## Recursive evaluation of $x^n$

Evaluate $x^n$ where n is a nonnegative integer

**Iterative Solution**

Multiply x by itself n times

O(n) algorithm

```
double power(double x, int n) {
   if (n == 0)
      return 1;
   else {
      double p = power(x,n/2);
      if (n % 2 == 0)
        return p * p;
      else
        return x * p * p;
   }
}
```

**Recursive Solution**

$x^0 = 1$
$x^n = (x^{n/2})^2$ n > 0 and even
$x^n = x\,(x^{(n-1)/2})^2$ n > 0 and odd

$O(\log_2 n)$ algorithm

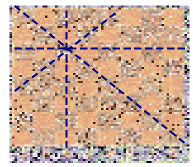
## The 8 Queens Problem

Arrange 8 queens on a standard 8 × 8 chessboard so that none of the queens attack each other.

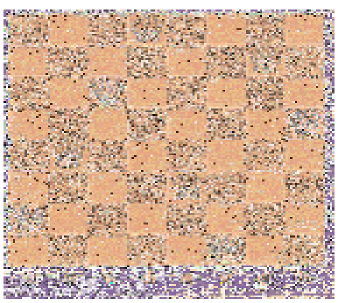
More general problem is to arrange n (n $\geq$ 1) queens on an n × n chessboard so that none of the queens attack each other.

A queen placed at a particular square attacks any other queen placed

- in the **same row**
- in the **same column**
- along the **two diagonals** passing through the square

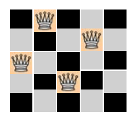


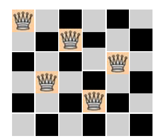
## A Particular Solution to the 8 Queens Problem



This is one of 92 solutions to the problem. 12 of these solutions are distinct; the others are rotations and reflections of each other.

## Solution to n Queens Problem





**Solution?**

Assume that queens have been correctly placed in rows 1 .. n-1. Then look at each column in row n to find where to place the queen in row n.

**Correctness?**

Base case : Yes - empty chessboard
Smaller caller : Yes - problem reduces to base case
General case : No - no guarantee that a solution for n-1 rows leads to one for n rows, or even that a solution exists at all!

**Backtracking**

If no solution exists for row n, "backtrack" to row n-1 and try to find a new column in which to place the queen, and then try row n again. If no column is suitable in row n-1, backtrack to row n-2, etc. This is now guaranteed to find a solution if one exists.

## Recursive Backtracking Algorithm for n Queens Problem

**Algorithm** *addQueen*(row)

Assume that queens have been validly placed in rows 1 .. n-1. Then add a validly placed new queen to the board in row n, if this is possible. If not, backtrack to previous row(s) and try again.

```
for col = 1 to 8
   if no queen attacks square (row,col) {

     place a queen at (row,col)

     if row = 8
       display positioned queens        // solution found!
     else
       addQueen(row+1)                  // try and position next queen in next row

     remove the queen at (row,col)      // either we couldn't position in next row and we
                                           need to backtrack, or we found a solution and
                                           we want to backtrack anyway to find
                                           another one
 }
```

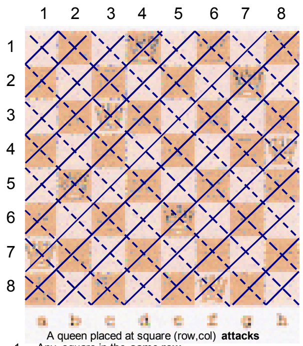
## Problems

- How do we represent the positions of the queens on the board?
- How do we determine whether two queens are attacking each other, i.e. are

  in the same row or column or

  on the same diagonal?

# Squares Queen Attacks




**Up-diagonal**

All (row,col) squares on the **same up-diagona**l have the **same** value for
**row + col**
in the range
**2 .. 16**

In general, for n queens, range is
**2 .. 2n**

**Down-diagonal**

All (row,col)squares on the **same down-diagonal** have the **same** value for
**row - col**
in the range
**-7 .. 7**

In general, for n queens, range is
**-(n-1) .. n-1**

A queen placed at square (row,col) **attacks**

1. Any square in the **same row**
2. Any square on the **same column**
3. Any square on the **same up-diagonal** passing through (row,col)
4. Any square on the **same down-diagona**l passing through (row,col)

# Position Representation

Use ***PArrays*** since array lower limits nonzero.

| ***PArray*** | **Range** | **Purpose** |
|---|---|---|
| *queenCol* | 1 .. n | Store positions of queens |
| *colStatus* | 1 .. n | Whether queen is located in col j |
| *upStatus* | 2 .. 2n | Whether queen is located on up-diagonal j |
| *downStatus* | -(n-1) .. n-1 | Whether queen is located on down-diagonal j |

**To position a queen at square (*row,col*)**

Set $queenCol_{row}$ to *col*
Set $colStatus_{col}$ to *ATTACKED*
Set $upStatus_{row + col}$ to *ATTACKED*
Set $downStatus_{row - col}$ to *ATTACKED*

**To check if a queen can be positioned at square (*row,col*)**

Check if any of
$colStatus_{col}$
$upStatus_{row + col}$
$downStatus_{row - col}$
are set to *ATTACKED*,
We don't need to check if there is a queen in row *row*. Our algorithm never tries to place more than 1 queen in a row anyway.