

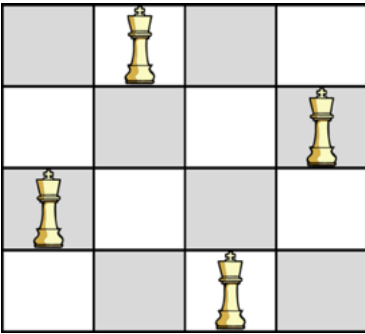
Lab 9: Using Recursion to Solve the N-Queens Problem

Due Monday by 11:59pm

Points 100

Submitting an external tool

In this lab exercise, we will hone our recursions skills while solving the famous n-queens problem. Specifically, two queens on a chess board can attack each other if one piece can reach the other by moving an arbitrary number of squares either horizontally, vertically, or diagonally. The n-queens problem asks us to place n queens on an $n \times n$ chess board so that no queen is attacking any other queen. An example of a solution to the 4-queens problem is below.



There can be many solutions to the n-queens problem. For $n = 8$, for example, there are 92 distinct solutions. Your task with this lab is to compute the number of solutions for any small value of n as quickly as possible.


To count all possible solutions, you should generate them recursively. For example, note that there must be exactly one queen in each row. We can therefore generate solutions row-by-row. That is, we loop through all possible locations for the queen in the first row, then recursively complete the board by checking for each of these all possible locations in the second row, and so on. In code, there are several ways we can write this; for instance, we can maintain the current state of the board in a global (possibly 2D) array, and write a function **checkRow(int r)** that tries all possibilities for row r one by one, with each of them calling **checkRow(r+1)** to recursively complete the board. You should feel welcome to experiment with different approaches to try and find the fastest one.


In order to achieve as much speed as possible, it will be important to prune your recursive search as early as possible. That is, any time you reach a state where two queens are attacking each other, don't recurse any further, but rather exit the current function and continue searching other alternatives.

Be sure to also exploit the fact that the problem is symmetric to cut down on the search time by a factor of two, by only trying half of the possible locations for the queen in the top row (be careful here if n is odd!)

Download the starter code - **nQueens.cpp** - which contains a function by the same name and a main function to test **nQueens()**. This function should take a single integer n as a parameter and return a single integer specifying the number of distinct solutions to the n -queens problem. You are free to implement any helper functions for **nQueens()**. Submit **nQueens.cpp** with the implemented **nQueens()** function.

[lab 9 slides.pptx](https://clemons.instructure.com/courses/195558/files/18699753?wrap=1) (<https://clemons.instructure.com/courses/195558/files/18699753?wrap=1>)_ 
(https://clemons.instructure.com/courses/195558/files/18699753/download?download_frd=1)

[Lab 9 - Using Recursion to Solve the N-Queens Problem Starter Code.zip](https://clemons.instructure.com/courses/195558/files/18699757?wrap=1)
(<https://clemons.instructure.com/courses/195558/files/18699757?wrap=1>)_ 
(https://clemons.instructure.com/courses/195558/files/18699757/download?download_frd=1)

[Lab 9 Sample Test Cases.zip](https://clemons.instructure.com/courses/195558/files/18699759?wrap=1) (<https://clemons.instructure.com/courses/195558/files/18699759?wrap=1>)_ 
(https://clemons.instructure.com/courses/195558/files/18699759/download?download_frd=1) - Main functions for a subset of the test cases on Gradescope (**main()** function in **nQueens.cpp** must be commented out): T#.cpp is the main function for test case #.

When submitting nQueens.cpp to Gradescope for this assignment or running the sample test cases be sure to comment out the main() function in nQueens.cpp.

This tool needs to be loaded in a new browser window

Load Lab 9: Using Recursion to Solve the N-Queens Problem in a new window

