Class:
- **Contains methods to solve the Knight's Tour problem and print the solution.**

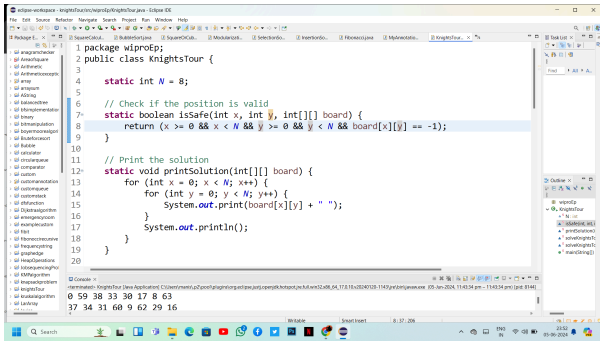
2. isSafe Method:
- **Checks if the current position `(x, y)` is valid (i.e., within the bounds of the board and not yet visited).**

3. printSolution Method:
- **Prints the chessboard solution where each cell indicates the move count at that position.**

4. solveKnightsTourUtil Method:
- **Recursively tries to solve the Knight's Tour problem using backtracking.**
- **Takes the current position `(x, y)`, the current move count, the chessboard, and possible moves `xMove` and `yMove`.**
- **Returns true if a solution is found; otherwise, backtracks and tries another move.**

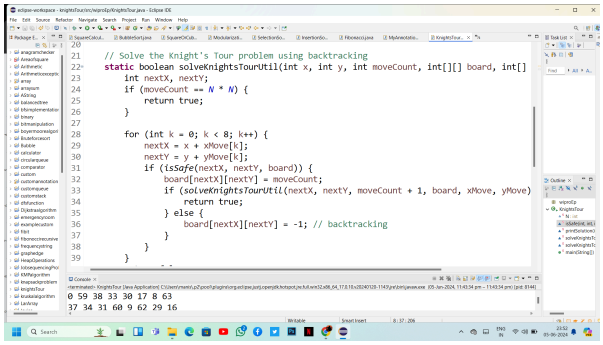
5. solveKnightsTour Method:
- **Initializes the chessboard and possible moves.**
- **Starts from position `(0, 0)` and tries to solve the Knight's Tour problem.**
- **Calls `solveKnightsTourUtil` to attempt to find a solution.**
- **Prints the solution if found; otherwise, prints that no solution exists.**

6. main Method:
- **Calls `solveKnightsTour` to start the solution process.**



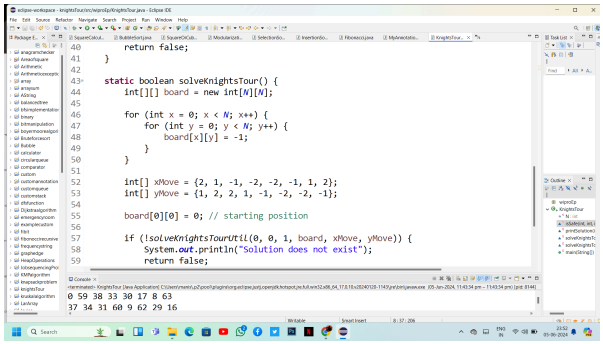
```
package wiprofp;
public class KnightsTour {

    static int N = 8;

    // Check if the position is valid
    static boolean isSafe(int x, int y, int[][] board) {
        return (x >= 0 && x < N && y >= 0 && y < N && board[x][y] == -1);
    }

    // Print the solution
    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();
        }
    }
```

```
0 59 38 33 30 17 8 63
37 34 31 60 9 62 29 16
```



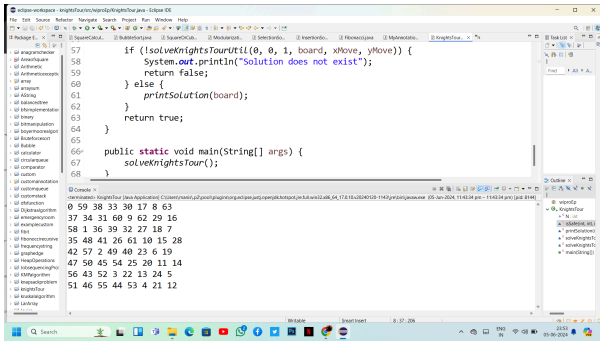
```
    // Solve the Knight's Tour problem using backtracking
    static boolean solveKnightsTourUtil(int x, int y, int moveCount, int[][] board, int[]
        int nextX, nextY;
        if (moveCount == N * N) {
            return true;
        }

        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 (solveKnightsTourUtil(nextX, nextY, moveCount + 1, board, xMove, yMove)
                    return true;
                } else {
                    board[nextX][nextY] = -1; // backtracking
                }
            }
        }
```

```
0 59 38 33 30 17 8 63
37 34 31 60 9 62 29 16
```



```
        return false;
    }

    static boolean 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;
            }
        }

        int[] xMove = {2, 1, -1, -2, -2, -1, 1, 2};
        int[] yMove = {1, 2, 2, 1, -1, -2, -2, -1};

        board[0][0] = 0; // starting position

        if (!solveKnightsTourUtil(0, 0, 1, board, xMove, yMove)) {
            System.out.println("Solution does not exist");
            return false;
```

```
0 59 38 33 30 17 8 63
37 34 31 60 9 62 29 16
```



```
        if (!solveKnightsTourUtil(0, 0, 1, board, xMove, yMove)) {
            System.out.println("Solution does not exist");
            return false;
        } else {
            printSolution(board);
        }
        return true;
    }

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

```
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.

Explanation:

1. MazeSolver Class:
- **Contains methods to solve the maze problem and print the solution.**

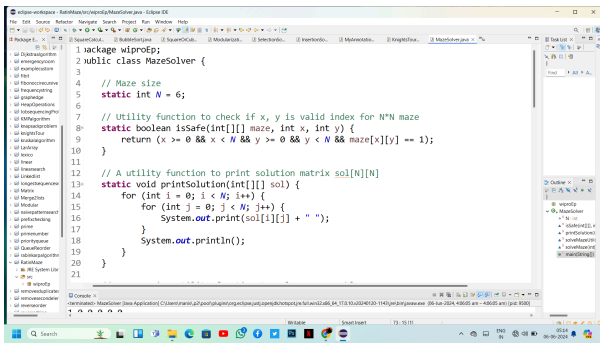
2. isSafe Method:
- **Checks if the current position `(x, y)` is valid (i.e., within the bounds of the maze and is a path).**

3. printSolution Method:
- **Prints the solution matrix where each cell indicates whether it's part of the path.**

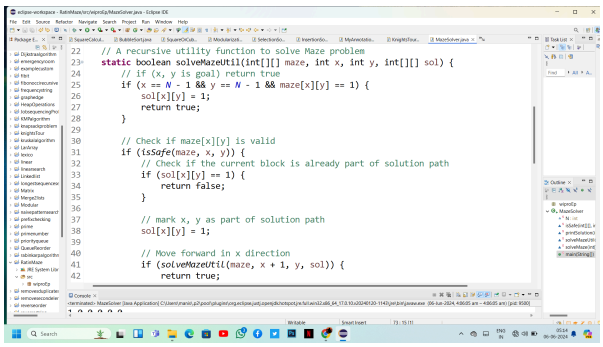
4. solveMazeUtil Method:
- **Recursively tries to solve the maze problem using backtracking.**
- **Takes the current position `(x, y)` and the solution matrix `sol`.**
- **Returns true if a solution is found; otherwise, backtracks and tries another move.**

5. solveMaze Method:
- **Initializes the solution matrix.**
- **Starts from position `(0, 0)` and tries to solve the maze problem.**
- **Calls `solveMazeUtil` to attempt to find a solution.**
- **Prints the solution if found; otherwise, prints that no solution exists.**

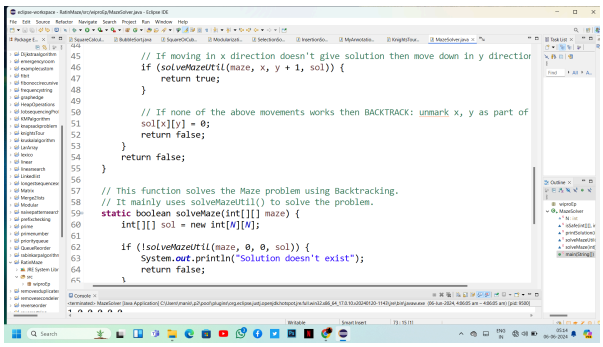
6. main Method:
- **Defines the sample maze.**
- **Calls the `solveMaze` method with the sample maze.**



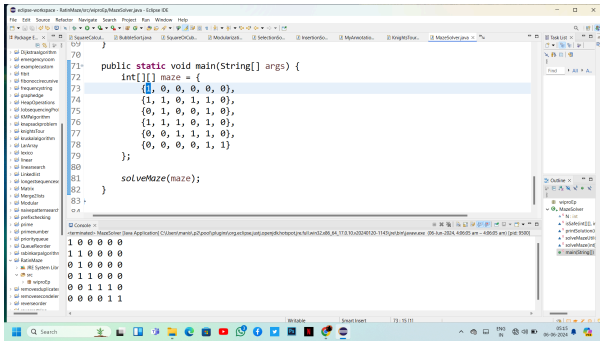
```java
package wiproEp;
public class MazeSolver {

    // Maze size
    static int N = 6;

    // Utility function to check if x, y is valid index for N*N maze
    static boolean isSafe(int[][] maze, int x, int y) {
        return (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1);
    }

    // A utility function to print solution matrix sol[N][N]
    static void printSolution(int[][] sol) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                System.out.print(sol[i][j] + " ");
            }
            System.out.println();
        }
    }
```



```java
    // A recursive utility function to solve Maze problem
    static boolean solveMazeUtil(int[][] maze, int x, int y, int[][] sol) {
        // If (x, y is goal) return true
        if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {
            sol[x][y] = 1;
            return true;
        }

        // Check if maze[x][y] is valid
        if (isSafe(maze, x, y)) {
            // Check if the current block is already part of solution path
            if (sol[x][y] == 1) {
                return false;
            }

            // mark x, y as part of solution path
            sol[x][y] = 1;

            // Move forward in x direction
            if (solveMazeUtil(maze, x + 1, y, sol)) {
                return true;
```



```java
            // If moving in x direction doesn't give solution then move down in y directior
            if (solveMazeUtil(maze, x, y + 1, sol)) {
                return true;
            }

            // If none of the above movements works then BACKTRACK: unmark x, y as part of
            sol[x][y] = 0;
            return false;
        }
        return false;
    }

    // This function solves the Maze problem using Backtracking.
    // It mainly uses solveMazeUtil() to solve the problem.
    static boolean solveMaze(int[][] maze) {
        int[][] sol = new int[N][N];

        if (!solveMazeUtil(maze, 0, 0, sol)) {
            System.out.println("Solution doesn't exist");
            return false;
        }
```



```java
    }

    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},
            {1, 1, 1, 0, 1, 0},
            {0, 0, 1, 1, 1, 0},
            {0, 0, 0, 0, 1, 1}
        };

        solveMaze(maze);
    }
```

```
1 0 0 0 0 0
1 1 0 0 0 0
0 1 0 0 0 0
0 1 1 0 0 0
0 0 1 1 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.

Explanation:

1. NQueenSolver Class:
- Contains methods to solve the N-Queens problem and print the solution.

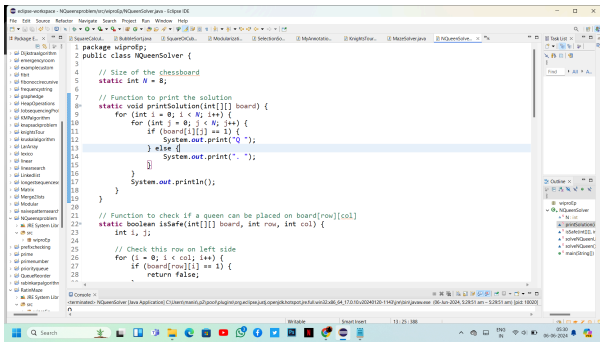
2. printSolution Method:
- Prints the solution board where 'Q' represents a queen and '.' represents an empty cell.

3. isSafe Method:
- Checks if it's safe to place a queen at `board[row][col]`.
- Verifies that no queens are present in the same row, upper diagonal, and lower diagonal on the left side.

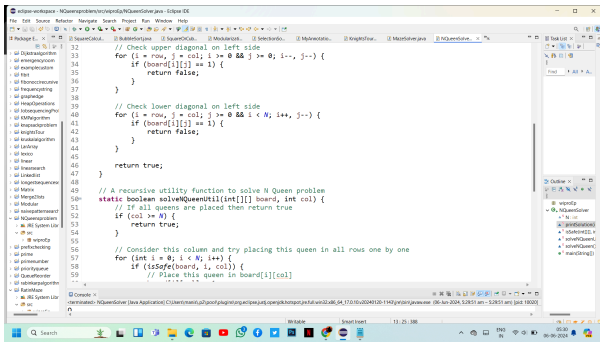
4. solveNQueenUtil Method:
- Recursively tries to place queens in each column.
- If all queens are placed, returns true.
- If placing a queen leads to a solution, returns true; otherwise, removes the queen (backtracks) and tries the next possibility.

5. solveNQueen Method:
- Initializes the chessboard.
- Calls `solveNQueenUtil` to attempt to find a solution.
- Prints the solution if found; otherwise, prints that no solution exists.

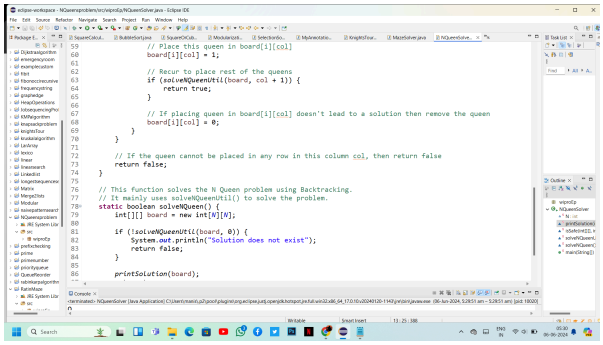
6. main Method:
- Calls the `solveNQueen` method to solve the problem.



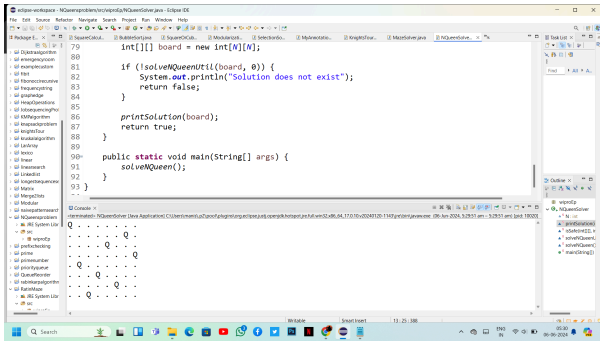
```
package wiproEp;
public class NQueenSolver {

    // Size of the chessboard
    static int N = 8;

    // Function to print the solution
    static void printSolution(int[][] board) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (board[i][j] == 1) {
                    System.out.print("Q ");
                } else {
                    System.out.print(". ");
                }
            }
            System.out.println();
        }
    }

    // Function to check if a queen can be placed on board[row][col]
    static 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
    static boolean solveNQueenUtil(int[][] board, int col) {
        // 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++) {
            if (isSafe(board, i, col)) {
                // Place this queen in board[i][col]
```



```
                // Place this queen in board[i][col]
                board[i][col] = 1;

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

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

        // If the queen cannot 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 solveNQueenUtil() to solve the problem.
    static boolean solveNQueen() {
        int[][] board = new int[N][N];

        if (!solveNQueenUtil(board, 0)) {
            System.out.println("Solution does not exist");
            return false;
        }

        printSolution(board);
```



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

        if (!solveNQueenUtil(board, 0)) {
            System.out.println("Solution does not exist");
            return false;
        }

        printSolution(board);
        return true;
    }

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

```
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```