The *n*-queens problem

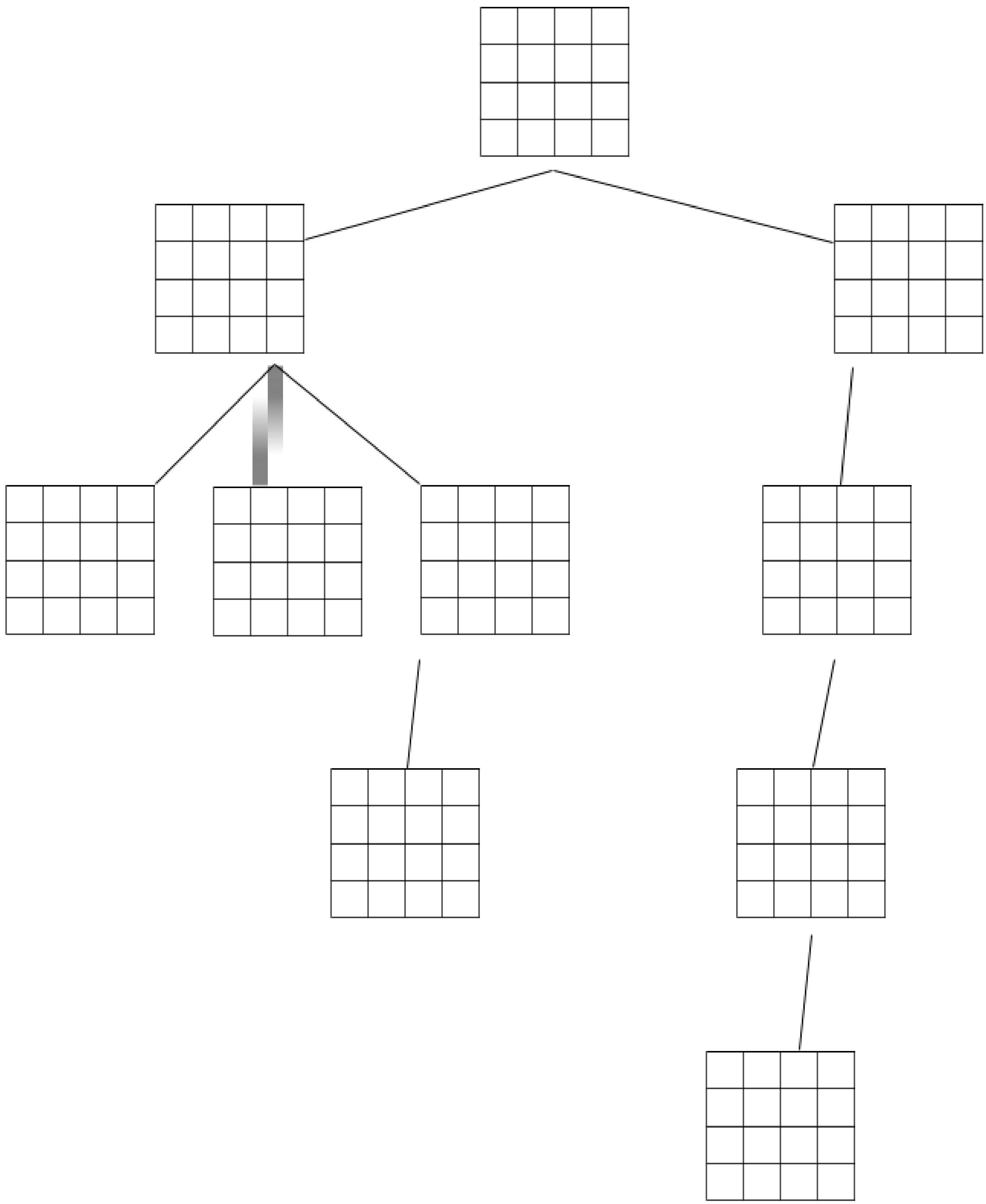
Given an $n \times n$ chessboard, place n queens on the board so that no two queens can attack each other (no two are in the same row, column or diagonal).

				X
	X			
			X	
X				
		X		

Solve the n-queens problem using a depth-first search and **backtracking**.

Partial solutions will be generated systematically and "stored" in a **search tree**. (A tree data structure is not actually created, but the recursive algorithm can be traced by viewing it as a search tree.) The order of inspecting various partial solutions is determined by depth first search.

- root = board with no queens
- place queens successively in each column of the board (beginning in the left column), working top to bottom
- when you place a queen in a new column, that arrangement is a child of the previous in the search tree
- arrangements at level i of the search tree will contain i queens, placed within the first i columns
- when you run out of possible placements for another queen, backtrack to the parent and adjust the queen in the preceding column by moving it down one row



`rn_queens(k, n)`

- Assumption: queens in columns `1` to `k-1` are properly placed (non-attacking)
- an attempt is made to place a queen in column `k`, checking that the queen in the previous columns are not in the same row or diagonal
- recursively call **`rn_queens(k+1, n)`** to place a queen in the next column

`row[i]` contains the row number (the index) of the queen in column `i`

Example: row[1] = 1, row[2] = 3 corresponds to the following arrangement

X			
	X		

NOTE: As with other algorithms in this class, indexing is from 1, not 0.

```
n_queens(n){
  rn_queens(1, n)
}

rn_queens(k, n)
{
  for j = 1 to n        // try putting the kth queen in each
  {                     // of rows 1 .. n
    row[k] = j
    if (position_ok(k, n, row))
    {
       if (k == n)          // all n queens placed => all done!!!
       {
          for i = 1 to n
             print (row[i] + " ")
          println()
       }
       else
         rn_queens(k+1, n)  // not done => place the next queen
    }
  }
}

position_ok(k, n, row)
{ // check that none of the previous k-1 queens are attacking
  // (no queen is in the same row or same diagonal)
}
```

```
position_ok(k, n, row)
{
  for i = 1 to k-1
    if (row[k] == row[i]  || abs(row[k]-row[i]) == (k-i))
      return false
  return true
}
```

(The absolute value thing for checking the diagonal is pretty sneaky.)
NOTE: This implementation continues to process and outputs ALL possible solutions.

Analyze the running time (worst case)

How much time as a function of *n*?
depending on implementation of position_ok
if position_ok uses a scan for i = 1 to k-1

upper bound:

each recursive call tries a new placement and calls on position_ok
just focusing on rows (and ignoring diagonals)
column 1: for EACH of the n possible placements
column 2: for EACH of the n – 1 possible placements
column 3: for EACH of the n – 2 possible placements
etc.
column n: 1 possible placement

=> time for position_ok * (n * n-1 * n-2 * … * 2)
= O(n * n!) (Or maybe O(1 * n!) if you're slick about position_ok)

OR

Analyze the running time using a recurrence. T(n) is the amount of time to place all n queens.
Worst case: T(n) = n * T(n-1), T(1) = 1
Solve this recurrence using the Iteration Method

$$
\begin{aligned}
T(n) &= n * T(n-1) \\
&= n * (n-1) * T(n-2) \\
&= n * (n-1) * (n-2) * T(n-3) \\
&\quad \text{etc.} \\
&= n * (n-1) * (n-2) * \ldots * 2 * 1 = n!
\end{aligned}
$$