Recursive and Stack-based Solutions to the N-Queens Puzzle

Lauren Chambers

December 2020

1 What is the N-Queens puzzle?

The n-Queens puzzle involves placing n Queens (one on each row) on an n x n chessboard so that no two queens threaten each other. Queens pose threat to each other if they are in the same row, column, or diagonal. There are solutions to n-Queens for all natural numbers greater than or equal to 4. Figure 1 depicts an example solution to n-queens for n = 4.

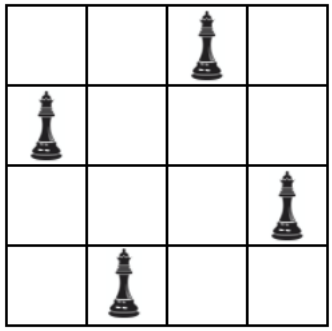

Figure 1: Example of n-Queens Solution: n = 4

2 Systematically Solving N-Queens

Though one could take a trial-and-error approach, there are systematic ways of solving n-Queens. If no Queen is placed on a cell under threat for all n rows of a chessboard, the puzzle has been solved. Let's take a visual walk through this approach with the 4-Queen solution. Row 1 is considered first. There are no cells in row 1 that are under threat because no Queens have been placed yet. We choose to place a Queen in column 3. After the placement of the Queen at

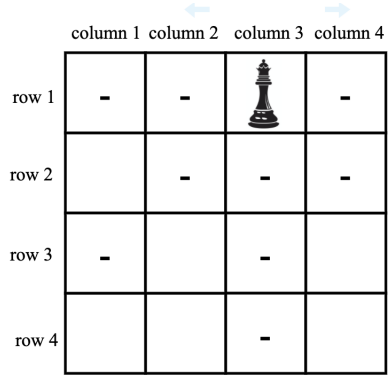

Figure 2: 4-Queens Solution example step 1

cell [1, 3] all cells in the same row, column, and diagonal as the cell are under threat. Figure 2 depicts the updated board with the Queen and a "-" on cells that are under threat.

A Queen has been placed in row 1, so we can move on to the placement of a Queen in row 2. Glancing at row 2 in Figure 2, we see that cell [2, 1] is the only cell that is not under threat. A Queen is placed in cell [2, 1] and all cells in its row, column, and diagonals are marked with another - to indicate threat. Figure 3 depicts the updated board which will be used to determine the placement of the third Queen.

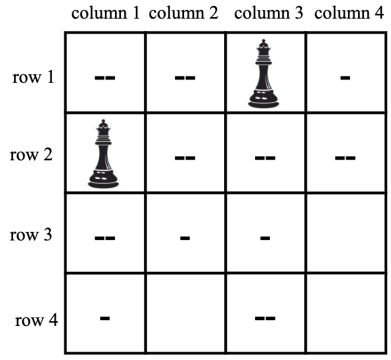

Figure 3: 4-Queens Solution example step 2

Row 3 in Figure 3 is examined and a Queen is placed on cell [3, 4] according to the so-called no-threat rule. See Figure 4 for the updated board. Lastly, row 4 is examined - there is no threat on cell [4, 2] so a Queen is placed there. Figure 5 shows the completed board. Because a Queen has been placed on a "safe" cell on on all 4 rows, Figure 5 depicts a solution to the 4-Queens problem.

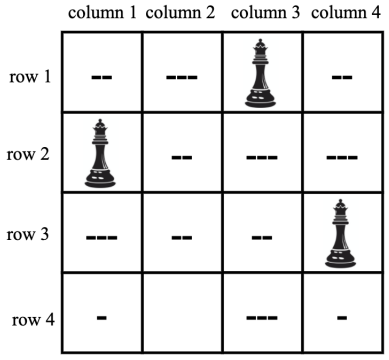

Figure 4: 4-Queens Solution example step 3

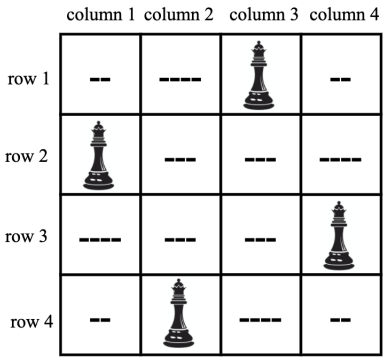

Figure 5: 4-Queens Solution example step 4

Summary: Place a queen in the first row and mark all cells in its same row, column, and diagonals as under threat. Place a new Queen on any of the unthreatened cells in row 2, and mark all cells under its threat (row, column, diagonals). Cells under the threat of multiples Queens should be marked multiple times. Repeat this process until a Queen is placed in row n of the board. If there are no safe cells in a row, remove the Queen that was just placed. Figure 6 depicts all the Queen placements that this algorithm generates for n = 4.

3 Recursive Solution to N-Queens

The process outlined in the previous section lends itself well to recursion. For a Queen placed on each cell of the top row, a recursive call is made to place a Queen on available cells in the next row down. After the recursive call, the Queen and its threat is removed from the board. If all cells in a row are under

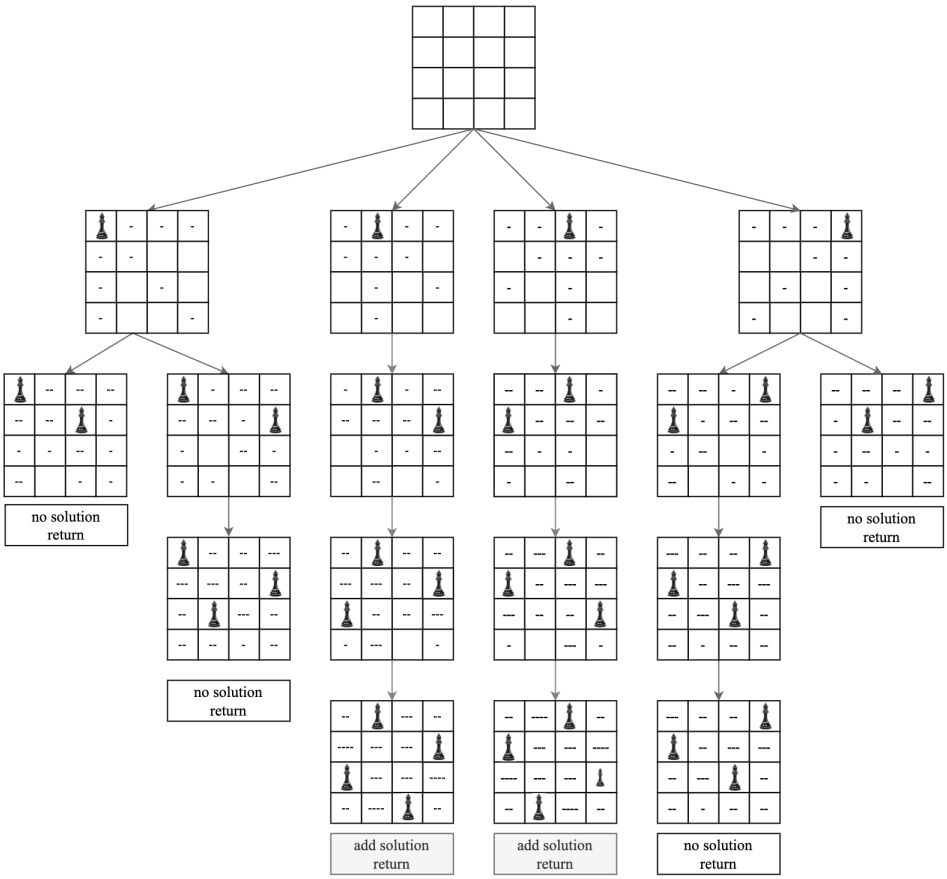

Figure 6: Pathways to 4-Queens Solutions

threat, no placements or recursive calls are made. The argument row number serves as the base case of the recursion - if it exceeds n, n Queens have been placed and a solution is found. See Algorithm 1 for pseudocode.

4 Non-recursive Solution to n-Queens

Recursion can reduce the time complexity of an algorithm. It can be an intuitive way of solving some problems like n-Queens and it is often easier to debug. However, if the input to the problem is large and many recursive calls must be made, overhead can eat up stack memory and add to the runtime because new stack frames have to be allocated for each recursive call. Both iteration and recursion are ways to break a large problem into chunks - depending on the nature of the problem and the average expected size of input, one may be a better choice than the other. The recursion-based algorithm for n-Queens

outlined in the previous section works well for boards sized 15 or smaller. A stack can be used to implement a non-recursive solution to the n-Queens problem. See Algorithm 2 for the pseudocode.

Algorithm 1 Finding All Solutions to n-Queens with Recursion

```
1: function findSolutions(board, n, row)
2:     if row > n then          → base case
3:         solution found :)
4:         return
5:     end if
6:     for column = 1 to n do
7:         if board[row][column] not under threat then
8:             place Queenq
9:             findSolutions(board, n, row + 1)
10:            remove Queen
11:        end if
12:    end for
13: end function
```

Algorithm 2 Finding All Solutions to n-Queens without Recursion

```
1: function findSolutions(board, n, row)
2:     create stack
3:     for column = 1 to n do
4:         push frame to stack for (row=1, column, placed = False)
5:     end for
6:     while stack is not empty do
7:         top = stack.top()
8:         if top.row > n then
9:             solution found!
10:            stack.pop()
11:        else if top.placed then
12:            remove queen at (row, column)
13:            stack.pop();
14:        else if !top.placed and top.(row, column) is not under threat then
15:            place Queen
16:            top.placed = true
17:            for column = 1 to n do
18:                if cell (top.row + 1, column) is safe then
19:                    push frame to stack for (row + 1, column, placed = False)
20:                end if
21:            end for
22:        end if
23:    end while
24: end function
```