The N-Queens Puzzle

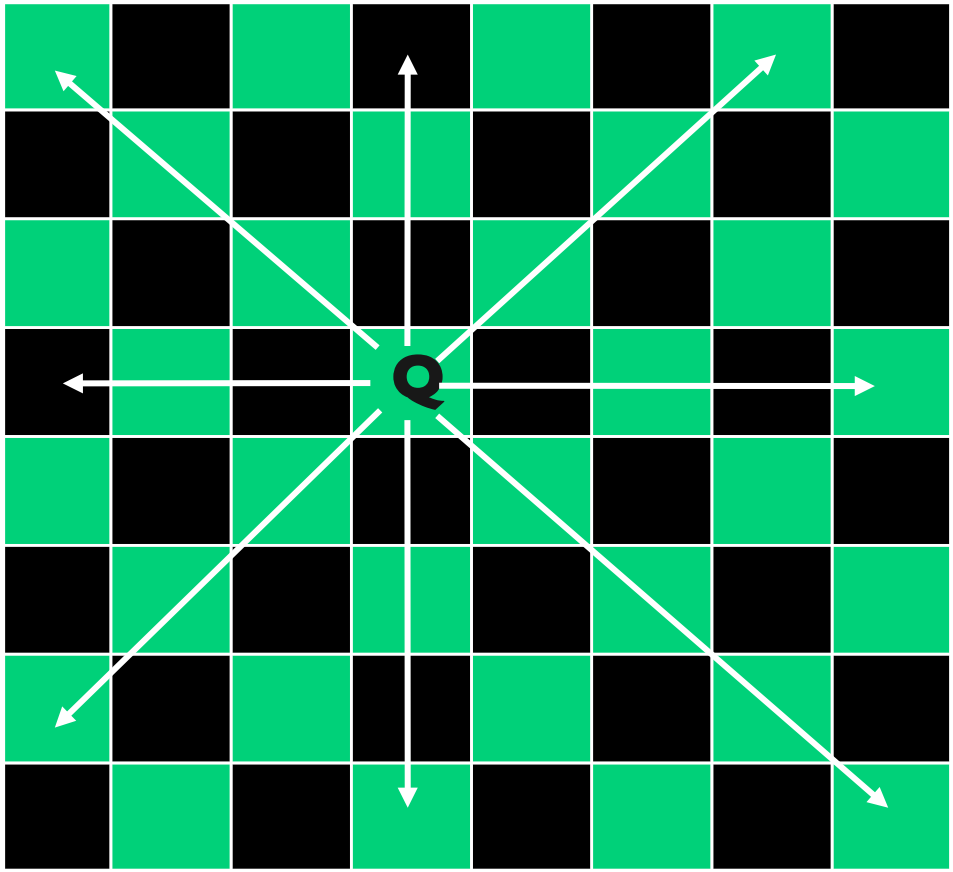
The eight queens puzzle is the problem of putting eight queens on an 8x8 chessboard such that none of them attacks any other.

We'll look at this puzzle in the next few slides, and then consider how a computer program can solve this problem.

Using the object-oriented approach, an elegant and intuitive solution to this problem can be found.

What is a legal Queen move?

The Queen can move in a straight line along any rank, file or diagonal, e.g.,



Placing Eight Queens on a Chessboard

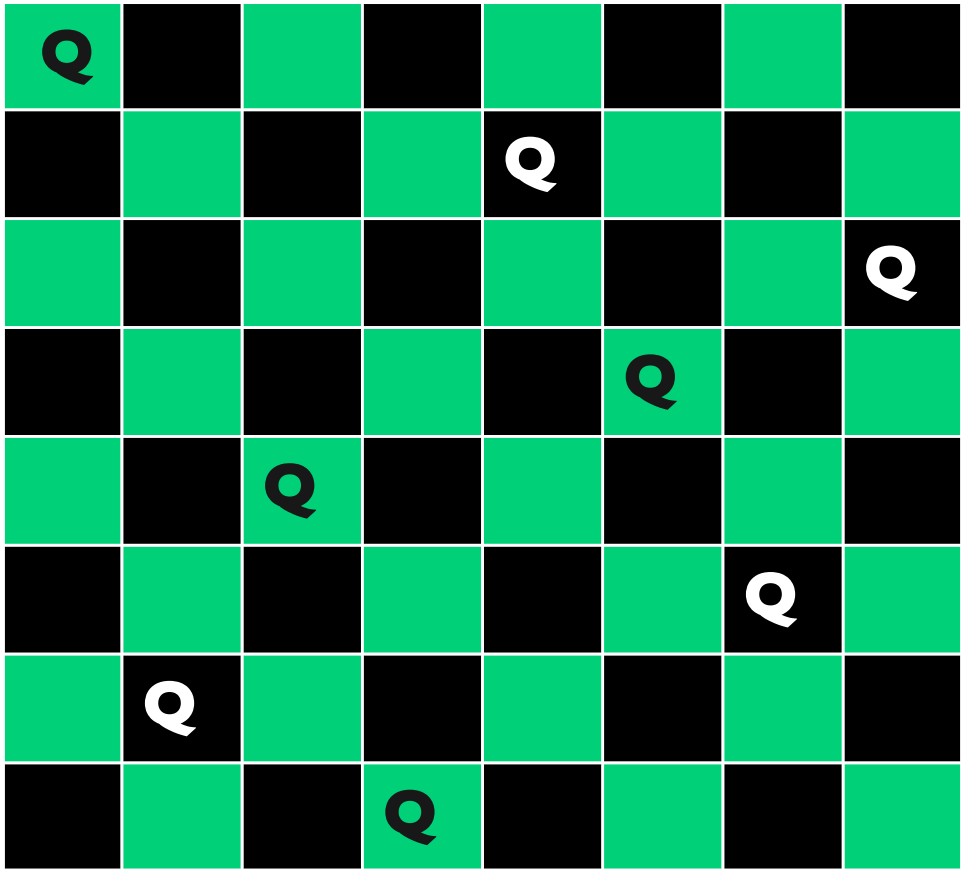
The problem is simply this: Place *eight* queens on an *8x8* chess board in such a way that no queen attacks any other queen.

Finding a solution by hand on a regular chess board is difficult, and it is not immediately apparent that this problem even admits a single solution.

There are in fact 92 solutions on a regular 8x8 chess board. 12 are *fundamental* solutions; the others are rotations and reflections of these.

One Possible Solution

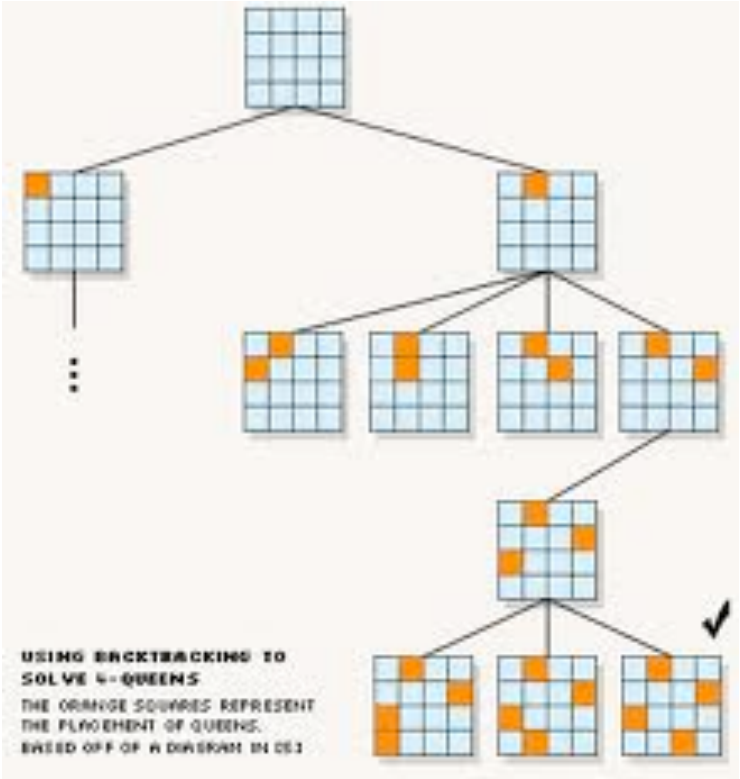
One solution on an 8x8 board is this:



The N-Queens Puzzle in Computer Science

The N-Queens puzzle is one of the classic problems in Computer Science. The N-Queens puzzle obviously involves placing *n* queens on an *nxn* board such that no queen attacks another.

It is commonly solved by building a tree and performing a depth-first search, in what is termed a **backtracking** solution.



This type of solution is applicable to a large class of other problems.

An Object-Oriented Approach to N-Queens

In this section we look at an **object-oriented** approach to solving this puzzle. While backtracking is still involved, it's hidden in the interaction between the objects.

Our solution will mimic how you would solve the problem using real-people as Queens on an open-air chess board.

Marostica

The solution presented here is based from Timothy Budd's *Introduction to Object-Oriented Programming*, chapter 6.

An Object-Oriented Approach

In our object-oriented solution, we'll model the queens as objects and empower them to co-operate to solve the problem.

Some of the questions we have to ask ourselves then are:

- Does each queen need to know about all the other queens (worst case) or just one other queen (best case)?
- Exactly what behaviour does each queen have?

One vital observation is this:

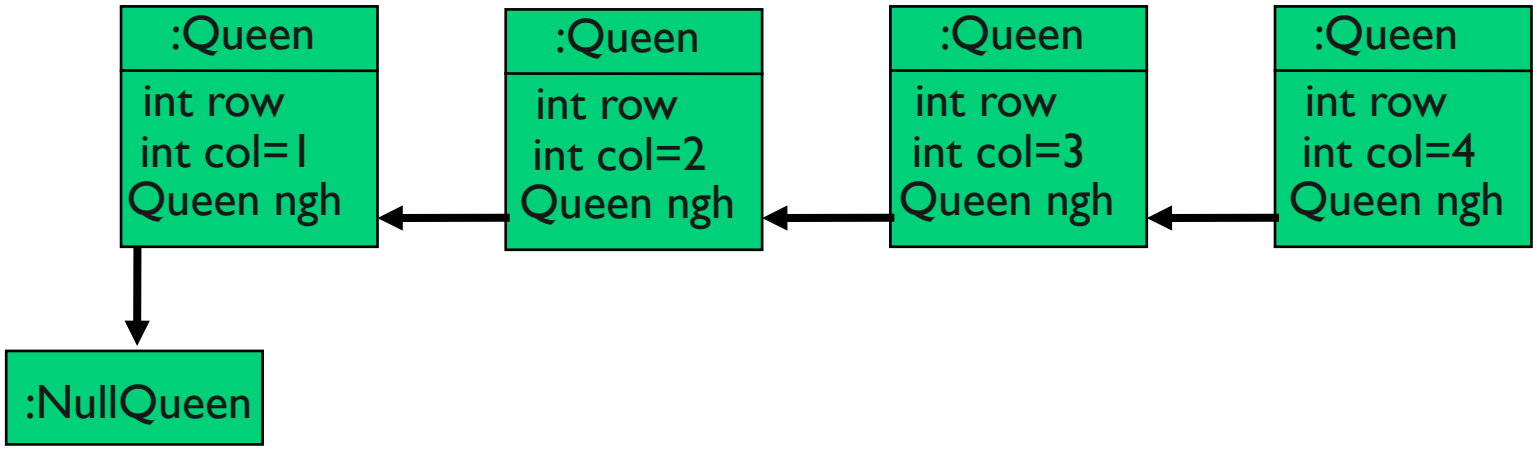
No two queens can occupy the same row or column.

=> We can assign each queen a **fixed column**, and just allow the row to vary.

The Queen Objects

Another observation is that each queen need only know about the queen to her immediate left.

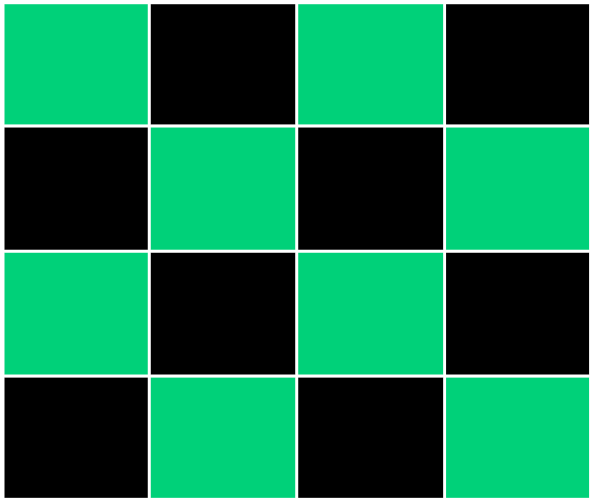
We can visualise the queen objects then like this (on a 4x4 board):



Consider the dynamics of how these queens will interact…

How the solution works...

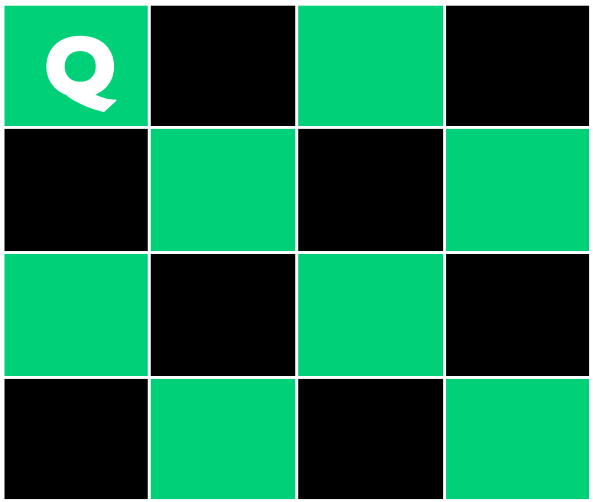
Consider how a solution in found on an 4x4 board like this:



We'll add Queens from left to right, and ensure the list is in a **partial solution** before we add the next Queen.

Placing the first Queen

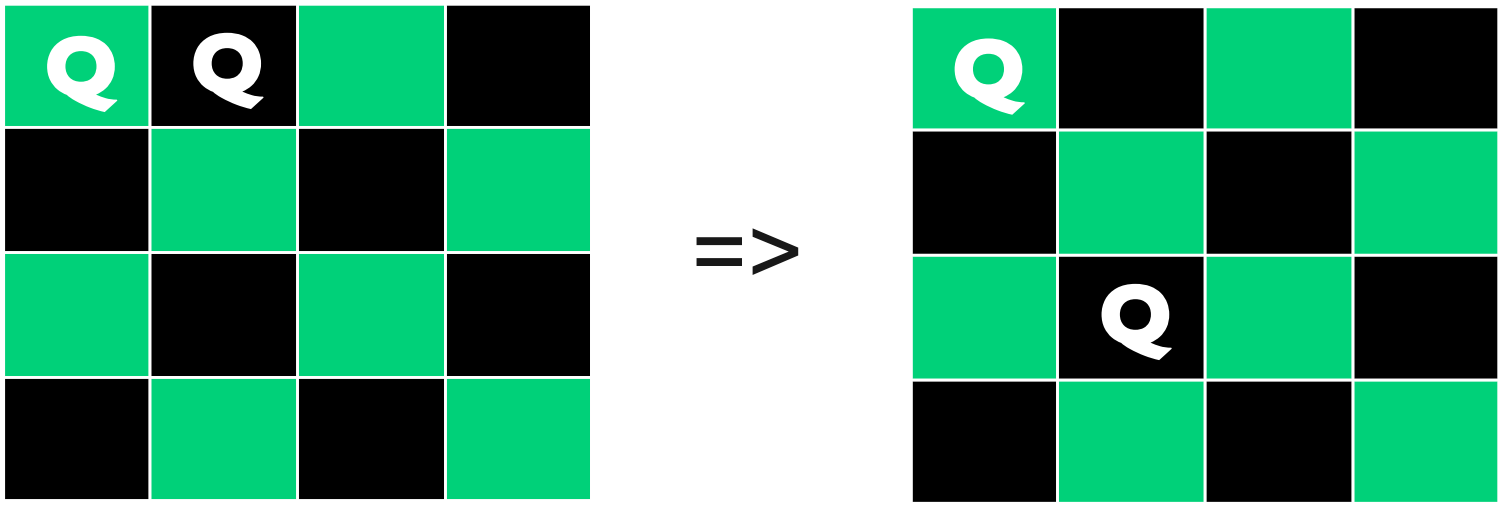
Placing the first queen is easy:



We achieve by invoking **find_solution** on this queen to make it find its "next" partial solution, which is trivial when there is only one queen on the board.

Placing the second Queen

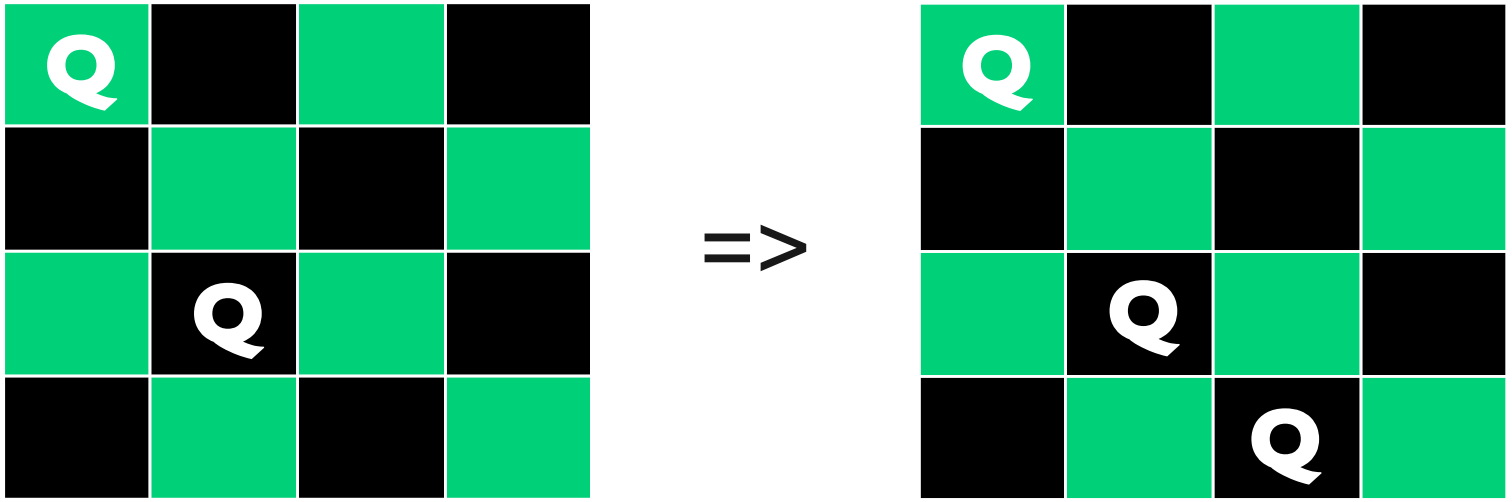
The second queen has to move twice to reach a partial solution:



So a queen object needs to be able to ask the queen next to it the question, **canYouAttackMe**?

The third Queen has no squares!

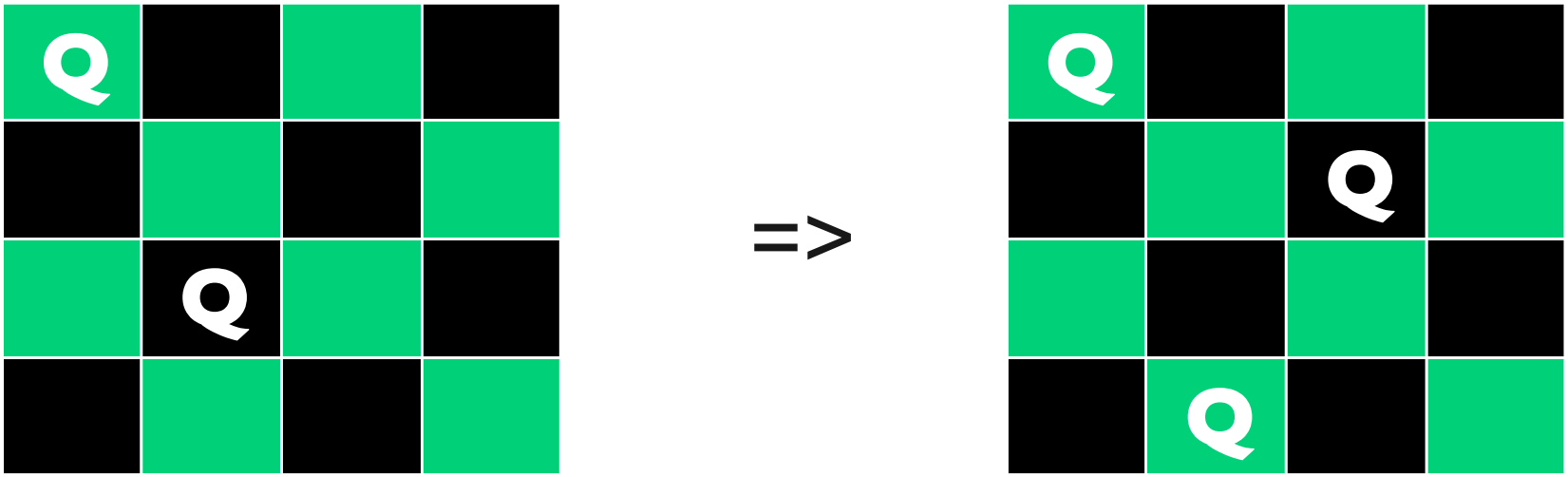
The third queen has to moves through all its possible squares but cannot reach a partial solution:



Now we know that a solution is impossible with the first two queens in their present position, so current queen asks its neighbour to **advance** to the next legal position.

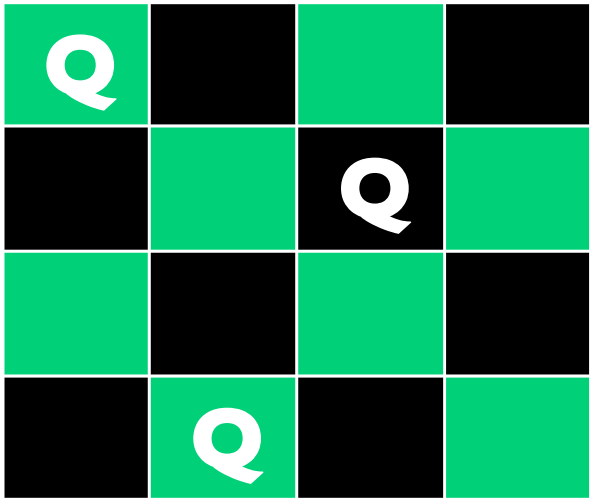
Advancing the first two Queens

After the first two queens advance, the third queen easily finds a square where there is a partial solution:



The 4th Queen has no squares

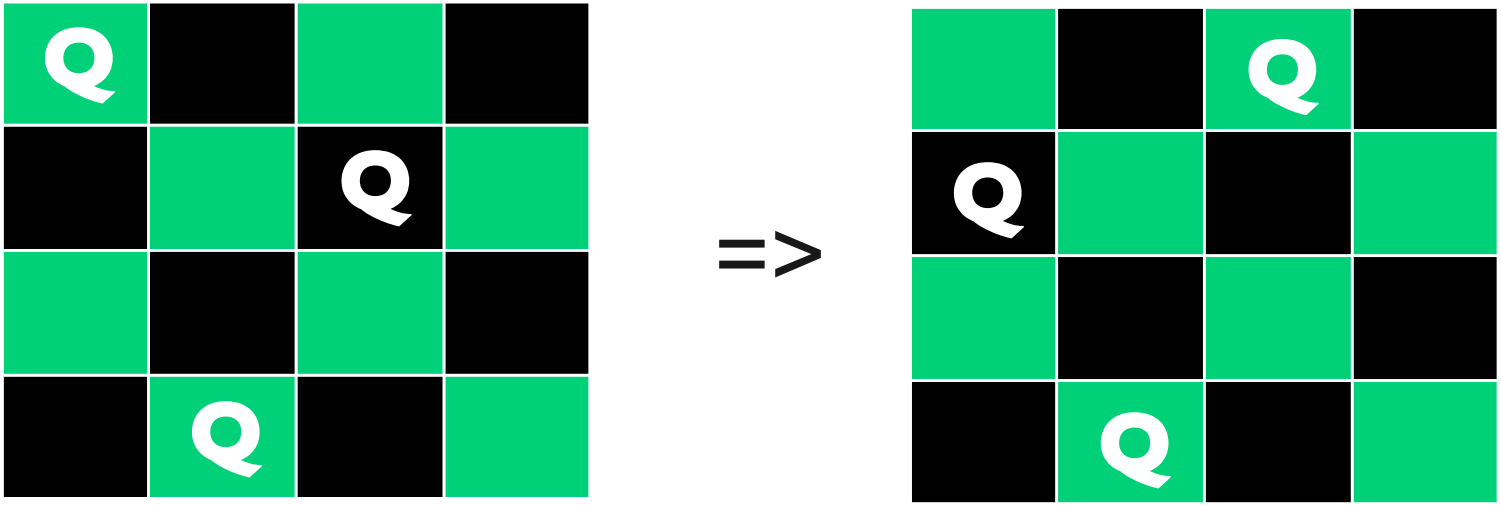
However, the fourth queen tries all options but finds no valid square:



so it asks its neighbours to **advance**...

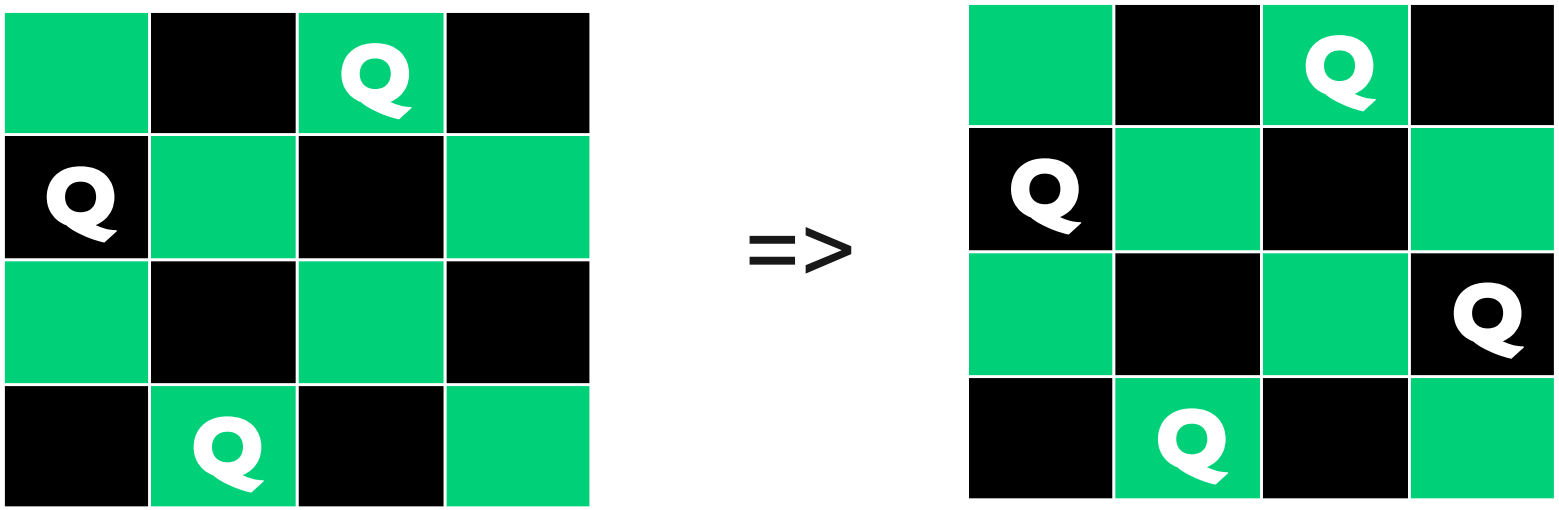
Advance the first 3 queens

The first three queens advance to their next legal position as follows:



Now the 4th Queen finds a square

So we get to our first correct solution this way:



Invoking **advance** on the first queen will find the next and subsequent solutions.

Methods of the Class Queen

From considering the dynamics of the Queen objects interacting, we see that the following methods are required:

- **can_attack (row, col)**: Ask a queen if she, of any of her neighours can attack the square (r, c).
- **find_solution**: If this queen and her neighbours are not in a solution state, find the next solution. Return true iff successful.
- **advance**: Find the next solution for this queen and neighbours. Return true iff successful.

Constructor for the Queen Class

The constructor for the `Queen` class is straightforward:

```
# initialise the column and neighbour values
def initialize(column, neighbour)
  @row = 1 # varies
  @column = column # fixed
  @neighbour = neighbour
end
```

Note that the `@col` attribute will never change. Ruby provides no annotation for this.

The can_attack Method

The `can_attack` method is conceptually simple, though a bit tricky to implement.

It returns true if this queen or any of her neighbours can attack a given position, false if not

```
def can_attack?(row, column)
  return true if row == @row

  cd = (column - @column).abs
  rd = (row - @row).abs
  return true if cd == rd

  @neighbour.can_attack?(row, column)
end
```

The find_solution Method

The `find_solution` method is as follows.

If this queen and neighbours are not in a solution state, it finds next solution, otherwise it does nothing. Returns true if a solution is found, false otherwise.

```
def find_solution?
  while @neighbour.can_attack?(@row, @column)
    if !advance?
      return false
    end
  end
  return true
end
```

The advance Method

The **advance** method finds the next legal solution for this queen and neighbours:

```
def advance?
  if @row == SIZE
    if !@neighbour.advance?
      return false
    else
      @row = 0
    end
  end
  @row += 1
  return find_solution?
end
```

The NullQueen class

This class is used solely as a **sentinel**, to indicate the end of the set of queens

```
class NullQueen
  def can_attack?(row, column)
    false
  end
  def find_solution?
    true
  end
  def advance?
    false
  end
end
```

This is an example of a **Null Object**.

Script to solve N-Queens

```
neighbour = NullQueen.new
last_queen = nil
num_solutions = 0;

1.upto(SIZE) do |column|
  last_queen = Queen.new(column, neighbour)
  last_queen.find_solution?
  neighbour = last_queen
end

if last_queen.find_solution?
  num_solutions += 1
  print "Solution number #{num_solutions}: "
  last_queen.get_state.each { |state|
    print "(#{state[1]},#{state[0]}) "
  }
  puts ""
end
```

Comments on the N-Queens Puzzle

The `find_solution` and `advance` methods are **mutually recursive**, which can be tricky to comprehend. Just believe that each one fulfills its specification.

Summary

We developed an object-oriented solution to the classic n-queens puzzle.

The metaphor in our solution was the notion of a number of autonomous Queen objects co-operating to find a solution.

The most common approach to this problem is to use backtracking. Our solution involved backtracking as well, but this was hidden behind the interaction of the queens.

… and finally, a solution in C…

```
#include <stdio.h>
int  v,i,j,k,l,s,a[99];
main(){
       for(scanf("%d",&s);*a-s;v=a[j*=v]
       -a[i],k=i<s,j+=(v=j<s&&(!k&&!!
       printf(2+"\n\n%c"-(!l<<!j)," #Q"
       [l^v?(l^j)&1:2])&&++l||a[i]
       <s&&v&&v-i+j&&v+i-j))&&!(l%=s),
       v||(i==j?a[i+=k]=0:++a[i])>=s*k&&
       ++a[--i]);
}
```

Given n, this program prints out all solutions to the n-queens puzzle. This program won the 1990 Obfuscated C competition (http://www.ioccc.org).

Don't program like this!