

Name : Tejaswini Ani Kamble

Email : teju000kamble@gmail.com

Assignments : 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.

Ans: Source Code

```
package com.ds.backtrackingalgo;

public class KnightsTourAlgo {
    // Possible moves of a Knight
    int[] pathRow = { 2, 2, 1, 1, -1, -1, -2, -2 };
    int[] pathCol = { -1, 1, -2, 2, -2, 2, -1, 1 };

    public static void main(String[] args)
    {
        KnightsTourAlgo knightTour = new KnightsTourAlgo();
        int[][] visited = new int[8][8];
        visited[0][0] = 1;

        if (!(knightTour.findKnightTour(visited, 0, 0, 1)))
        {
            System.out.println("Soultion Not Available :(");
        }
    }
}
```

```

private boolean findKnightTour(int[][] visited, int row, int col, int move)
{
    if (move == 64)
    {
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                System.out.printf("%2d ",visited[i][j]);
            }
            System.out.println();
        }
        return true;
    } else
    {
        for (int index = 0; index < pathRow.length; index++)
        {
            int rowNew = row + pathRow[index];
            int colNew = col + pathCol[index];
            // Try all the moves from current coordinate
            if (isValidMove(visited, rowNew, colNew))
            {
                // apply the move
                move++;
                visited[rowNew][colNew] = move;
                if (findKnightTour(visited, rowNew, colNew,
move)) {
                    return true;
                }
                // backtrack the move
                move--;
                visited[rowNew][colNew] = 0;
            }
        }
    }

    return false;
}

private boolean isValidMove(int[][] visited, int rowNew, int colNew)

```

```

{
if (rowNew >= 0 && rowNew < 8 && colNew >= 0 && colNew < 8 &&
visited[rowNew][colNew] == 0)
{
return true;
}
return false;
}
}

```

```

1 package com.ds.backtrackingalgo;
2
3 public class KnightsTourAlgo {
4     // Possible moves of a Knight
5     int[] pathRow = { 2, 2, 1, 1, -1, -1, -2, -2 };
6     int[] pathCol = { -1, 1, -2, 2, -2, 2, -1, 1 };
7
8     public static void main(String[] args) {
9         KnightsTourAlgo knightTour = new KnightsTourAlgo();
10        int[][] visited = new int[8][8];
11        visited[0][0] = 1;
12
13        if (!knightTour.findKnightTour(visited, 0, 0, 1)) {
14            System.out.println("Solution Not Available :(");
15        }
16    }
17
18    private boolean findKnightTour(int[][] visited, int row, int col, int move) {
19        if (move == 64) {
20            for (int i = 0; i < 8; i++) {
21                for (int j = 0; j < 8; j++) {
22                    System.out.printf("%2d ", visited[i][j]);
23                }
24                System.out.println();
25            }
26            return true;
27        } else {
28            for (int index = 0; index < pathRow.length; index++) {
29                int rowNew = row + pathRow[index];
30                int colNew = col + pathCol[index];
31                // Try all the moves from current coordinate

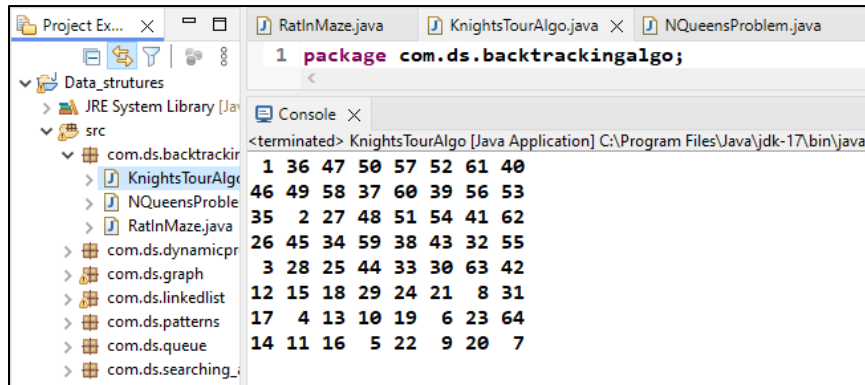
```

```

31                // Try all the moves from current coordinate
32                if (isValidMove(visited, rowNew, colNew)) {
33                    // apply the move
34                    move++;
35                    visited[rowNew][colNew] = move;
36                    if (findKnightTour(visited, rowNew, colNew, move)) {
37                        return true;
38                    }
39                    // backtrack the move
40                    move--;
41                    visited[rowNew][colNew] = 0;
42                }
43            }
44        }
45    }
46 }

```

Output:



The screenshot shows an IDE with a project named 'Data_structures'. The source code for 'KnightsTourAlgo.java' is visible, showing the package 'com.ds.backtrackingalgo;'. The console output displays a 6x6 maze solution as a grid of numbers:

```
<terminated> KnightsTourAlgo [Java Application] C:\Program Files\Java\jdk-17\bin\java
1 36 47 50 57 52 61 40
46 49 58 37 60 39 56 53
35 2 27 48 51 54 41 62
26 45 34 59 38 43 32 55
3 28 25 44 33 30 63 42
12 15 18 29 24 21 8 31
17 4 13 10 19 6 23 64
14 11 16 5 22 9 20 7
```

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.

Ans: Source Code

```
package com.ds.backtrackingalgo;

public class RatMazeAssignment
{
    private static final int MAZE_SIZE = 6;

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

        if (solveMaze(maze))
```

```

{
    System.out.println("Path found!");
} else {
    System.out.println("No path found :(");
}
}

public static boolean solveMaze(int[][] maze)
{
    int[][] solution = new int[MAZE_SIZE][MAZE_SIZE];
    if (!findPath(maze, 0, 0, solution))
    {
        return false;
    }

    printSolution(solution);
    return true;
}

private static boolean findPath(int[][] maze, int row, int col, int[][] solution)
{
    if (row == MAZE_SIZE - 1 && col == MAZE_SIZE - 1)
    {
        solution[row][col] = 1;
        return true;
    }
    if (isValidMove(maze, row, col))
    {
        solution[row][col] = 1;
        if (findPath(maze, row, col + 1, solution))
        {
            return true;
        }
        if (findPath(maze, row + 1, col, solution))
        {
            return true;
        }
        solution[row][col] = 0;
        return false;
    }

    return false;
}

```

```

    }

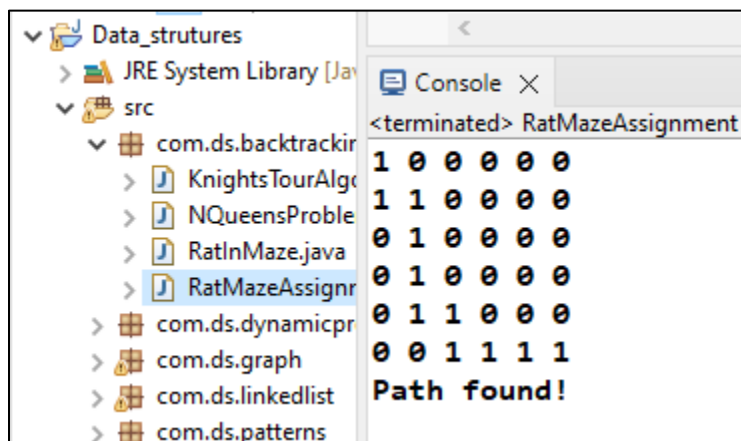
    private static boolean isValidMove(int[][] maze, int row, int col) {

return row >= 0 && row < MAZE_SIZE && col >= 0 && col < MAZE_SIZE
&& maze[row][col] == 1;
    }

    private static void printSolution(int[][] solution) {
        for (int i = 0; i < MAZE_SIZE; i++)
        {
            for (int j = 0; j < MAZE_SIZE; j++)
            {
                System.out.print(solution[i][j] + " ");
            }
            System.out.println();
        }
    }
}

```

Output:



The screenshot shows an IDE with a project named 'Data_structures'. The 'src' folder contains a package 'com.ds.backtrackir' with several classes: 'KnightsTourAlgo', 'NQueensProblem', 'RatInMaze.java', and 'RatMazeAssignnr'. The 'RatMazeAssignnr' class is selected. The console window shows the output of the program, which is a 6x6 maze solution and the message 'Path found!'.

```

<terminated> RatMazeAssignment
1 0 0 0 0 0
1 1 0 0 0 0
0 1 0 0 0 0
0 1 0 0 0 0
0 1 1 0 0 0
0 0 1 1 1 1
Path found!

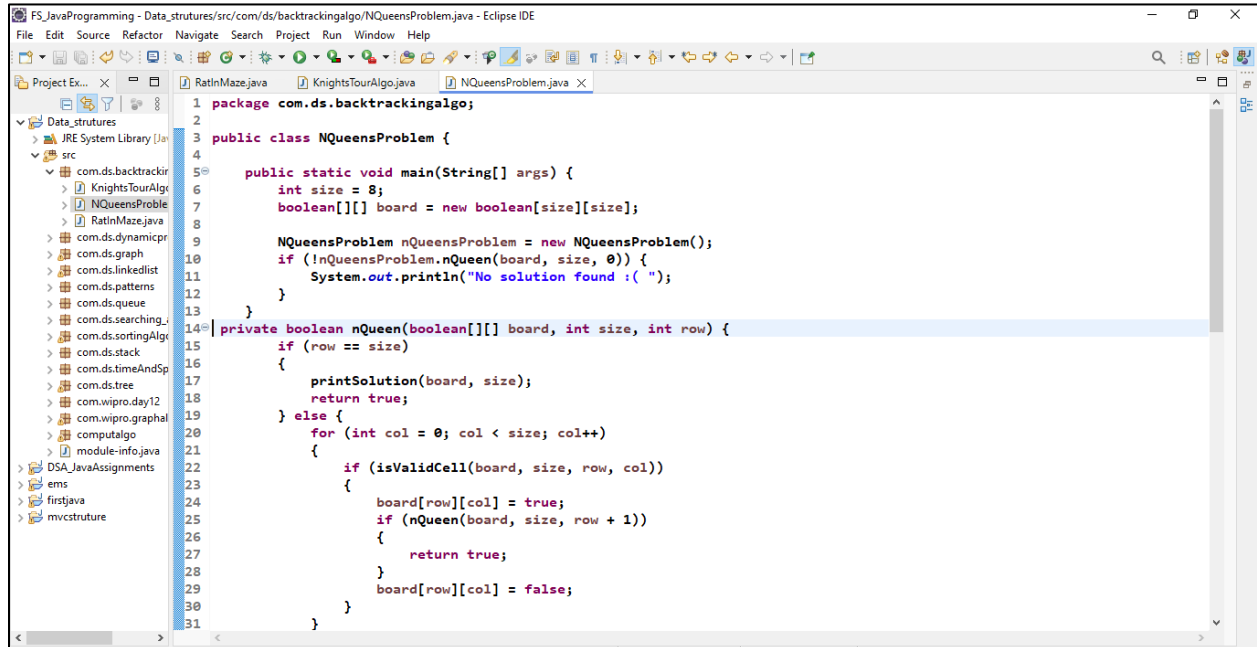
```

Task 3: N Queen Problem

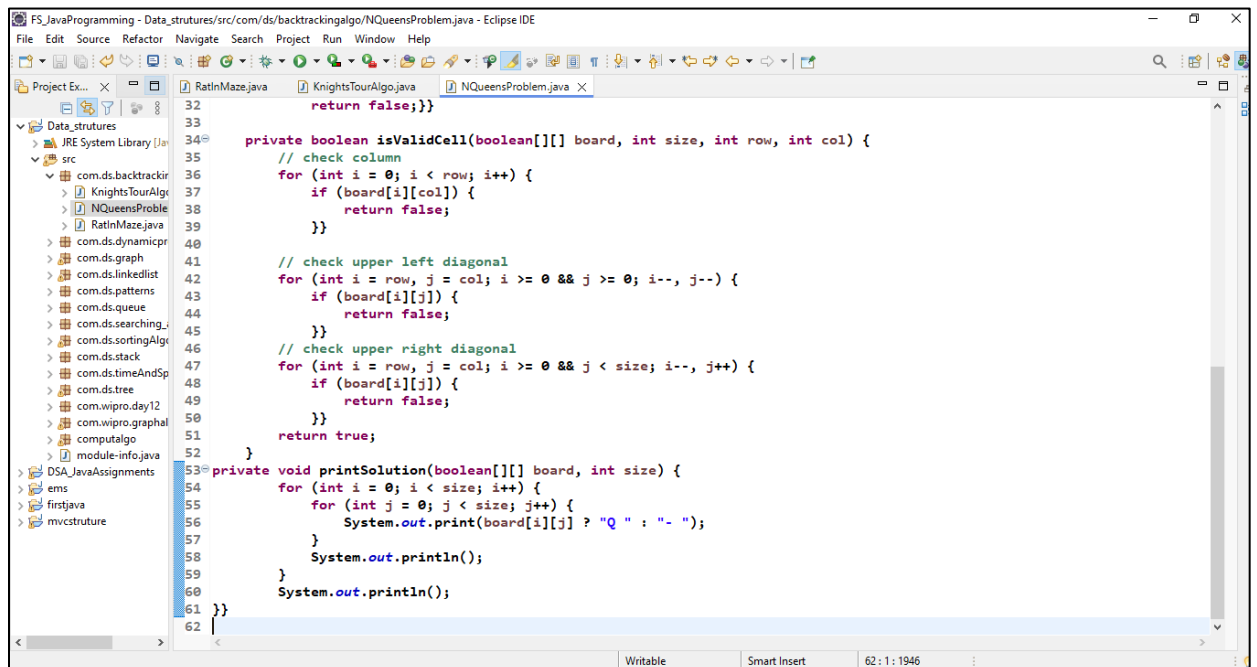
Write a function `bool SolveNQueen(int[,] board, int col)` in C# that places N queens on an N x N chessboard so that no two queens attack each other using

backtracking. Place N queens on the board such that no two queens can attack each other. Use a standard 8x8 chessboard.

Ans: Source Code

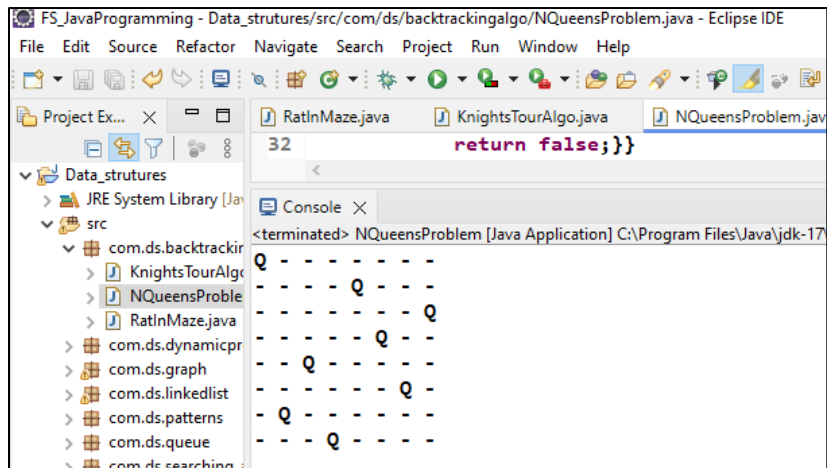


```
1 package com.ds.backtrackingalgo;
2
3 public class NQueensProblem {
4
5     public static void main(String[] args) {
6         int size = 8;
7         boolean[][] board = new boolean[size][size];
8
9         NQueensProblem nQueensProblem = new NQueensProblem();
10        if (!nQueensProblem.nQueen(board, size, 0)) {
11            System.out.println("No solution found :( ");
12        }
13    }
14    private boolean nQueen(boolean[][] board, int size, int row) {
15        if (row == size)
16        {
17            printSolution(board, size);
18            return true;
19        } else {
20            for (int col = 0; col < size; col++)
21            {
22                if (isValidCell(board, size, row, col))
23                {
24                    board[row][col] = true;
25                    if (nQueen(board, size, row + 1))
26                    {
27                        return true;
28                    }
29                    board[row][col] = false;
30                }
31            }
32        }
33    }
34    private boolean isValidCell(boolean[][] board, int size, int row, int col) {
35        // check column
36        for (int i = 0; i < row; i++) {
37            if (board[i][col]) {
38                return false;
39            }
40        }
41        // check upper left diagonal
42        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
43            if (board[i][j]) {
44                return false;
45            }
46        }
47        // check upper right diagonal
48        for (int i = row, j = col; i >= 0 && j < size; i--, j++) {
49            if (board[i][j]) {
50                return false;
51            }
52        }
53        return true;
54    }
55    private void printSolution(boolean[][] board, int size) {
56        for (int i = 0; i < size; i++) {
57            for (int j = 0; j < size; j++) {
58                System.out.print(board[i][j] ? "Q " : "- ");
59            }
60            System.out.println();
61        }
62    }
63 }
```



```
32
33
34    private boolean isValidCell(boolean[][] board, int size, int row, int col) {
35        // check column
36        for (int i = 0; i < row; i++) {
37            if (board[i][col]) {
38                return false;
39            }
40        }
41        // check upper left diagonal
42        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
43            if (board[i][j]) {
44                return false;
45            }
46        }
47        // check upper right diagonal
48        for (int i = row, j = col; i >= 0 && j < size; i--, j++) {
49            if (board[i][j]) {
50                return false;
51            }
52        }
53        return true;
54    }
55    private void printSolution(boolean[][] board, int size) {
56        for (int i = 0; i < size; i++) {
57            for (int j = 0; j < size; j++) {
58                System.out.print(board[i][j] ? "Q " : "- ");
59            }
60            System.out.println();
61        }
62    }
63 }
```

Output:



The screenshot shows the Eclipse IDE interface. The top menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, and Help. The toolbar contains various icons for file operations and development tools. The Project Explorer on the left shows the project structure: Data_structures (JRE System Library, src). The src folder contains the following files: com.ds.backtracking (KnightsTourAlgo.java, NQueensProblem.java, RatInMaze.java), com.ds.dynamicprogramming, com.ds.graph, com.ds.linkedlist, com.ds.patterns, com.ds.queue, and com.ds.searching. The main editor displays the NQueensProblem.java file, showing the code for the NQueensProblem class. The console output shows the execution of the NQueensProblem class, displaying the solution for the NQueensProblem (Java Application) C:\Program Files\Java\jdk-17. The output is a 6x6 grid of characters representing the solution for the NQueensProblem.

```
Q - - - Q - - -  
- - - Q - - -  
- - - - - Q  
- - - - - Q - -  
- - Q - - - -  
- - - - - Q -  
- Q - - - -  
- - - Q - - -
```