Intro to Computer Science II

Recursions (2)

Recursive binary search

```
def  bin_search(nums, target, left, right):

    if left > right:   # not found
        return False

    mid = (left + right) // 2
    if nums[mid] == target:
        return True
    elif nums[mid] < target:  # search for upper half
        left = mid + 1
        return bin_search(nums, target, left, right)
    else:
        right = mid -1        # search for lower half
        return bin_search(nums, target, left, right)

nums = [2, 5, 6, 7, 9, 10, 12]
print(bin_search(nums, 2, 0, len(nums)-1))     # True
print(bin_search(nums, 12, 0, len(nums)-1))    # True
print(bin_search(nums, 6, 0, len(nums)-1))     # True
print(bin_search(nums, 22, 0, len(nums)-1))    # False
print(bin_search(nums, 0, 0, len(nums)-1))     # False
```

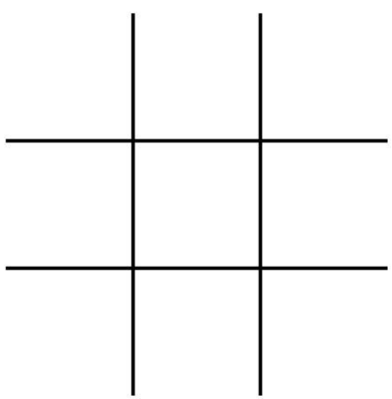
Check if a number is a prime

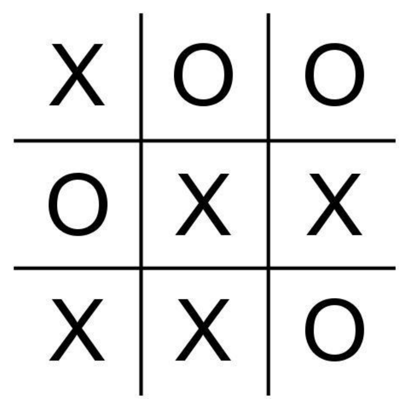
```
def is_prime(b, x):
    if x == 1:
        return True
    elif b % x == 0:
        return False
    else:
        return is_prime(b, x - 1)

print('is_prime(5, 4) ', is_prime(5,4))
print('is_prime(13, 12) ', is_prime(13, 12))
print('is_prime(20, 19) ', is_prime(20,19))
print('is_prime(33, 32) ', is_prime(33, 32))
```

Playing Tic-Tac-Toe

- Consider the game of tic-tac-toe
- If you play tic-tac-toe against a computer, how does the computer make its decisions?





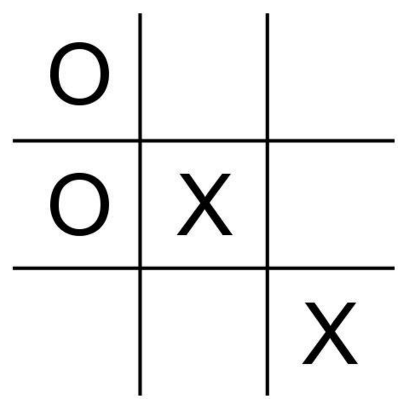
Game Tree

- Provides the sequence of all possible moves that can be made in the game for both opponents.
  - The computer can evaluate the game tree and determine its best move.
  - For tic-tac-toe, the best move is one that
    - allows it to win before its opponent
    - in the fewest possible moves.
  - The “computer” can evaluate every possible move much faster than a human.

Game Tree Example

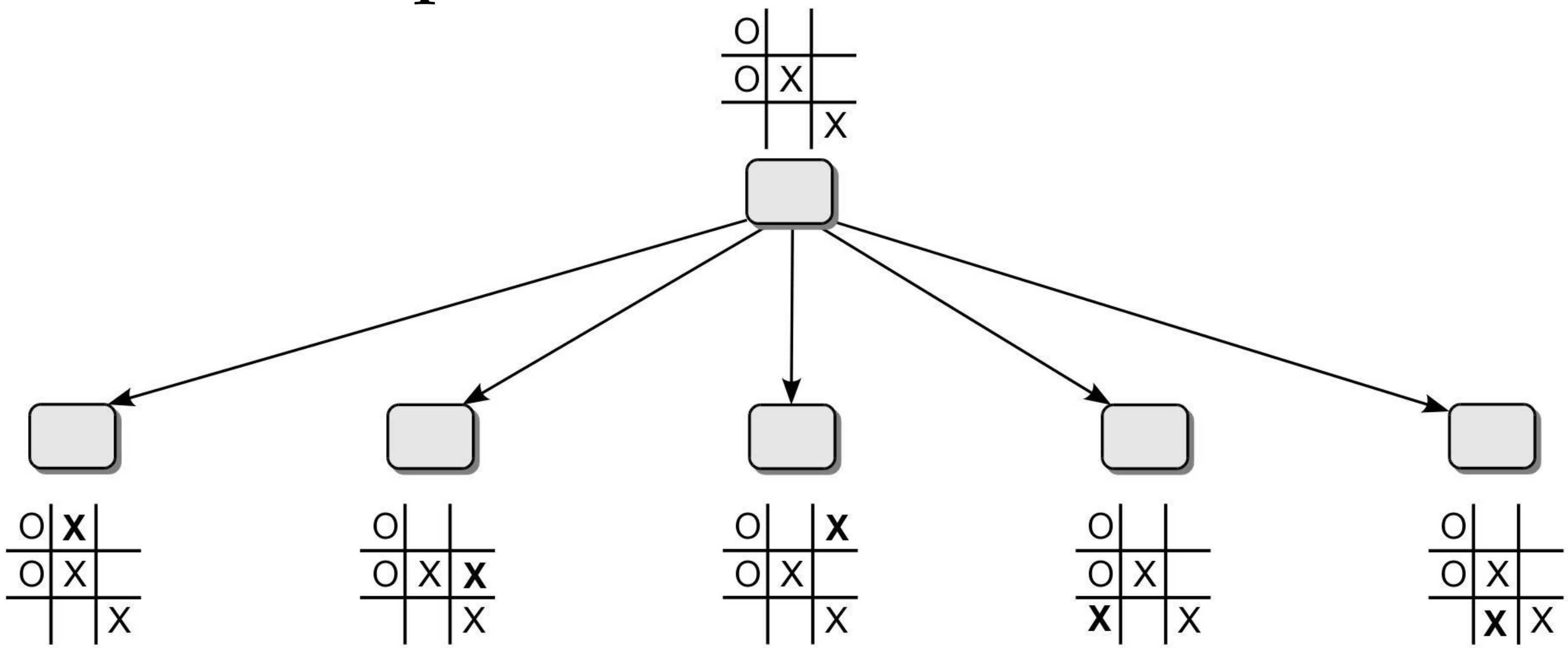
- Suppose you (O) have been playing against the computer (X)

and now it's the computer's turn.



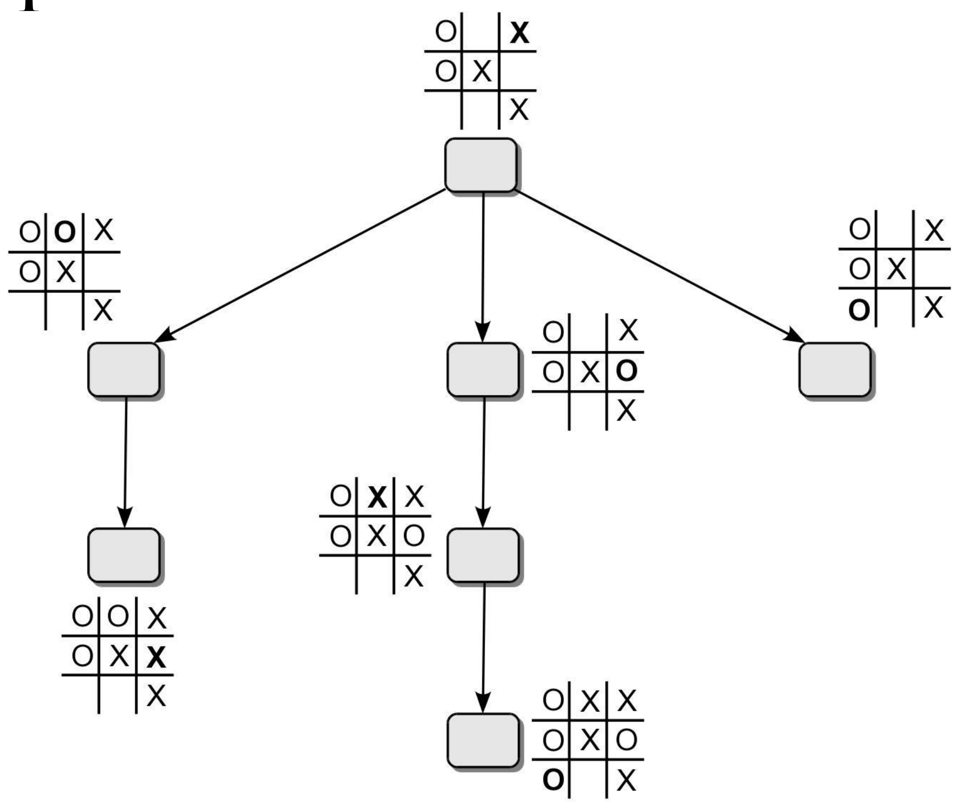
Game Tree Example

- The computer would need to evaluate all of its possible moves to determine



Game Tree Example

- The following figure shows the rest of the tree for a movement in the upper-right square.

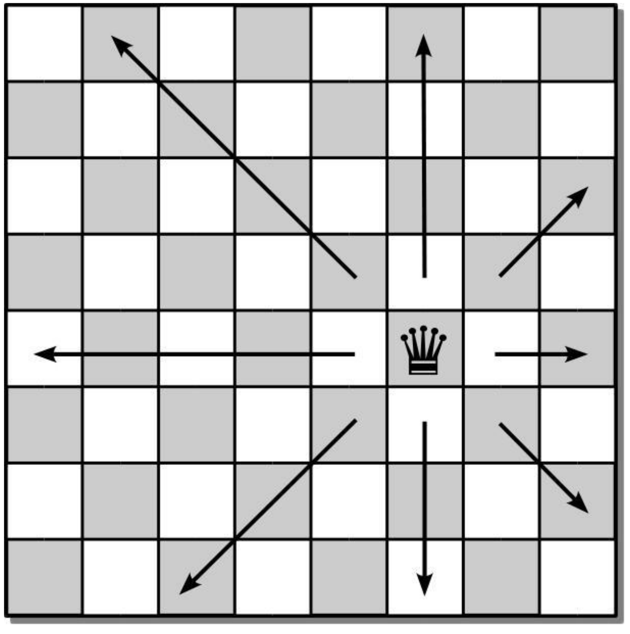


The 8-Queens Problem

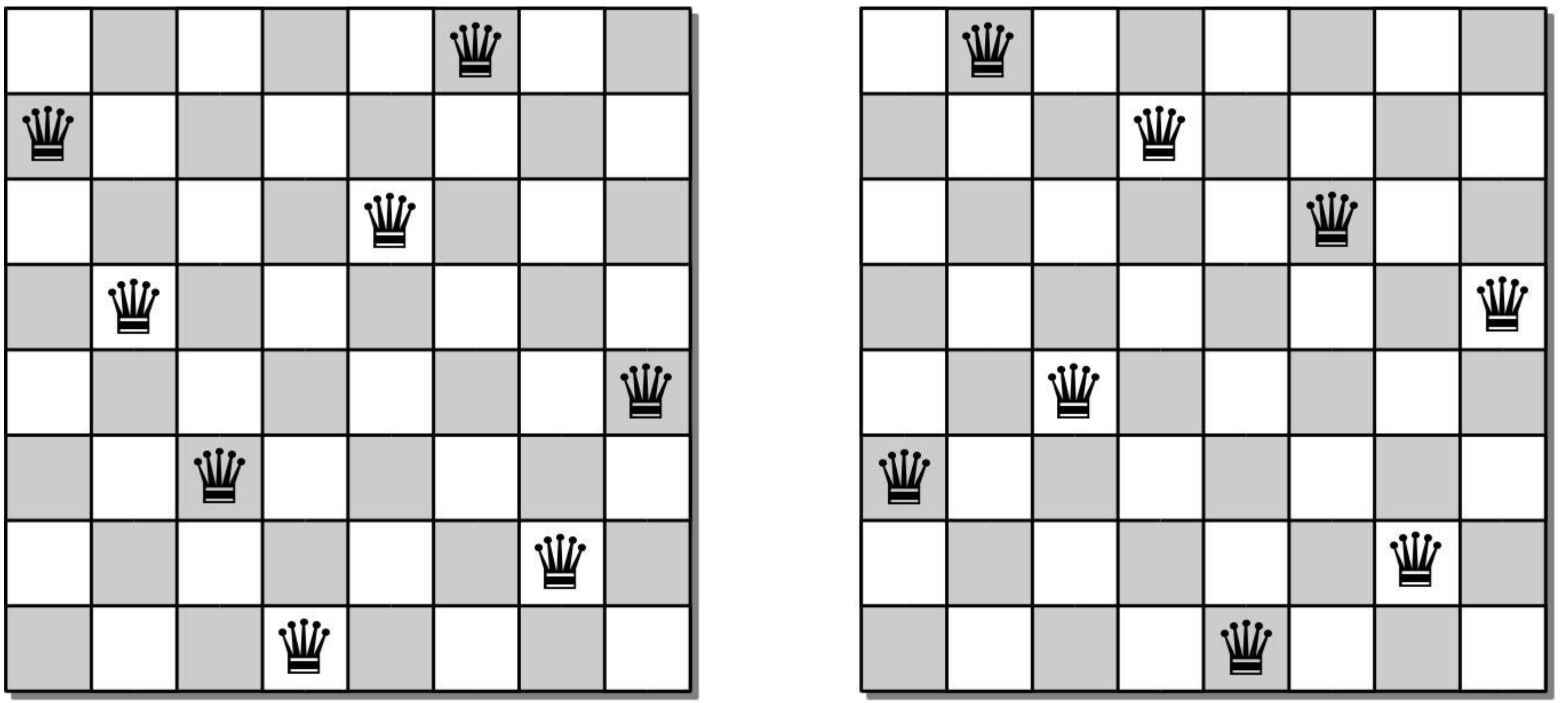
- The task is to place 8 queens onto a chessboard such that no queen can attack another queen.
  - Uses a standard 8 x 8 chess board.
  - There are 92 solutions to this problem.

Queen Moves

- The queen can move and attack any piece of the opponent by moving in any direction along a straight line.

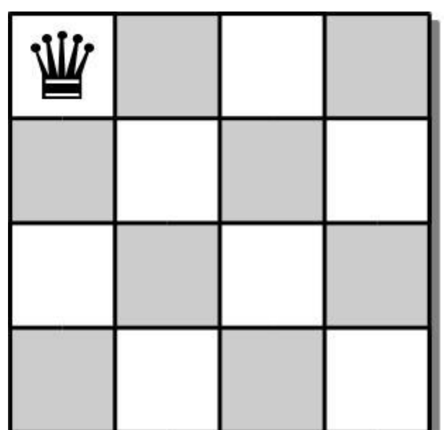


Sample Solutions



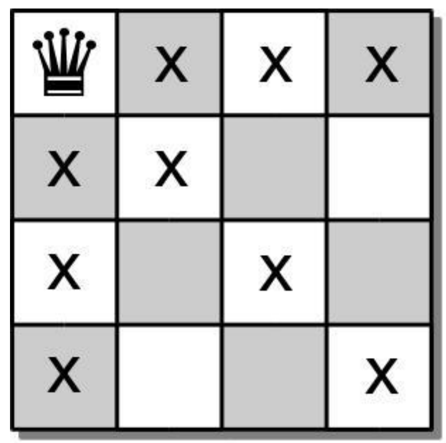
4-Queens Problem

- To develop an algorithm, we consider the smaller 4-queens problem.
  - Since no two queens can occupy the same column, we can proceed one column at a time.
  - Place a queen in position (0, 0).



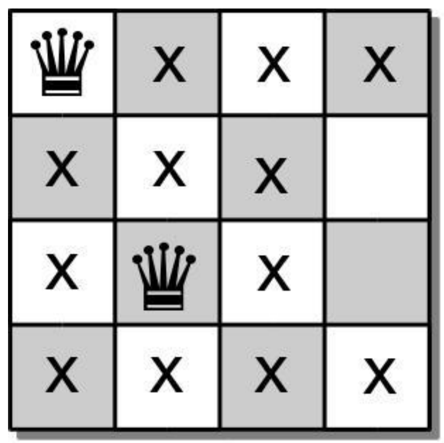
4-Queens Problem

- This move eliminates a number of squares for the placement of additional queens.



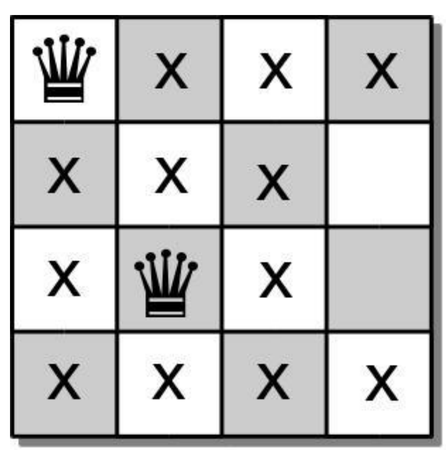
4-Queens Problem

- We move to the second column and place a queen at position (2, 1)



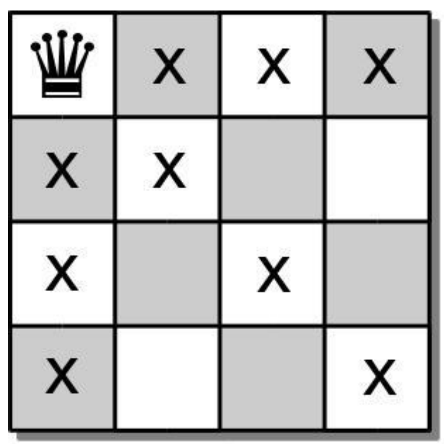
4-Queens Problem

- The 3rd queen should be placed in the 3rd column.
  - But there are no open cells in the third column.
  - So there is no solution based on the placement of the first 2 queens.



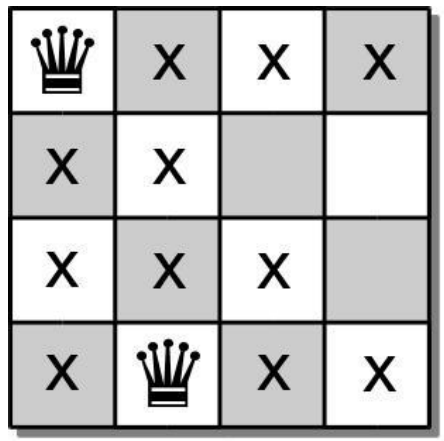
4-Queens Problem

- We have to backtrack:
  - go back to the previous column
  - pickup the last queen placed
  - try to find another valid cell in that column.



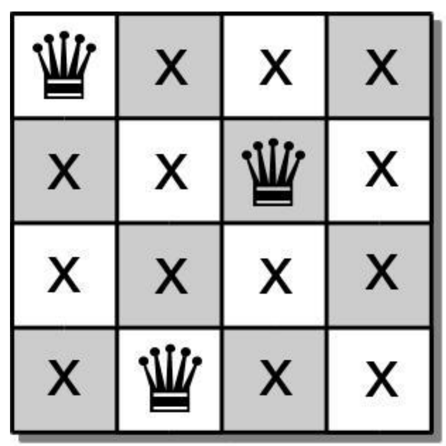
4-Queens Problem

- Place a queen at position (3,1) and move forward.



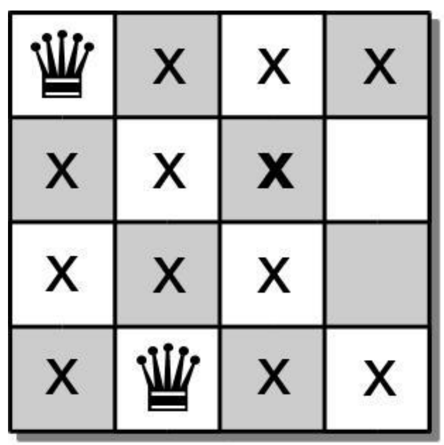
4-Queens Problem

- In the 3rd column, we can now place a queen at position (1,2).
- But now we have no open slots in the 4th column.



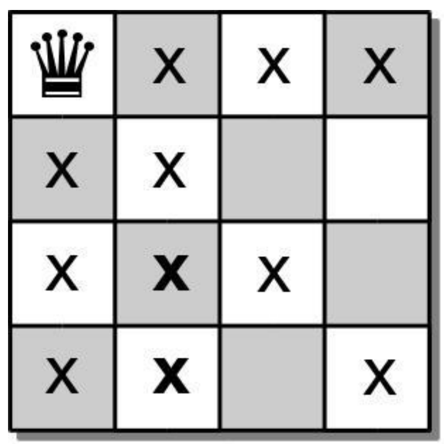
4-Queens Problem

- We again must backtrack and pick up the queen from the 3rd column.
- But there are no other empty cells in the 3rd column.



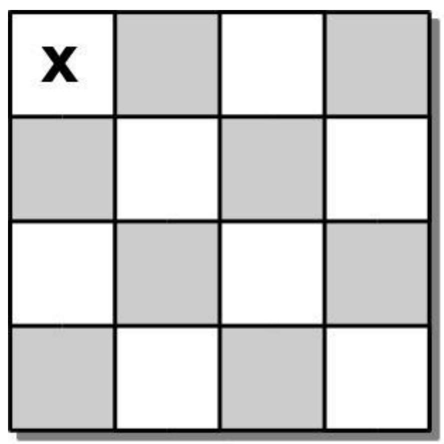
4-Queens Problem

- We must backtrack yet again and pick up the queen from the 2rd column.
- But there are no other empty cells in the 2nd column either.



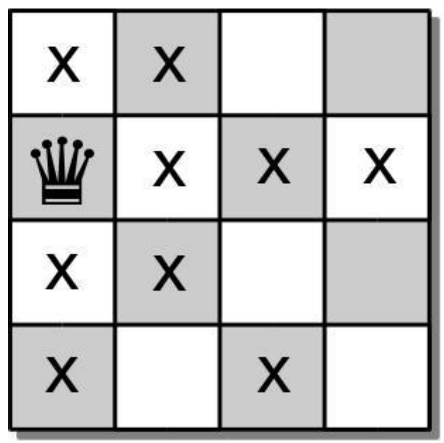
4-Queens Problem

- So we backtrack one more time and pick up the queen from the 1st column.
- We then try again to place a queen in the 1st column.



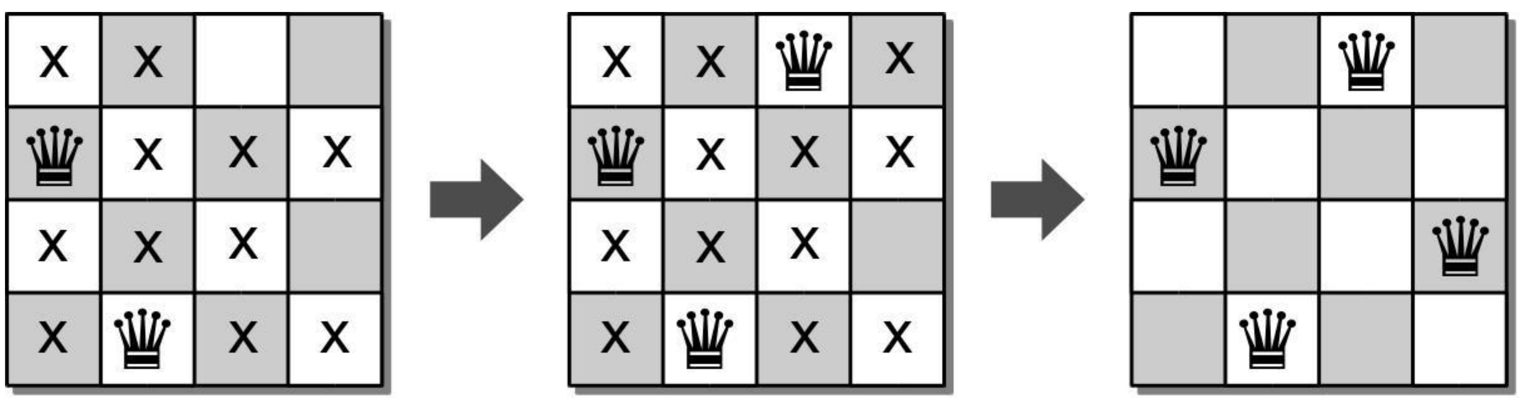
4-Queens Problem

- In the 1st column, we can place a queen at position (1, 0).



4-Queens Problem

- We again continue with the process and attempt to find open positions in each of the remaining columns.
- We can use a similar approach to solve the original 8-queens problem.



N-Queens Board ADT

- The *n-queens board* is used for positioning queens on a square board for use in solving the n-queens problem.
  - consists of *n x n* squares.
  - each square is identified by index *[0...n)*

• NQueensBoard(n) • size() • numQueens() • unguarded(row, col)	• placeQueen(row, col) • removeQueen(row, col) • reset() • draw()

8-Queens Solution

```
def solveNQueens( board, col ):
  if board.numQueens() == board.size() :
    return True
  else :
    for row in range( board.size() ):
      if board.unguarded( row, col ):
        board.placeQueen( row, col )
        if board.solveNQueens( board, col+1 ) :
          return True
        else :
          board.removeQueen( row, col )

    return False
```