Jayden Younger

Ian Riley

Fnd Algorithm Comp Appl

6 May 2024

Project 1 lab report

the NQueens problem

- **What is the NQueens' problem?**

  Let's ask the question, How can N queens be placed on an NxN chessboard so that no two of them attack each other? The Nqueens' problem is that we have a digital board that N ✕ N board size where we placed a number of queens on the board till no queen can be attacked.

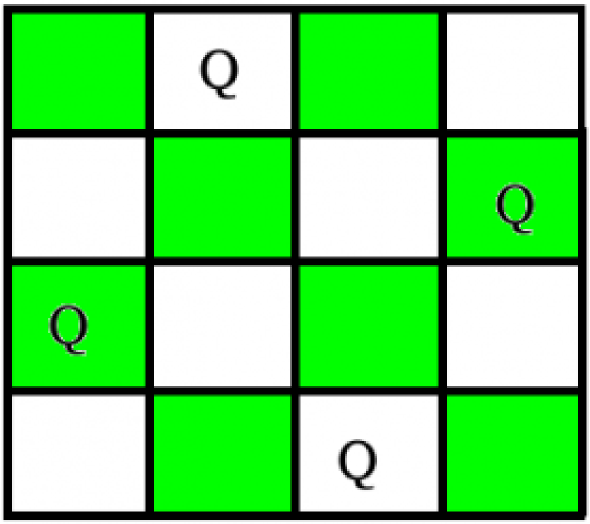


Image from geeksforgeeks.org/printing-solutions-n-queen-problem/

- **How does it work?**

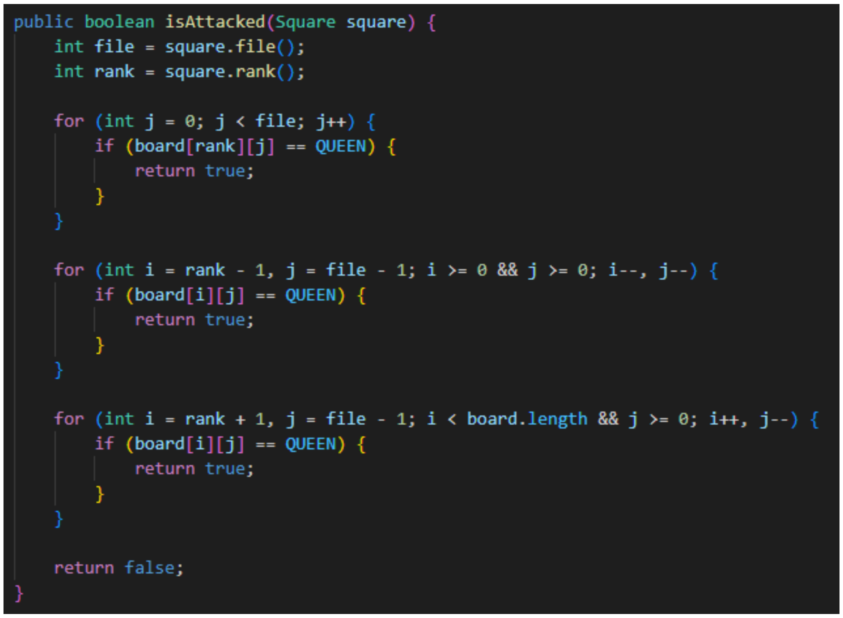
  We will place several queens on a board and move them around until none of them can be attacked by any other queen. Each queen will have a unique ID to identify it. First, we will place the queens on the board and perform a check to see if any queen can attack another. If a queen is under threat, we will move it to a new location where it is no longer in danger. This process will be repeated until all queens are placed in safe positions.

- Performance: O(N!), since the performance of the board is determined by the size of the board and the number of queens. Larger boards and too many queens can slow down the algorithm.

Methods

- **isAttacked()**

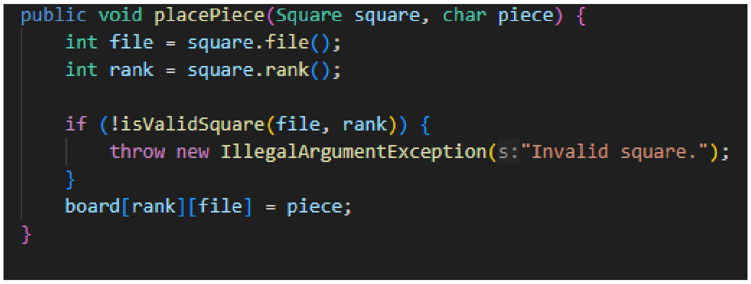
The isattacked method returns a bool value that indicates if a square has been attacked we used it in our SolveNQueens method to continuously check the board for a queen. It works by iteratively checking the board horizontally, vertically, and diagonally. When a queen is detected we return the method as true.



```
public boolean isAttacked(Square square) {
    int file = square.file();
    int rank = square.rank();

    for (int j = 0; j < file; j++) {
        if (board[rank][j] == QUEEN) {
            return true;
        }
    }

    for (int i = rank - 1, j = file - 1; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == QUEEN) {
            return true;
        }
    }

    for (int i = rank + 1, j = file - 1; i < board.length && j >= 0; i++, j--) {
        if (board[i][j] == QUEEN) {
            return true;
        }
    }

    return false;
}
```

- **placePiece()**

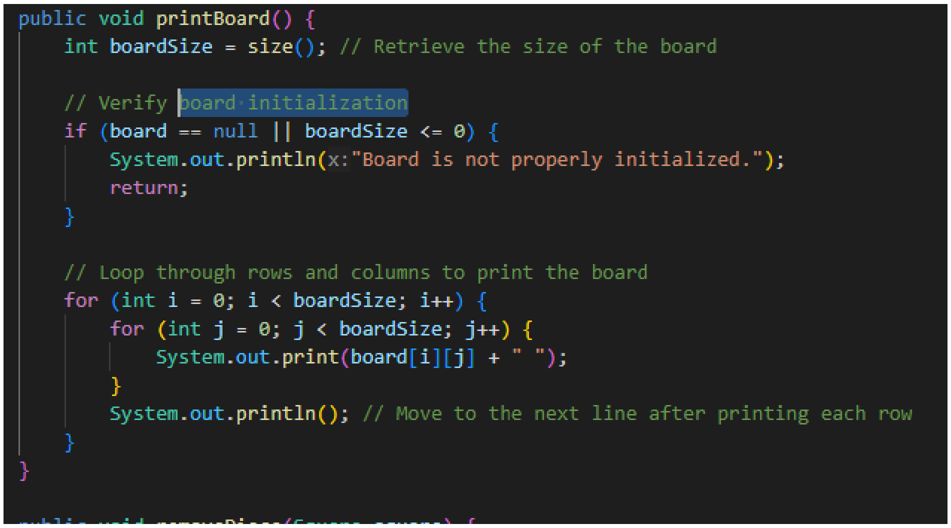
This method places a queen on a board if the square is valid.



```
public void placePiece(Square square, char piece) {
    int file = square.file();
    int rank = square.rank();

    if (!isValidSquare(file, rank)) {
        throw new IllegalArgumentException(s:"Invalid square.");
    }
    board[rank][file] = piece;
}
```

- **printBoard()**

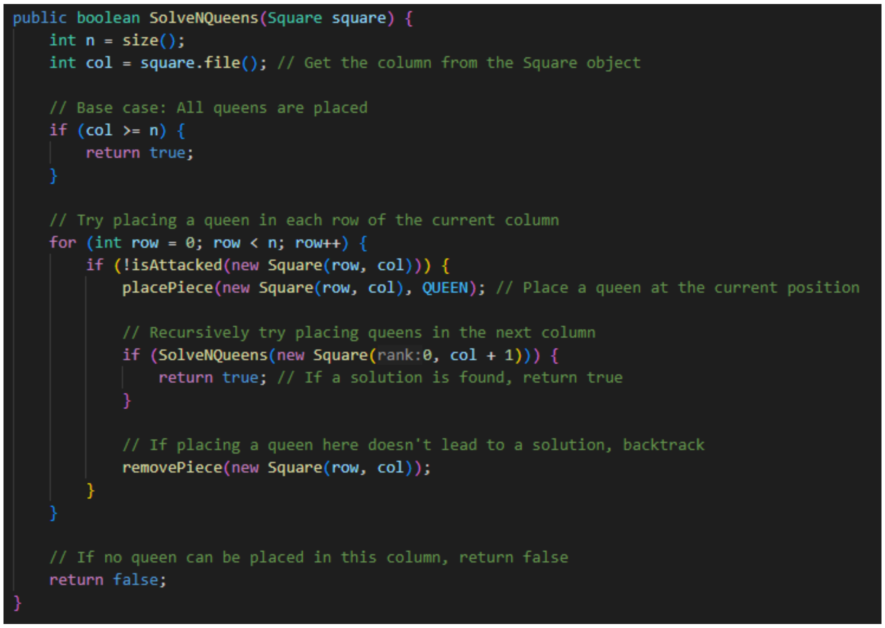
This method prints our board with our queens in place. It first checks our board initialization and verifies it has been initialized. We next do a for loop that prints our board.



```java
public void printBoard() {
    int boardSize = size(); // Retrieve the size of the board

    // Verify board initialization
    if (board == null || boardSize <= 0) {
        System.out.println(x:"Board is not properly initialized.");
        return;
    }

    // Loop through rows and columns to print the board
    for (int i = 0; i < boardSize; i++) {
        for (int j = 0; j < boardSize; j++) {
            System.out.print(board[i][j] + " ");
        }
        System.out.println(); // Move to the next line after printing each row
    }
}

public void removePiece(Square square) {
```

- **SolveNQueens()**

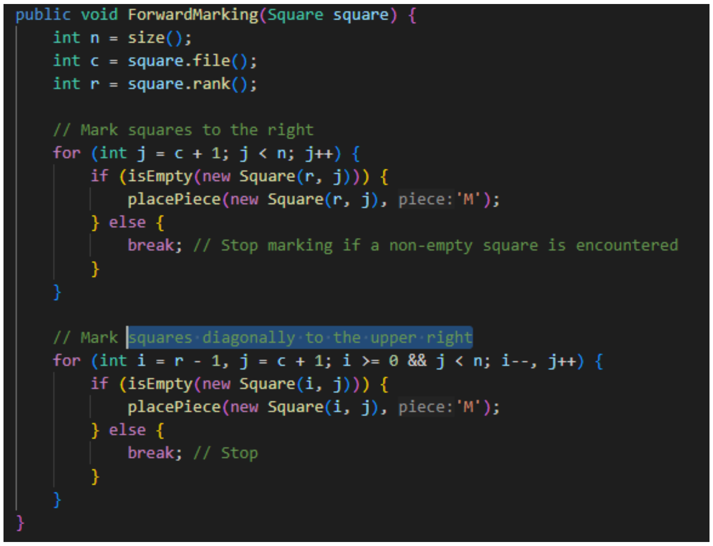
The SolveNQueens solves the solution by placing the queen recursively and when the board is solved we will return the method bool value to true.



```
public boolean SolveNQueens(Square square) {
    int n = size();
    int col = square.file(); // Get the column from the Square object

    // Base case: All queens are placed
    if (col >= n) {
        return true;
    }

    // Try placing a queen in each row of the current column
    for (int row = 0; row < n; row++) {
        if (!isAttacked(new Square(row, col))) {
            placePiece(new Square(row, col), QUEEN); // Place a queen at the current position

            // Recursively try placing queens in the next column
            if (SolveNQueens(new Square(rank:0, col + 1))) {
                return true; // If a solution is found, return true
            }

            // If placing a queen here doesn't lead to a solution, backtrack
            removePiece(new Square(row, col));
        }
    }

    // If no queen can be placed in this column, return false
    return false;
}
```

- **ForwardMarking()**

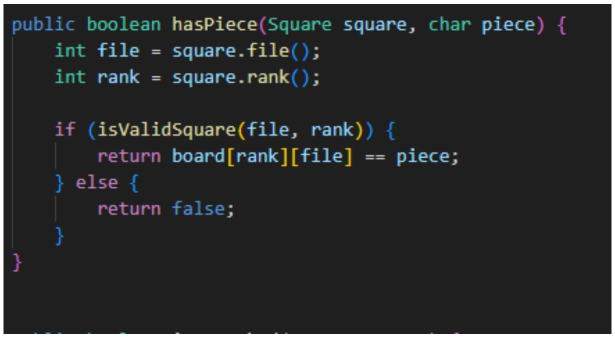
We Mark squares to the right and the squares diagonally to the upper right to check for a queen.



```java
public void ForwardMarking(Square square) {
    int n = size();
    int c = square.file();
    int r = square.rank();

    // Mark squares to the right
    for (int j = c + 1; j < n; j++) {
        if (isEmpty(new Square(r, j))) {
            placePiece(new Square(r, j), piece:'M');
        } else {
            break; // Stop marking if a non-empty square is encountered
        }
    }

    // Mark squares diagonally to the upper right
    for (int i = r - 1, j = c + 1; i >= 0 && j < n; i--, j++) {
        if (isEmpty(new Square(i, j))) {
            placePiece(new Square(i, j), piece:'M');
        } else {
            break; // Stop
        }
    }
}
```

- **hasPiece()**

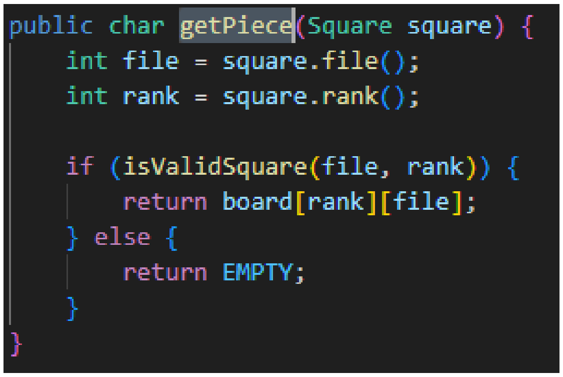
The method returns a bool value that checks for pieces at a given square, if there is a piece at a square, we return true.



```java
public boolean hasPiece(Square square, char piece) {
    int file = square.file();
    int rank = square.rank();

    if (isValidSquare(file, rank)) {
        return board[rank][file] == piece;
    } else {
        return false;
    }
}
```

- **getPiece()**

Returns the piece at a given given square, if no piece is in the square, we return it as an empty square.



```
public char getPiece(Square square) {
    int file = square.file();
    int rank = square.rank();

    if (isValidSquare(file, rank)) {
        return board[rank][file];
    } else {
        return EMPTY;
    }
}
```