PROJECT#3

Practice writing recursive solutions.

#	Name:		ID	Section
1	Ali Mostafa Almousa		1945427	BA

Instructors:
Dr. Abdulghani M. Al-Qasimi
Eng. Omar Alama

Contents

Introduction:

This lab introduces recursion as a programming technique in which the system stack is used implicitly to simplify some algorithmic problems. A class of problems can use a stack structure for backtracking; among these is the classical n-queens problem, where it is required to place n queens on an n by n chess board such that no two queens can attack each other. Many approaches exist for this problem, however, in this lab a recursive algorithm is utilized to solve the n-queens problem. Moreover, the solution steps should be animated graphically on a chess board displayed on the screen.

Procedure:

The approach followed to solve this problem is simple as it's not making any smart moves instead it's acting in a brute force manner attempting all possible moves until all queens are placed in the board without being able of attacking one another. Basically, the algorithm starts from the bottom row of the board by placing a queen in the first cell then moving to the upper row and searches for a safe spot for the next queen. In case there is no possible spot for a queen in the given row, the algorithm will backtrack to the previous queen in the lower row and change its position. This procedure is repeated until all n queens are placed in the n by n board in which all queens are incapable of attacking one another.

Pseudocode:

```
1) Start in the leftmost column of the bottom row
2) If all queens are placed
    return true
3) Try all columns in the current row.
   Do following for every tried column.
    a) If the queen can be placed safely in this column
       then mark this [row, column] as part of the
       solution and recursively check if placing
       queen here leads to a solution.
    b) If placing the queen in [row, column] leads to
       a solution then return true.
    c) If placing queen doesn't lead to a solution then
       unmark this [row, column] (Backtrack) and go to
       step (a) to try other columns.
4) If all columns have been tried and nothing worked,
   return false to trigger backtracking.
```

Methods:

Solve method is used to show in animation on the generated n by n chess board, how a recursive algorithm can find a solution to the n-queens problem. It first crate a matrix of the same size of the chess board, in which 0 denotes an empty cell and 1 denotes a cell with a queen. Next, n queens are drawn on top of each other in the first column to the left. Finally, the backtracking method is invoked to start the implementation of the solution.

solve:

```
public static void solve(int n) {
        // create a matrix to keep track
        // of queens on the board
        // 0 --> empty cell
        // 1 --> Queen
        int[][] board = new int[n][n];
        // Initialize the board with empty cells
        for (int[] arr : board) Arrays.fill(arr, 0);

        // Draw queens on leftmost column
        for (int i = 0; i < n; i++) cb.drawQueen(0, i, 0);
        cb.delay(d);

        // start positioning the queens
        // refresh the display if solved
        if (backtracking(board, board.length - 1) == true ) cb.display();
        // otherwise close graphical system with
        // false flag
        else cb.finish(false);
}
```

Backtracking is a utility function used to solve the n queen problem recursively following the procedure described above.

Backtracking:

```
/**
 * A recursive utility function to solve N
 * Queen problem by placing queens on board
 * starting from the bottom row and testing
 * every columns for safe spot
 * @param board is a matrix representing the chess board
 * @param row is the staring row for placing queens
 * @return true if a solution is found and false otherwise
 */
private static boolean backtracking(int[][] board, int row) {
        /* base case: If all queens are placed
        then return true */
        if (row < 0)
                return true;

        /* Consider this row and try placing
        this queen in all columns one by one */
        for (int i = 0; i < board.length; i++) {
                /* Check if the queen can be placed on
                board[row][i] */
                if (isValid(board, row, i)) {
                        /* Place this queen in board[row][i] */
                        board[row][i] = 1;
                        cb.drawQueen(i + 1, (board.length - 1) - row, 0);
                        cb.delay(d);

                        /* recur to place rest of the queens */
                        if (backtracking(board, row - 1) == true)
                                return true;

                        /* If placing queen in board[row][i]
                        doesn't lead to a solution then
                        remove queen from board[row][i] */
                        board[row][i] = 0; // BACKTRACK
                        cb.clearQueen(i + 1, (board.length - 1) - row);
                }
        }

        /* If the queen can not be placed in any column in
        this row, then return false */
        return false;
}
```

IsValid acts as the checker of the backtracking algorithm as it receives a particular position in the board and returns true or false according to the safety of placing a queen at this position. Ture indicates a safe cell to move to while false denotes that this cell is under attack from another queen.

isValid:

```
/**
 * A utility function to check if a queen can
 * be placed on board[row][col]. This
 * function is called when "row" queens are
 * already placed in rows from row - 1 to 0.
 * So we need to check for attacking queens
 * form the following sides only:
 * - Lower rows at the same column
 * - Lower left diagonal
 * - Lower right diagonal
 * @param board is a matrix representing the chess board
 * @param row is the row of the cell to test
 * @param col is the column of the cell to test
 * @return true if this cell is safe and false otherwise
 */
static boolean isValid(int board[][], int row, int col){
        int i, j;
        // clear the queen to move
        // from the leftmost column
        cb.clearQueen(0, (board.length - 1) - row);
        cb.delay(d);

        // draw a queen on current
        // cell being tested
        cb.drawQueen(col + 1, (board.length - 1) - row, 0);

        /* Check this col on from bottom */
        for (i = board.length - 1; i > row; i--)
                if (board[i][col] == 1) {
                        // flash the attacking and
                        // attacked queens and keep
                        // the attacking one on board
                        flash2QueensKTimes(3, i, col, row, col);
                        // Move the attacked queen
                        // back to leftmost column
                        cb.drawQueen(0, (board.length - 1) - row, 0);
                        return false;
                }

        /* Check lower diagonal on right side */
        for (i = row, j = col; i < board.length && j < board.length; i++, j++)
                if (board[i][j] == 1) {
                        // flash the attacking and
                        // attacked queens and keep
                        // the attacking one on board
                        flash2QueensKTimes(3, i, j, row, col);
                        // Move the attacked queen
                        // back to leftmost column
                        cb.drawQueen(0, (board.length - 1) - row, 0);
                        return false;
                }
        /* Check lower diagonal on left side */
```

```java
        for (i = row, j = col; j >= 0 && i < board.length; i++, j--)
                if (board[i][j] == 1) {
                        // flash the attacking and
                        // attacked queens and keep
                        // the attacking one on board
                        flash2QueensKTimes(3, i, j, row, col);
                        // Move the attacked queen
                        // back to leftmost column
                        cb.drawQueen(0, (board.length - 1) - row, 0);
                        return false;
                }

        return true;
}
```

The last method is a helper method used mainly for graphical purposes. It flashes both the attacking and attacked queens then removes the attacked one while keeping the attacking one on board.

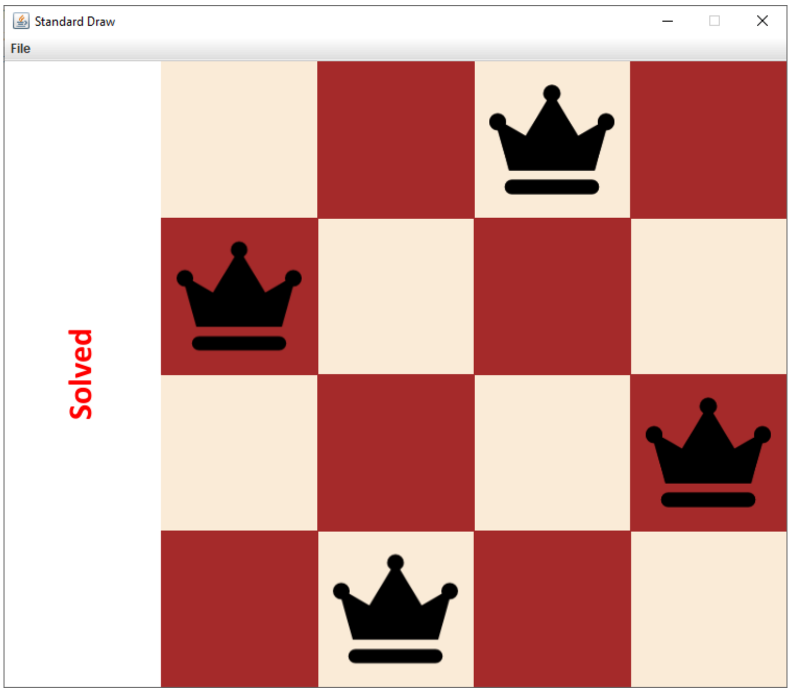
flash2QueensKTimes:

```java
/* This method will flash two queens k times then clear
 the queen under attack and keep the attacking queen
 in black color.
 (row1, col1) --> position of the attacking queen
 (row2, col2) --> position of the queen under attack*/
private static void flash2QueensKTimes(int k, int row1, int col1, int
row2, int col2) {
        // flash the 2 queens k times
        for (int i = 0; i < k; i++) {
                cb.clearQueen(col1 + 1, (n - 1) - row1);
                cb.clearQueen(col2 + 1, (n - 1) - row2);
                cb.delay(d);
                cb.drawQueen(col1 + 1, (n - 1) - row1, 1);
                cb.drawQueen(col2 + 1, (n - 1) - row2, 1);
                cb.delay(d);
        }
        // clear the attacked queen
        cb.clearQueen(col2 + 1, (n - 1) - row2);
        cb.delay(d);

        // draw the attacking queen in black
        cb.clearQueen(col1 + 1, (n - 1) - row1);
        cb.drawQueen(col1 + 1, (n - 1) - row1, 0);
        cb.delay(d);

}
```

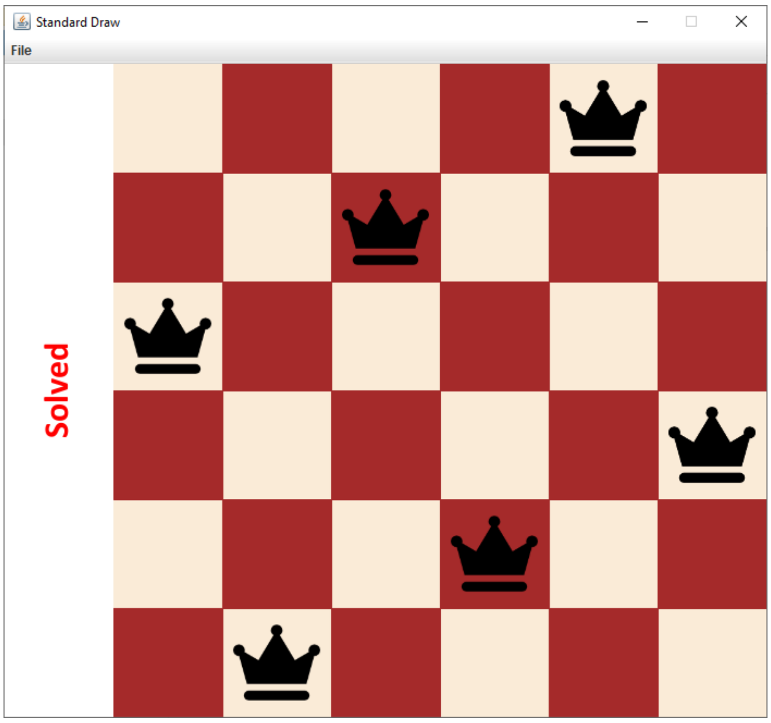
<u>Output Sample:</u>

For a video demonstration of the following output sample please refer to this link: https://youtu.be/X2w3Pp7lqGE

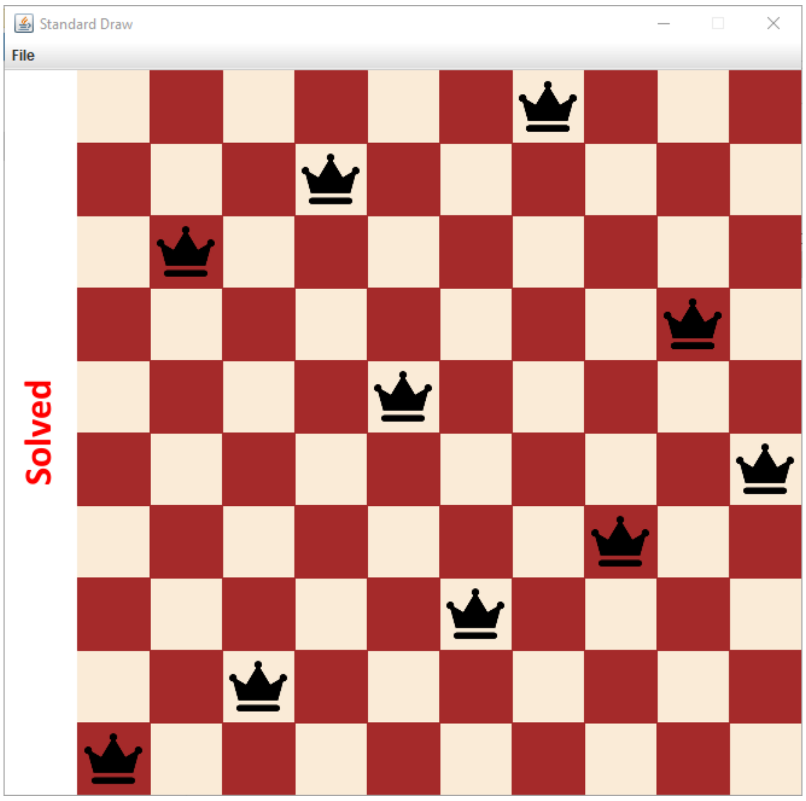
4 by 4 chess board



6 by 6 chess board



10 by 10 chess board



Conclusion:

This lab has demonstrated how to deal with recursion and how to use this technique to find a correct solution to a problem at hand, all while having the privilege of visually watching the algorithm move the queens and making correct decisions to reach the target state in which all queens are safely placed in the board.