Solution Review: Place N Queens on an NxN Chessboard

In this lesson, we will go over the solution to the challenge from the previous lesson.

We'll cover the following

Solution

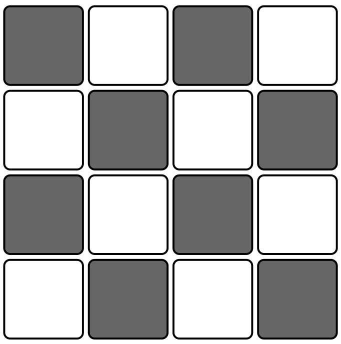
```
def isSafe(i, j, board):
  for c in range(len(board)):
    for r in range(len(board)):
      # check if i,j share row with any queen
      if board[c][r] == 'q' and i==c and j!=r:
        return False
      # check if i,j share column with any queen
      elif board[c][r] == 'q' and j==r and i!=c:
        return False
      # check if i,j share diagonal with any queen
      elif (i+j == c+r or i-j == c-r) and board[c][r] == 'q':
        return False
  return True

def nQueens(r, n, board):
  # base case, when queens have been placed in all rows return
  if r == n:
    return True, board
  # else in r-th row, check for every box whether it is suitable to place queen
  for i in range(n):
    if isSafe(r, i, board):
      # if i-th columns is safe to place queen, place the queen there and check recursively for ot
      board[r][i] = 'q'
      okay, newboard = nQueens(r+1, n, board)
      # if all next queens were placed correctly, recursive call should return true, and we should
      if okay:
        return True, newboard
      # else this is not a suitable box to place queen, and we should check for next box
      board[r][i] = '-'
  return False, board

def placeNQueens(n, board):
```

```
  return nQueens(0, n, board)[1]

def main():

  n = 4
  board = [["-" for _ in range(n)] for _ in range(n)]
  qBoard = placeNQueens(n, board)
  qBoard =  "\n".join(["".join(x) for x in qBoard])
  print (qBoard)
main()
```

Explanation

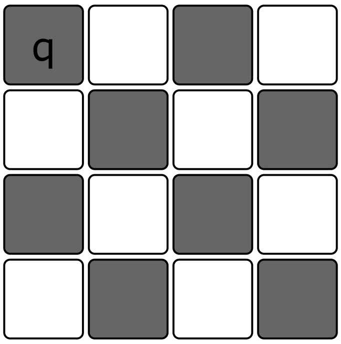
Let's break down what we did there. The basic idea is to place the queen at all possible positions to find out what fits our needs. We start off placing a queen in the first row's first box and then make a recursive call to place a queen in the second row. Here we place a queen in a safe position and check recursively again for the next rows. If any of the recursive calls return false, we check the next box on the previous row, and so on.

Look at the visualization of a dry run on an example where `n = 4`.



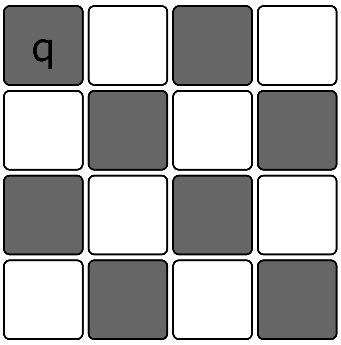
placeNQueens(4)

1 of 33



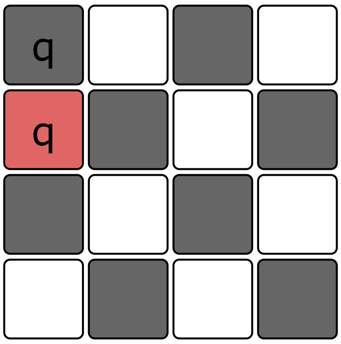
place a queen in first row's first box and make a recursive call

2 of 33



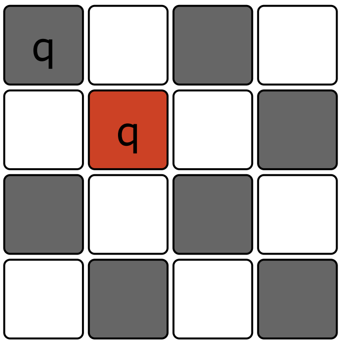
try to find a suitable box to place queen in second row

3 of 33



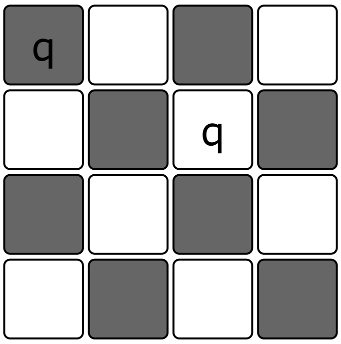
box unsafe; shares column with a queen

4 of 33

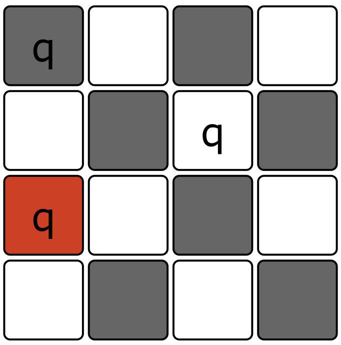


box unsafe; shares diagonal with a queen

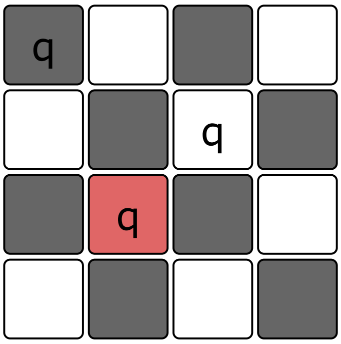
5 of 33



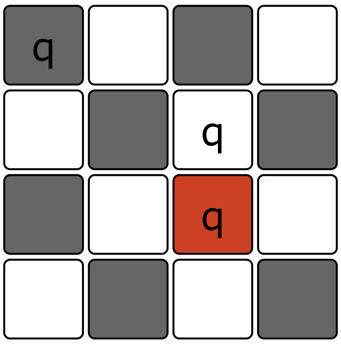
queen placed, make recursive call



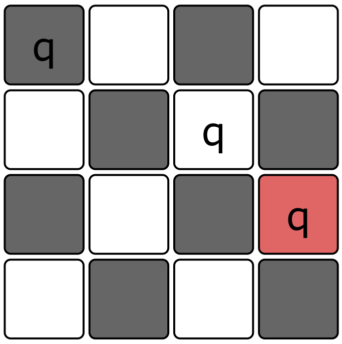
box unsafe; shares column with a queen



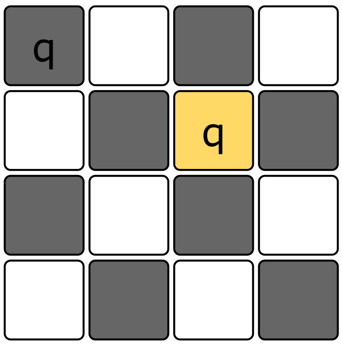
box unsafe; shares diagonal with a queen



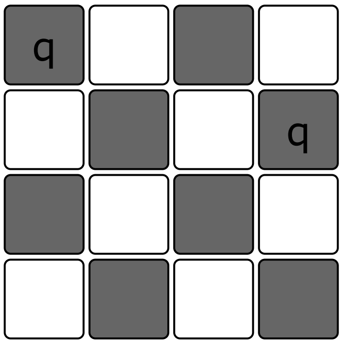
box unsafe; shares column with a queen



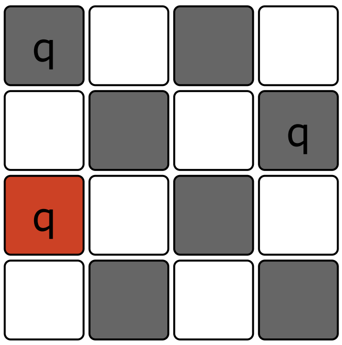
box unsafe; shares diagonal with a queen



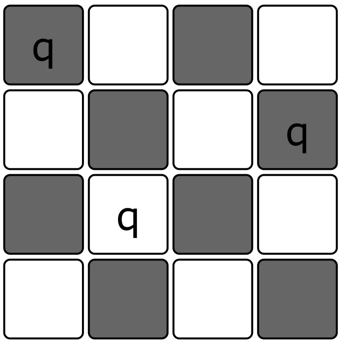
cannot place queen in rows below; check next box



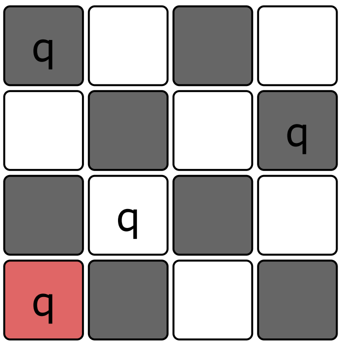
queen placed, make recursive call



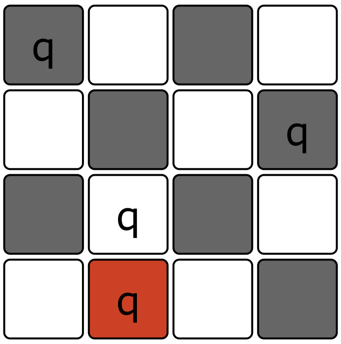
box unsafe; shares column with a queen



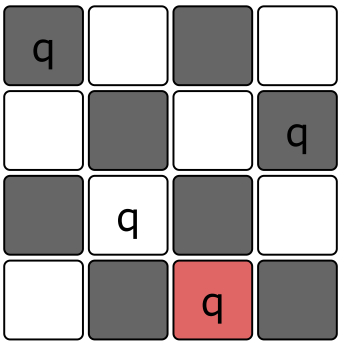
queen placed, make recursive call



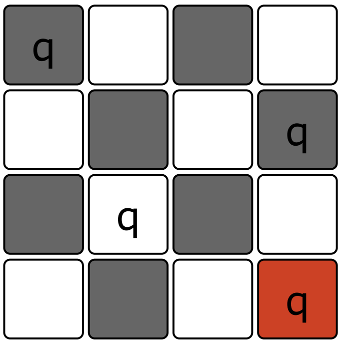
box unsafe; shares column with a queen



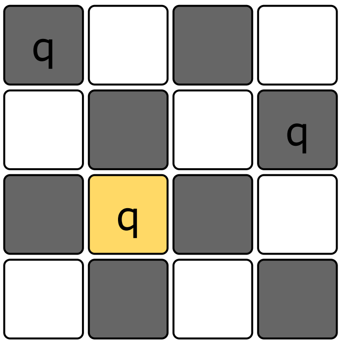
box unsafe; shares column with a queen



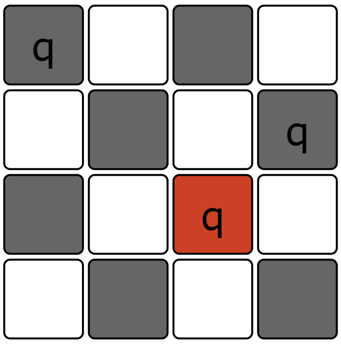
box unsafe; shares diagonal with a queen



box unsafe; shares column with a queen

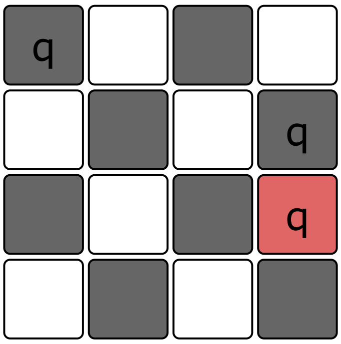


cannot place queen in rows below; check next box



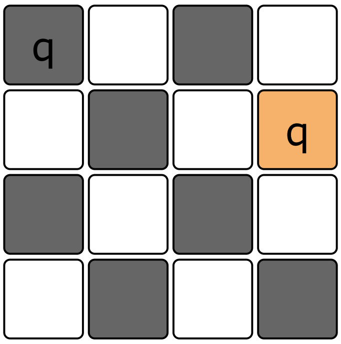
box unsafe; shares diagonal with a queen

20 of 33

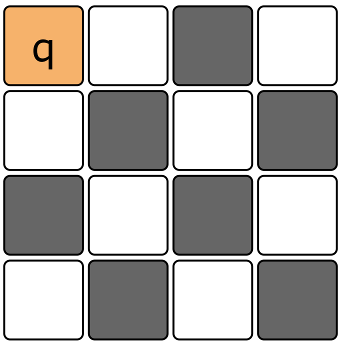


box unsafe; shares column with a queen

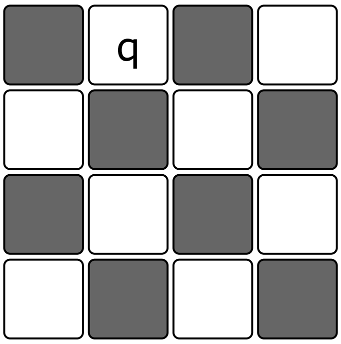
21 of 33



cannot place queen in rows below;

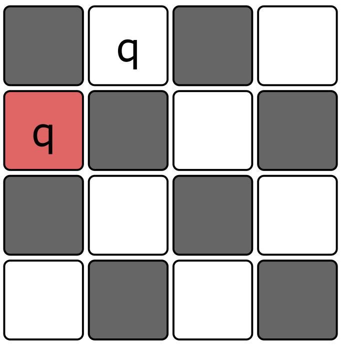


cannot place queen in rows below; check next box



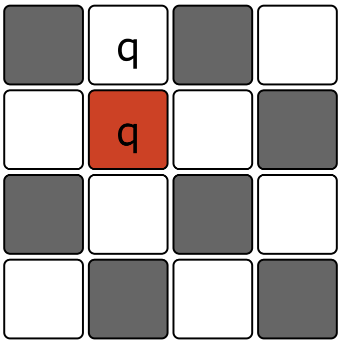
queen placed, make recursive call

24 of 33

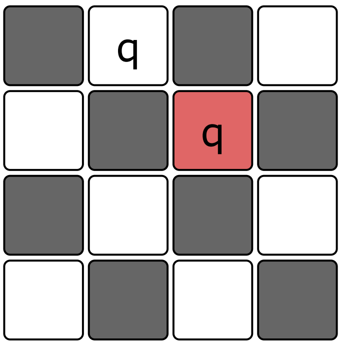


box unsafe; shares diagonal with a queen

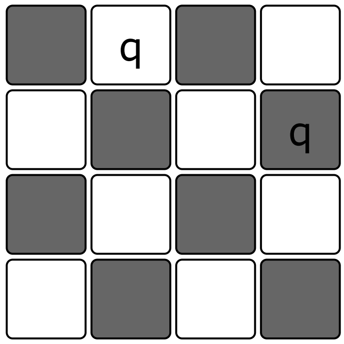
25 of 33



box unsafe; shares column with a queen

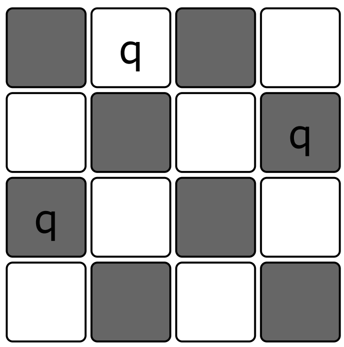


box unsafe; shares diagonal with a queen



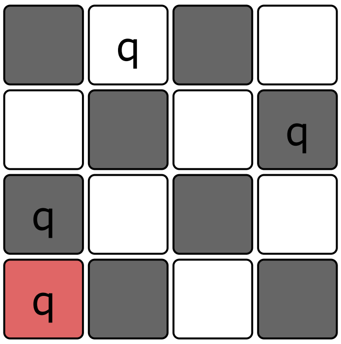
queen placed, make recursive call

28 of 33

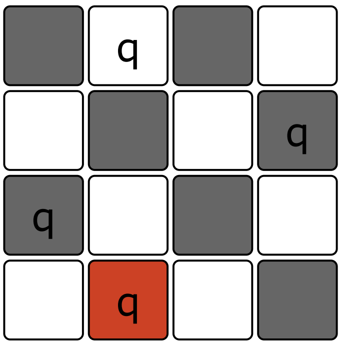


queen placed, make recursive call

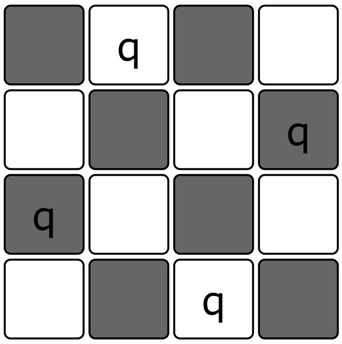
29 of 33



box unsafe; shares column with a queen

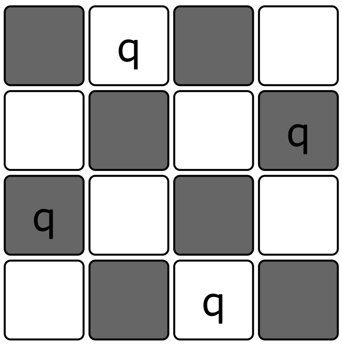


box unsafe; shares diagonal with a queen



queen placed, make recursive call

32 of 33



n queens have been placed, return True, board

33 of 33

Observe how simple and intuitive this solution is. All we are doing is placing queens in a row and checking whether, after placing that queen, we can place queens in proceeding rows. Now if you look at the code, start with *line 20*, where we iterate over the `r`th row and see which box is safe to place a queen (*line 21*). When we find such a box, we update its value (*line 23*) and make a recursive call to check for proceeding rows (*line 24*). If this recursive call returns `True`, this means all `n` queens have been placed and we should return `True` as well (*lines 26-27*). Otherwise, we need to revert the queen we placed (*line 29*) and continue checking for the rest of the boxes in `r`th row. If no box is suitable, we return `False`.

Now, let's come to our base case. Since we are going row by row, when we have placed queens safely in `n` rows, we do not need to do anything further. This is our base case, given in *lines 17-18*.

Time complexity

This algorithm has a pretty high time complexity: **O(n^n)**. Think about the solution space. In the first row you can place a queen at `n` places. For each of these `n` options, you have the option to place a queen at `n` more in the second row, and so on up to `n`. Therefore:

$$n \times n \times n \ldots \times n = n^n$$

Plug in a slightly bigger number (~20) in the code playground above and see how the response time changes.

This was it for simple recursion, but don't worry, because we will be working on even more recursion problems in later chapters. In the next lesson, we will look at why recursion alone is not enough.