Eight Queens

210CT 2015/16

Week A, Block 2, Stardate 453.221, Last Seed

1 Introduction

The Eight Queens problem is a chess puzzle, first devised by the chess composer Max Bezzel in 1848.

Can you place 8 queens on an 8×8 chess board so that no queen threatens another? How many ways can this be done?

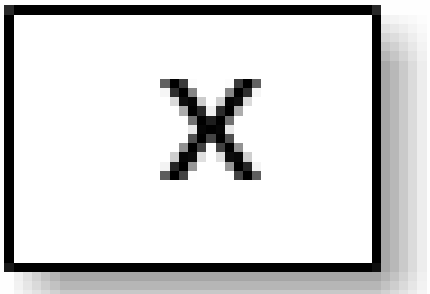
For today, we're going to concentrate solely on finding a single solution.

Why? A little practice thinking about algorithms before we move on to more complex issues

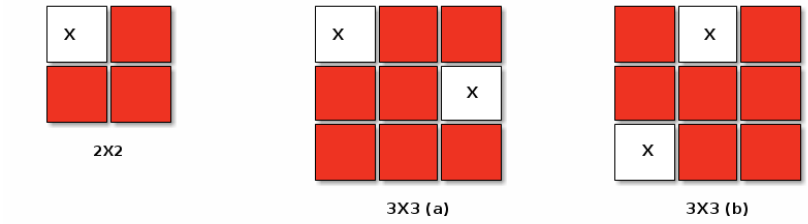
2 Thinking About the Algorithm

Sometimes it helps to reduce the problem to its smallest example.

What if we had a board with one cell, that needs one queen?

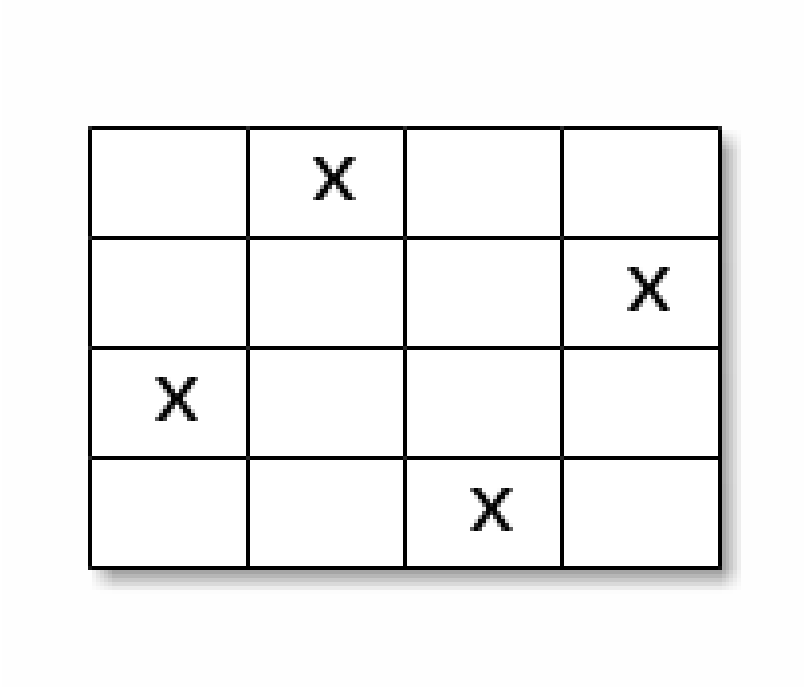


Interestingly, a 2×2 board with 2 queens can't work, and the same problem applies to 3×3 with 3 queens. After that, we're fine.



Those are all the options you have for the 2- and 3-Queens problems. Any other starting point or next step is just a rotation and/or reflection of those.

Here's 4×4 with 4 queens.



3 Generalisation

So we have generalised the problem to n queens on an $n \times n$ board.

This is good for two reasons:

3.1 Reason 1

- Often problems belong to a "class".
- Any given instance of a problem probably has very specific constraints.
- In our case, it was the size of the board and the rules for the piece given.
- You might be able to solve the given problem within those constraints without generalising, but you will have done it by writing code that you might never use again.
- If you generalised, you might have something that can be used to solve other problems. The eight queens problem is probably not the most useful, but imagine this with sorting algorithms, route finding, etc.

3.2 Reason 2

- A problem can be made easier to fathom if you work out which parts can be altered in such a way that the principle for solving the problem stays the same, but the complexity of the current case is lower.
- The "four queens" problem is a good example of that. It was easy to solve that one and we can now use this to help us work out the general strategy.

4 Eight Queens Algorithm 1

- Try all combinations of 8 pieces
- Work out how many need to be tested for an 8×8 or $n \times n$ board.
- Think of the spaces as being choices. We have $8 \times 8 = 64$ of them. We can pick 4. This is a non-repeating combination, in which order doesn't matter, so...
- $\frac{64!}{4!(64-4)!} = \frac{1268869321858841641034333893351614808028655161745451921988018943752147042304000000000000000}{24\times832098711274139014427634118322336438075417260636124595244927769640960000000000000000}$
- $= 635,376$

5 Eight Queens algorithm 2

First, we begin on the top row of the board. Since the rules mean no other queen can be placed on this row afterwards and also means every row must have a queen on it (**think about why we can say this just from the rules we have!**), we can solve the problem one row at a time.

1. For the row we are interested in, find a space that a queen can stand without being seen by any queens on the board. This is the same as saying "find a place where a queen can stand such that it cannot see another already-placed queen". **Why is this a useful realisation?**
2. Move on to the next row.
3. Repeat.

Simple, yes?

6 Defining our data

Along with every algorithm is the data that it operates on. It's important to consider this data along with the algorithm. Often they're tied very closely together.

For this problem, I'm going to use a "list of lists" in python. In other languages, this might be called a 2D array.

Here's an example board being made and queried:

```
board=[]  #This is an empty list.

for i in range(4):

  #Make a  row
  row=[False] * 4  #results in [False, False, False. False]

  #Add it to the board
  board.append(row)
board[3][2] = True #row 3, column 2
print board
```

That's a little ugly, so let's think of a way to display boards in a general purpose way...

```
board=[]  #This is an empty list.

for i in range(4):

    #Make a  row
    row=[False] * 4  #results in [False, False, False. False]

    #Add it to the board
    board.append(row)

board[3][2] = True #row 3, column 2
def displayBoard(b):
    boardsize=len(b) #How many rows?

    #A string multiplied by an int repeats the string...
    divider=("+——"*boardsize)+"+"

    for row in b:
        print divider

        pieces=[] #Turn booleans into string representations
        for i in row:
            if i:
                pieces.append("X")
            else:
                pieces.append("␣")

        #join makes a single string from a list of strings by
        #inserting a string between each item
        print "|␣"+"␣|␣".join(pieces)+"␣|"

    print divider

displayBoard(board)
```

7 Some useful functions

Here are some more functions to reduce the complexity of our answer. Each one takes away some of the "fiddly bits" and leaves us to worry about the N-queens algorithm.

```
def makeBoard(size):
    board=[]
    for i in range(size):
        board.append([])
        for j in range(size):
            board[-1].append(False)
    return board

def displayBoard(b):
    divider=("+---"*len(b))+"+"
    for row in b:
        print divider
        print "|"+"|".join({True:"␣X␣",False:"␣␣␣"}[i] for i in row)+"|"
    print divider


def setAppend(s,i):
    """␣Add␣i␣to␣s␣unless␣i␣is␣already␣in␣s␣"""
    if not i in s: s.append(i)

def inBoard(p,size):
    """␣if␣point␣is␣valid␣for␣a␣board␣of␣given␣size,␣return␣True.␣Else␣return␣False␣"""
    if 0<=p[0]<size and 0<=p[1]<size:
        return True
    else:
        return False

def pointShift(p,r,c):
    """␣Return␣position␣of␣cell␣r,c␣away␣from␣given␣point␣p␣"""
    return (p[0]+r,p[1]+c)

def appendIfInBounds(s,p,size):
    """␣If␣point␣p␣is␣within␣the␣bounds␣of␣a␣board␣of␣given␣size,␣append␣to␣s␣unless␣it's␣al
    if inBoard(p,size):
        setAppend(s,p)

def queenSees(pos,size):
    """␣Return␣a␣list␣of␣all␣squares"In␣view"␣of␣a␣queen␣in␣position␣pos␣on␣a␣board␣of␣size"
    inView=[]
    #Row and column
    for i in range(size):
        #Column
        setAppend(inView,(i,pos[1]))
        #Row
        setAppend(inView,(pos[0],i))
        #Diagonals
        for r in [-1,1]:
            for c in [-1,1]:
                appendIfInBounds(inView, pointShift(pos,r*i,c*i), size)
    #Take out position of queen so she doesn't see herself...
    inView.remove(pos)
    return inView

def hasQueen(board, points):
    """␣Returns␣True␣if␣any␣of␣the␣given␣points␣on␣the␣given␣board␣contain␣a␣queen␣"""
    for p in points:
        if board[p[0]][p[1]]:
```

```
                return True
        return False

    def cloneBoard(b,size):
        """ Make a copy of a board.  Boards are objects (lists are objects) so a=b just makes the
        c=makeBoard(size) #clone
        for i in range(size):
            for j in range(size):
                c[i][j]=b[i][j]
        return c
```

8 Implementing the algorithm

```
    def fillBoardNaive(size):
        b=makeBoard(size)                                   #blank board of given size
        for r in range(size):                               #For each row...
            for c in range(size):                           #   Check each cell until...
                if not hasQueen(b,queenSees((r,c),size)):   #      We find one suitable
                    b[r][c]=True                            #      Put a queen there
                    break                                   #      Skip to the next row
            else:                                           #   If the loop didn't break...
                break                                       #      Break the outer loop
                                                            #         because something went wron
        return b
```

And now let's use it...

```
b=fillBoardNaive(4) #Use our simple case first...
displayBoard(b)
```

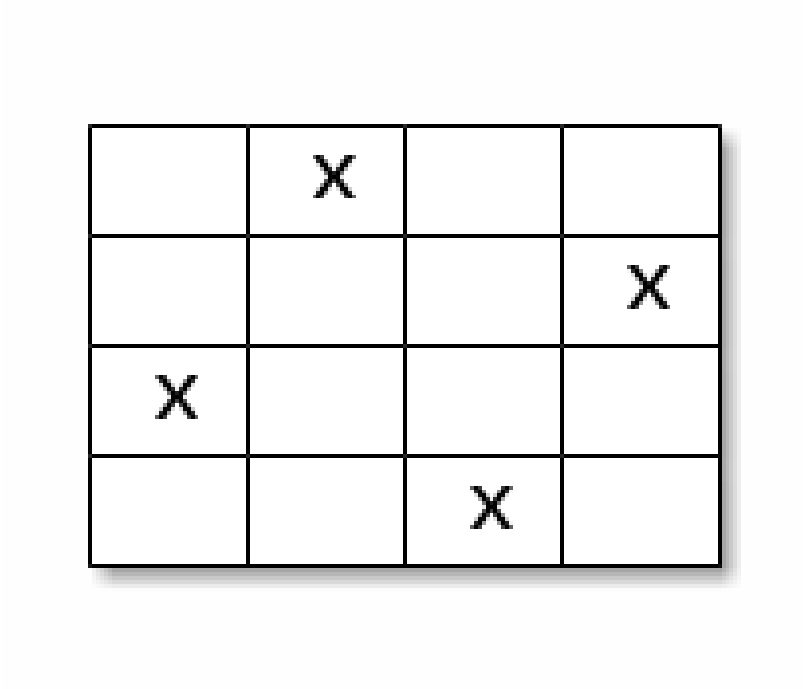
```
+---+---+---+---+
| X |   |   |   |
+---+---+---+---+
|   |   | X |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
```

Great, so now... oh, wait... it didn't work.

It got two lines in and then could go no further because there was no place to put a queen. But we know it's possible to complete a 4×4 grid!

9 Why does it fail?

Here was our answer from earlier:



We know it can be done, but our algorithm wasn't up to the job. Let's investigate.

Follow the algorithm:

1. We picked the top left corner first, because it was first in the row.
2. Clearly we can place a queen there because there are none on the board
3. Next we pick the first available slot on the next row that can have a queen
4. Row 3 has no spaces suitable.

Try making a different choice at row 2...

... nope. In fact, it turns out that the first placement was the one that prevented us from completing.

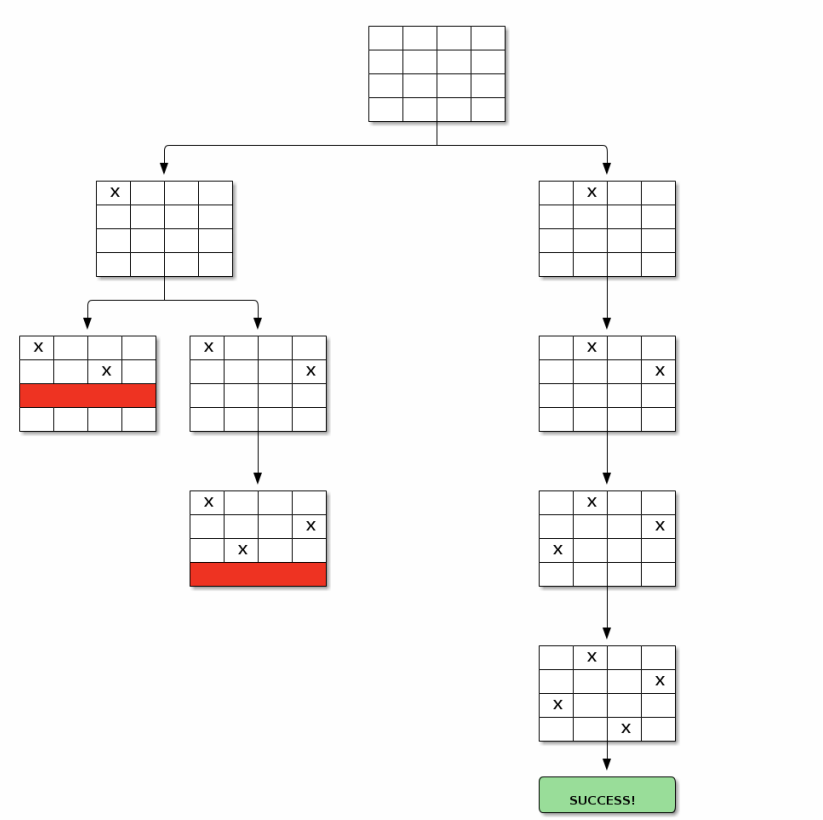
So, our algorithm is a bit naive. We can't just take the first available space, because it might cause us to end up in a position from which we can't complete.

We need a way to go back and undo decisions when we find that there has been a problem.

10 Eight Queens Algorithm 3: Recursion

What we would do as humans, is to try our solution and if it doesn't work, go back a step and try the alternatives. If they don't work, go back a step further and try the alternatives there.

Let's think about how that might work...



- So our **base case** is: If all rows have been done, finish.
- And our **recursive case** is:
  - if you have a cell that a queen can be placed in that has not already been checked, place a queen and then examine the next row
  - if all cells that could have a queen have been checked, go back a row to try a different cell. This is called **back tracking**

10.1 Recursive implementation

```
def fillBoardRecursion(board,row, size):
    """ Given a board completed to given row, try all possible positions for next row and co
    if row==size:
        #base case
        return board
    else:
        for col in range(size):
            if not hasQueen(board, queenSees((row,col),size)):
```

```
            b=cloneBoard(board,size)
            b[row][col]=True
            result= fillBoardRecursion(b,row+1,size)
            if result!=False:
                return result
    return False
```

11 Other Pieces

What would we have to do to work with other pieces?

- Not much, possibly
- If we can still work row by row, all we need to do is look at the function that checks the validity of a piece placement.
- The "queenSees" function in our implementation is the one that determines the places we mustn't see another piece, so we could just swap this out.
- Think about how this would work with:
  - Rooks. (Easy)
  - Bishops. (Easy, odd, but valid, result...)
  - King. (Very easy)
  - Pawn. (Easyish. Unlike before, you need to determine direction of the board)
  - Knight. (Errr... row-by-row makes no sense!)

12 For fun

- If you don't care about readability and like python tricks...

Here's an alternative to the board drawing function with identical output. It uses list comprehension to convert from bools to strings.

```
def displayBoard(b):
    divider=("+---"*len(b))+"+"
    for row in b:
        print divider
        print "|"+"|".join({True:"␣X␣",False:"␣␣␣"}[i] for i in row)+"|"
    print divider
```

13 Homework

13.1 Pre-homework

1. In your language of choice, create a list/aray of 40 integers

2. Randomly fill it with 0s and 1s
3. Print it
4. Now make a new list of the same length and for each element, fill it only if in the previous list, this element plus the one on either side, adds up to 1 or 2.
5. Jump to 3

Example output:

13.2 Actual

- Look carefully at the "queenSees" function. Write two new functions:
  - rookSees
  - knightSees
- The functions should perform the same purpose and return data in the same format, but for the given chess piece.

```
def queenSees(pos,size):
    """ Return a list of all squares"In view" of a queen in position pos on a board of size"
    inView=[]
    #Row and column
    for i in range(size):
        #Column
        setAppend(inView,(i,pos[1]))
        #Row
        setAppend(inView,(pos[0],i))
        #Diagonals
        for r in [-1,1]:
            for c in [-1,1]:
                appendIfInBounds(inView, pointShift(pos,r*i,c*i), size)
    #Take out position of queen so she doesn't see herself...
    inView.remove(pos)
```

13.3 Complete code

```
def makeBoard(size):
    board=[]
    for i in range(size):
        board.append([])
        for j in range(size):
            board[-1].append(False)
    return board

def displayBoard(b):
    divider=("+---"*len(b))+"+"
    for row in b:
        print divider
```

```
        print "|"+"|".join({True:"␣X␣",False:"␣␣␣"}[i] for i in row)+"|"
    print divider


def setAppend(s,i):
    """␣Add␣i␣to␣s␣unless␣i␣is␣already␣in␣s␣"""
    if not i in s: s.append(i)

def inBoard(p,size):
    """␣if␣point␣is␣valid␣for␣a␣board␣of␣given␣size,␣return␣True.␣Else␣return␣False␣"""
    if 0<=p[0]<size and 0<=p[1]<size:
        return True
    else:
        return False

def pointShift(p,r,c):
    """␣Return␣position␣of␣cell␣r,c␣away␣from␣given␣point␣p␣"""
    return (p[0]+r,p[1]+c)

def appendIfInBounds(s,p,size):
    """␣If␣point␣p␣is␣within␣the␣bounds␣of␣a␣board␣of␣given␣size,␣append␣to␣s␣unless␣it's␣al
    if inBoard(p,size):
        setAppend(s,p)

def queenSees(pos,size):
    """␣Return␣a␣list␣of␣all␣squares␣"In␣view"␣of␣a␣queen␣in␣position␣pos␣on␣a␣board␣of␣size
    inView=[]
    #Row and column
    for i in range(size):
        #Column
        setAppend(inView,(i,pos[1]))
        #Row
        setAppend(inView,(pos[0],i))
        #Diagonals
        for r in [-1,1]:
            for c in [-1,1]:
                appendIfInBounds(inView, pointShift(pos,r*i,c*i), size)
    #Take out position of queen so she doesn't see herself...
    inView.remove(pos)
    return inView

def hasQueen(board, points):
    """␣Returns␣True␣if␣any␣of␣the␣given␣points␣on␣the␣given␣board␣contain␣a␣queen␣"""
    for p in points:
        if board[p[0]][p[1]]:
            return True
    return False

def cloneBoard(b,size):
    """␣Make␣a␣copy␣of␣a␣board.␣␣Boards␣are␣objects␣(lists␣are␣objects)␣so␣a=b␣just␣makes␣the
    c=makeBoard(size) #clone
    for i in range(size):
        for j in range(size):
            c[i][j]=b[i][j]
    return c
```

```
def fillBoardRecursion(board,row, size):
    """ Given a board completed to given row, try all possible positions for next row and co
    if row==size:
        #Base case
        return board
    else:
        for col in range(size):
            #If we put a queen here, would it "see" another?
            if not hasQueen(board,queenSees((row,col),size)):
                b=cloneBoard(board,size)
                b[row][col]=True
                result= fillBoardRecursion(b,row+1,size)
                if result!=False:
                    return result
        return False #Failed at this point, so return False
b=makeBoard(8)
b=fillBoardRecursion(b,0,8)
displayBoard(b)
```

Emacs 23.3.1 (Org mode 8.0.3)