

# N-Queens Visualizer

COURSE NAME: DATA STRUCTURES AND ALGORITHMS

SUBMITTED TO:- MR. RAHUL SINGH RAJPUT

SUBMITTED BY: GAUTAM KRISHNA

LPU REGISTRATION NUMBER: 12214473

SUBMISSION DATE: 13-07-2024

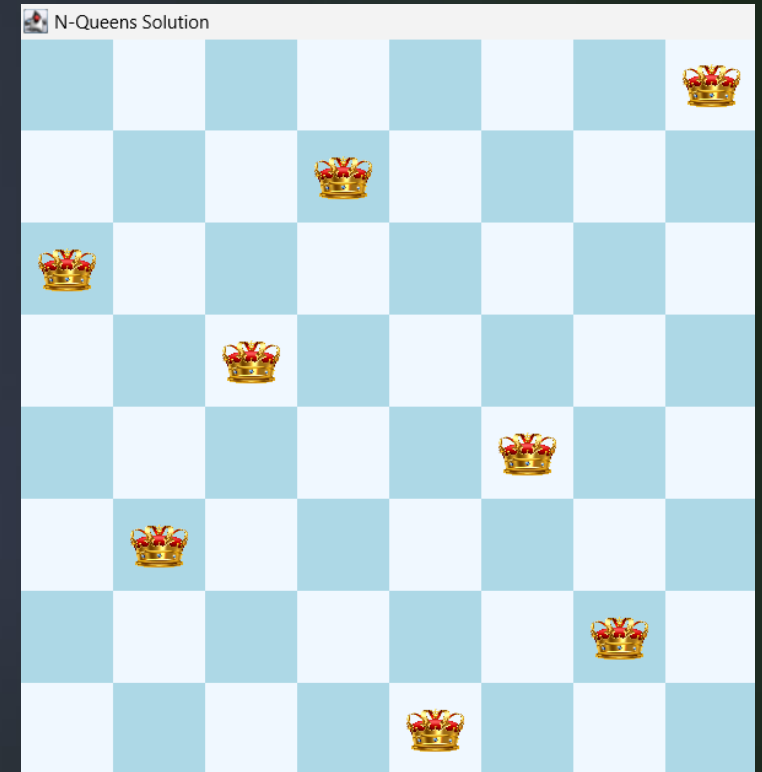


# Introduction

- The N-Queens problem is a classic puzzle in which the goal is to place N queens on an  $N \times N$  chessboard so that no two queens threaten each other.
- This means that no two queens can be in the same row, column, or diagonal.
- The problem is an example of a more general class of problems called constraint satisfaction problems, and it can be solved using various algorithms, including backtracking.

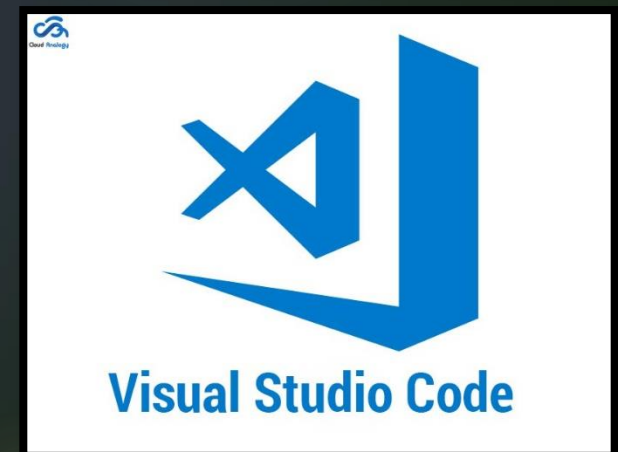
# Objectives of the Project

- ▶ Create a Visual Representation
- ▶ Dynamic Input
- ▶ Animated Solving Process
- ▶ Display All Possible Solutions
- ▶ Enhance User Experience



# Technology and Tools

- ▶ **Programming Language: Java**
- ▶ **Development Environment :- Visual Studio Code (VSCode)**
- ▶ **Libraries and Frameworks Used: Swing, Java Standard Library**



# Project Design

## ► NQueensVisualizer Class

The NQueensVisualizer class is the core component of the project, responsible for rendering the chessboard, managing the placement of queens, and executing the backtracking algorithm. It extends JPanel to leverage Swing's painting capabilities.



```
public class NQueensVisualizer extends JPanel {
```

# paintComponent(Graphics g)

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            if ((row + col) % 2 == 0) {
                g.setColor(new Color(r:173, g:216, b:230)); // Light Blue
            } else {
                g.setColor(new Color(r:240, g:248, b:255)); // Alice Blue
            }
            g.fillRect(col * CELL_SIZE, row * CELL_SIZE, CELL_SIZE, CELL_SIZE);
            if (board[row][col] == 1) {
                g.drawImage(queenImage, col * CELL_SIZE + 10, row * CELL_SIZE + 10, CELL_SIZE - 20, CELL_SIZE - 20, this);
            }
        }
    }
}
```

# placeQueens(int queens)

```
public void placeQueens(int queens) {  
    new Thread(() -> {  
        if (solveNQueens(row:0, queens)) {  
            JOptionPane.showMessageDialog(this, message:"All solutions displayed");  
        } else {  
            JOptionPane.showMessageDialog(this, message:"No solutions found");  
        }  
    }).start();  
}
```

# solveNQueens(int row, int queens)

```
private boolean solveNQueens(int row, int queens) {
    if (queens == 0) {
        int[][] solution = new int[size][size];
        for (int i = 0; i < size; i++) {
            solution[i] = board[i].clone();
        }
        displaySolution(solution);
        return true;
    }

    if (row >= size) {
        return false;
    }

    boolean foundSolution = false;
```

```
    for (int col = 0; col < size; col++) {
        if (isSafe(row, col)) {
            board[row][col] = 1;
            repaint();
            try {
                Thread.sleep(DELAY);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            foundSolution |= solveNQueens(row + 1, queens - 1);

            board[row][col] = 0;
            repaint();
            try {
                Thread.sleep(DELAY);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
    return foundSolution;
}
```



# isSafe(int row, int col)

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

# displaySolution(int[][] solution)

```
private void displaySolution(int[][] solution) {  
    JFrame frame = new JFrame(title:"N-Queens Solution");  
    NQueensVisualizer visualizer = new NQueensVisualizer(size);  
    visualizer.board = solution;  
    frame.add(visualizer);  
    frame.setSize(size * CELL_SIZE, size * CELL_SIZE);  
    frame.setVisible(true);  
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);  
    try {  
        Thread.sleep(Delay * 5);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

# Features

- ▶ **Interactive Input**
- ▶ **Graphical Visualization**
- ▶ **Animation of the Algorithm**
- ▶ **Multiple Solutions Display**


# Testing and Results

## Test Cases


Several test cases were designed to evaluate the functionality and correctness of the N-Queens Visualizer:

- ▶ **Basic Test Case:** Verify correct placement of queens on a small board size.
- ▶ **Edge Cases:** Test with minimal board size (1x1) and maximal board size (e.g., 15x15).
- ▶ **Invalid Inputs:** Test with invalid inputs such as negative board size or number of queens.
- ▶ **Performance Test:** Evaluate the visualizer's responsiveness and animation smoothness for larger board sizes (e.g., 8x8, 10x10) and varying numbers of queens.

Input ✕


 Enter board size:

Input ✕

 Enter number of queens:



Message ✕

 All solutions displayed

# Conclusion

The N-Queens Visualizer project aimed to create an interactive tool using Java Swing to solve and visualize solutions to the N-Queens problem. It provided users with a graphical representation of the chessboard and animated the backtracking algorithm's process of placing queens.

Thank you