The "8 Queens" problem

- Consider the problem of trying to place 8 queens on a chess board such that no queen can attack another queen.

- What are the "choices"?

- How do we "make" or "un-make" a choice?

- How do we know when to stop?
Naive algorithm

- for (each square on board):
  - Place a queen there.
  - Try to place the rest of the queens.
  - Un-place the queen.

- How large is the solution space for this algorithm?
  - $64 \times 63 \times 62 \times \ldots$
Better algorithm idea

- Observation: In a working solution, exactly 1 queen must appear in each row and in each column.

- Redefine a "choice" to be valid placement of a queen in a particular column.

- How large is the solution space now?
  - $8 \times 8 \times 8 \times \ldots$
Recall: Backtracking

A general pseudo-code algorithm for backtracking problems:

Explore(\textit{choices}): 
- if there are no more \textit{choices} to make: stop.
- else, for each available choice \textit{C}:
  - Choose \textit{C}.
  - Explore the remaining \textit{choices}.
  - Un-choose \textit{C}, if necessary. (backtrack!)
Exercise

- Suppose we have a `Board` class with these methods:

<table>
<thead>
<tr>
<th>Method/Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public <code>Board(int size)</code></td>
<td>construct empty board</td>
</tr>
<tr>
<td>public boolean <code>isSafe(int row, int column)</code></td>
<td>true if queen can be safely placed here</td>
</tr>
<tr>
<td>public void <code>place(int row, int column)</code></td>
<td>place queen here</td>
</tr>
<tr>
<td>public void <code>remove(int row, int column)</code></td>
<td>remove queen from here</td>
</tr>
<tr>
<td>public String <code>toString()</code></td>
<td>text display of board</td>
</tr>
</tbody>
</table>

- Write a method `solveQueens` that accepts a `Board` as a parameter and tries to place 8 queens on it safely.
  - Your method should stop exploring if it finds a solution.
// Searches for a solution to the 8 queens problem
// with this board, reporting the first result found.
public static void solveQueens(Board board) {
    if (solveQueens(board, 1)) {
        System.out.println("One solution is as follows:");
        System.out.println(board);
    } else {
        System.out.println("No solution found.");
    }
}
Exercise solution, cont'd.

// Recursively searches for a solution to 8 queens on this board, starting with the given column, returning true if a solution is found and storing that solution in the board. // PRE: queens have been safely placed in columns 1 to (col-1)
public static boolean solveQueens(Board board, int col) {
    if (col > board.size()) {
        return true;  // base case: all columns are placed
    } else {
        // recursive case: place a queen in this column
        for (int row = 1; row <= board.size(); row++) {
            if (board.isSafe(row, col)) {
                board.place(row, col);  // choose
                if (explore(board, col + 1)) {  // explore
                    return true;  // solution found
                }
                board.remove(row, col);  // un-choose
            }
        }
        return false;  // no solution found
    }
}
Graphical User Interfaces

- Involve large numbers of interacting objects and classes
  - Highly framework-dependent

- Path of code execution unknown
  - Users can interact with widgets in any order
  - Event-driven

- In Java, AWT vs. Swing; GUI builders vs. writing by hand
Swing Framework

- Great case study in OO design
Composite Layout

Draw out desired result

Divide into regions

Figure out appropriate layout managers and components