More Recursion: NQueens

- continuation of the recursion topic
  - notes on the NQueens problem
  - an extended example of a recursive solution
Recursion & Backtracking

- backtracking
  - an algorithmic tool
  - used in artificial intelligence (& other) programs

- problems where a backtracking strategy works
  - when there are many different
    - possible solution paths
    - each consisting of a sequence of steps
      - from a start state to a solution state
    - & it is not known which path is optimal
Recursion & Backtracking

- the backtracking strategy
  - systematically, recursively builds a path
    - out of a sequence of choices
  - if a solution cannot be found on the current path
    - then undo the last step: \textit{backtrack}
    - & try an alternative path
  - note: the backtracking may
    - go all the way back to the \textit{first} step in the process
Recursion & Backtracking
Recursion & Backtracking

- an example
  - the NQueens problem
  - a solution consists of:
    - placing n queens on an n x n chessboard
    - so that no queen "threatens" conflicts with any other
    - so, only 1 queen per column, row, & diagonal

- recursive backtracking solution follows
  - recursion is a necessary part of such an algorithm
  - makes it much easier to write
NQueens

- constraint for the NQueens problem
  - each queen in a separate column, row & diagonal
- example: single Queen on 8x8 grid (chessboard)
  - & who she threatens, potential conflicts
one sample solution for the 8-Queens problem on an 8x8 grid, no queen threatens another.

how many solutions are there? what are they?
NQueens

- sample algorithm to find one solution
  - provided the problem has a solution
    - other algorithms might find all solutions
  - uses recursion & backtracking
  - it is relatively easy to solve this problem for small n
    - for the example using n=4, can show each step
      - break out to 4QueensDemo
  - watch for the backtracking!
NQueens

- solving the nqueens problem
  - number the rows & columns from zero
  - note that only one Queen can occupy each column
  - therefore each column \textit{must} have a Queen
    - move across the grid, column by column
    - place a queen in each column
  - start from column zero & go to column n-1
    - place the queen for the current column in a row & diagonal
    - such that she doesn't threaten previously placed queens
solving the nqueens problem
  pseudocode for a recursive method
    assumes placing queens using a Board object
    full code of the method on the next slide

// board size n x n
boolean solveNQ (int col)
  if col >= size then all done!
  for row 0 to row n-1
    if (row, col) is a safe (non-threatened) position
      place a Queen at (row, col)
      if solveNQ (col + 1) is true then // recursive step
        return true
      else
        remove Queen from (row, col) // backtracking step
  (Outside of loop:) return false
NQueens

public static boolean solveNQ(Board bd, int col) {
    // anchor/base case: successful solution
    if (col >= bd.getSize()) return true;

    // try putting a queen in each row of the current column
    for (int row = 0; row < bd.getSize(); row++) {
        if (safePosition(bd, row, col)) {
            bd.putQueen(row, col);
            if (solveNQ(bd, col+1)) { // recursive step
                return true;
            } else {
                bd.removeQueen(row, col); // backtrack step
            }
        }
    }

    // anchor/base case: there is no solution
    return false;
} // end solveNQ
solving the nqueens problem

Board.java
- a class used to represent an n x n board
  - fixed size (8) in the sample implementation
- stores locations of queens
  - allows checking for occupied locations, board size
  - allows removal of a queen, display of board status

NQueenRecursive.java
- the application program includes
- the recursive backtracking solution method
- a method to check for threats/conflicts
- & uses a Board object to place queens, display result