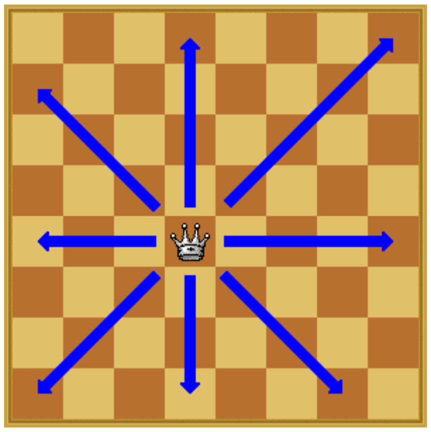
**CSE 15**
**Introduction to Data Structures**
**Programming Assignment 2**

In this project, you will write a C program that uses recursion to find all solutions to the $n$-Queens problem, for $1 \leq n \leq 15$. (Students who took CMPS 12A from me worked on an iterative, non-recursive approach to this same problem in Java. You can see it at https://classes.soe.ucsc.edu/cmps012a/Spring18/pa5.pdf.)

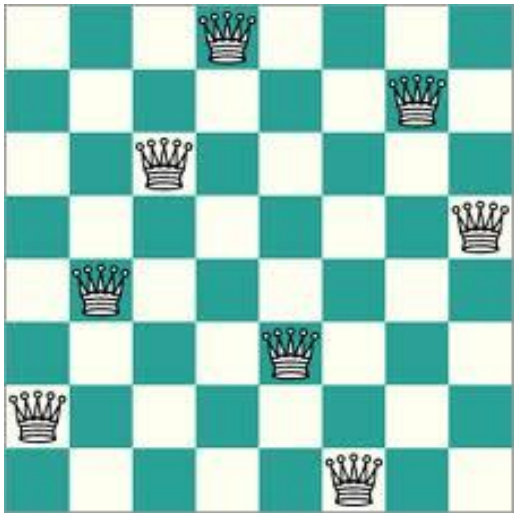
Begin by reading the Wikipedia article on the Eight Queens puzzle at:

http://en.wikipedia.org/wiki/Eight_queens_puzzle

In the game of Chess a queen can move any number of spaces in any linear direction: horizontally, vertically, or along a diagonal.

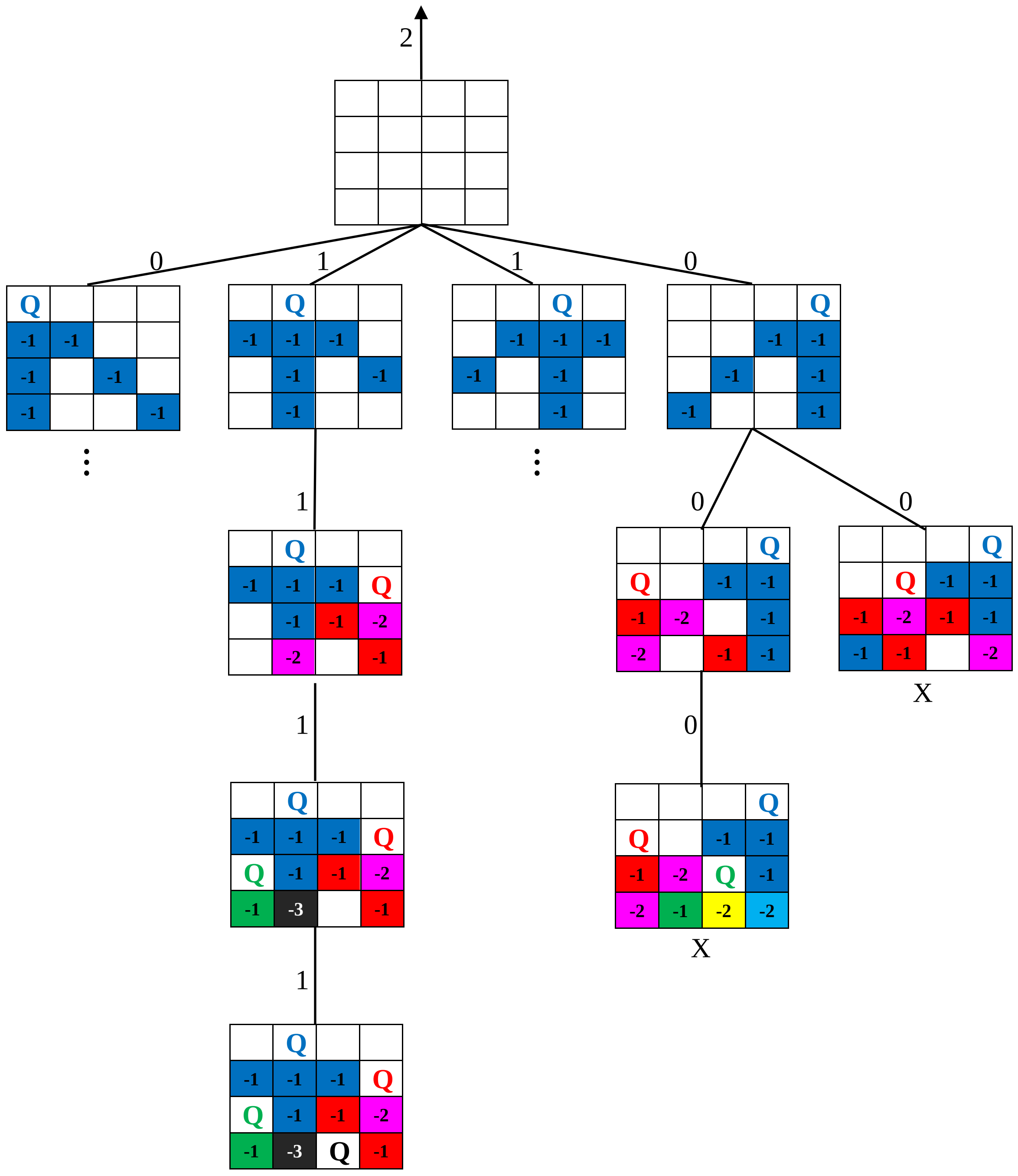


The Eight Queens puzzle is to find a placement of 8 queens on an otherwise empty $8 \times 8$ chessboard in such a way that no two queens confront each other. In particular, no two queens lie on the same row, column or diagonal. One solution to this problem is pictured below.

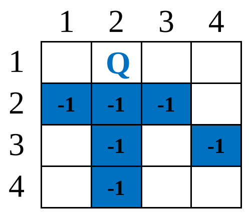


The $n$-Queens problem is the natural generalization, in which $n$ queens are placed on an $n \times n$ chessboard in such a way that no two queens lie on the same row, column or diagonal. There are many ways of solving this problem, some of which are described in the Wikipedia article linked above. Our algorithm will recursively locate a square on the next row down, where a queen can be placed, without being attacked by (or attacking) a previously placed queen. The illustration below is a partial box trace of the case $n = 4$.

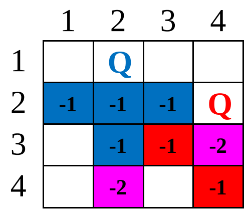
At the top (level 0) of the recursion, we have placed no queens, illustrated by an empty $4 \times 4$ chessboard. Notice that initially, there are no restrictions on where a queen can be placed on row 1. At level 1 of the recursion therefore, we place a queen on each of the 4 safe positions in row 1. These 4 queen placements generate 4 separate recursive calls.



In the above illustration we have color-coded the queens, so that the queen on row 1 is blue, that on row 2 is red, that on row 3 is green and the queen on row 4 is black. For each queen placement, it is necessary to keep track of which squares it attacks on the rows below it. To this end, the squares on rows 2, 3 and 4 that are attacked by the blue queen, are shaded blue, and no subsequent queens can be placed on those squares. For instance, after placing a blue queen on row 1, column 2, we have:

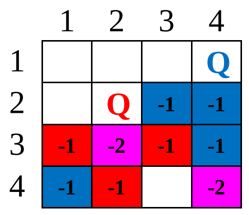


Observe that there is only one safe square in row 2, namely square (2, 4). Therefore the placement of a queen on (1, 2) generates just one recursive call, and this box has only one child in the box trace, in which a red queen is placed on (2, 4). The squares on rows 3 and 4 that are attacked by the red queen are of course shaded red. Note however that two of these squares, (3, 4) and (4, 2), are attacked by both the red queen and the blue queen, and are accordingly shaded purple (see https://www.youtube.com/watch?v=90bhBUJRY20).



In what follows, we'll see that it is most important to keep track of the *number* of queens attacking a given square from a row above. We signify this in the box trace by placing the *negative* of that number in the given square. Thus (3, 2) contains -1 since it is attacked by only the blue queen, (3, 3) contains -1 since it is attacked by only the red queen, and (3, 4) contains -2 since it is attacked by both.

If we succeed in placing a queen on row 4, then we have found a solution to 4-queens, and the algorithm returns value 1 from that box. If the next row contains no safe squares, then the box generates no recursive calls, and returns 0. For instance, after placing a red queen on (2, 2) below, row 3 has no safe square on which to place a queen. We signify this by placing an X under the dead-end box.



In general, each box returns the sum of the returned values of each of its children. As an exercise, complete the box trace on the preceding page by determining the children of the boxes with the symbol $\vdots$ under them. (Don't try to determine the colors, which were given here only for illustration, but do try to determine the number of queens attacking each square from above.) As a further (more challenging) exercise, try to construct a complete box trace for the case $n = 5$, which has 10 solutions.

Let $a(n)$ denote the number of solutions to the $n$-Queens problem. Much research has been done on the sequence $\left(a(n)\right)_{n=0}^{\infty}$, which begins $(1, 1, 0, 0, 2, 10, 4, 40, 92, 352, 724, 2680, 14200, \ldots)$. See the article https://oeis.org/A000170 for links to some of this work. The algorithm we are studying in this project is practical for values of $n$ up to $n = 15$, or 16 at most. Beyond that, it slows significantly. (I managed to compute $a(16) = 14{,}772{,}512$ in just over 2 minutes on my Windows machine.) The iterative algorithm we studied in CMPS 12A was useful only up to about $n = 13$.

**Program Operation**

Your program for this project will be called Queens.c. You will include a Makefile that creates an executable file called Queens, allowing one to run the program by typing `Queens` at the command line. Your program will read an integer *n* from the command line, indicating the size of the Queens problem to solve. The program will operate in two modes: normal and verbose (which is indicated by the command line option "`-v`"). In normal mode, the program prints only the *number* of solutions to *n*-Queens. In verbose mode, *all solutions* to *n*-Queens will be printed in the order they are found by the algorithm, and in a format described below, followed by the *number* of such solutions. Thus to find the number of solutions to 8-Queens you will type:

```
$ Queens 8
```

To print all 92 unique solutions to 8-Queens type:

```
$ Queens -v 8
```

If the user types anything on the command line other than the option `-v` and a number $n$, the program will print a usage message to stderr and quit. A sample session is included below (remember that `$` is the unix prompt and you do not type it.)

```
$ Queens
Usage: Queens [-v] number
Option: -v  verbose output, print all solutions
$ Queens blah
Usage: Queens [-v] number
Option: -v  verbose output, print all solutions
$ Queens 4
4-Queens has 2 solutions
$ Queens -v 4
(2, 4, 1, 3)
(3, 1, 4, 2)
4-Queens has 2 solutions
$ Queens -v 5
(1, 3, 5, 2, 4)
(1, 4, 2, 5, 3)
(2, 4, 1, 3, 5)
(2, 5, 3, 1, 4)
(3, 1, 4, 2, 5)
(3, 5, 2, 4, 1)
(4, 1, 3, 5, 2)
(4, 2, 5, 3, 1)
(5, 2, 4, 1, 3)
(5, 3, 1, 4, 2)
5-Queens has 10 solutions
$
```

Solutions are encoded as an $n$-tuple where the $i^{\text{th}}$ element is the column number of the queen residing in row $i$. For instance, in the case $n = 4$, the 4-tuple `(2, 4, 1, 3)` encodes the solution:

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |   | Q |   |   |
| 2 |   |   |   | Q |
| 3 | Q |   |   |   |
| 4 |   |   | Q |   |

and the 4-tuple `(3, 1, 4, 2)` encodes the other solution:

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 |   |   | Q |   |
| 2 | Q |   |   |   |
| 3 |   |   |   | Q |
| 4 |   | Q |   |   |

Observe that these solutions are given in the order in which they would be found by our algorithm, as illustrated by the (partial) box trace on page 2 of this document, provided we work from left to right in each row. It is recommended that you write helper functions to perform basic subtasks such as printing the usage message then quitting.

**Program Representation of the Chessboard**

Your program will represent an $n \times n$ chessboard by a 2-dimensional int array of size $(n + 1) \times (n + 1)$. The extra row and column are there so row and column numbers in the array correspond directly with row and column numbers on the chessboard. Specifically, if the array is called `B`, then `B[i][j]` corresponds to the square in row $i$, column $j$ of the chessboard. The following encoding of chessboard states will be used. For $1 \le i \le n$ and $1 \le j \le n$,

$$B[i][j] = \begin{cases} \phantom{-}1 & \text{if square } (i, j) \text{ contains a queen} \\ \phantom{-}0 & \text{if square } (i, j) \text{ is empty, and not under attack from any square above it} \\ -k & \text{if square } (i, j) \text{ is under attack from } k \text{ queens lying above it} \end{cases}$$

In this context, "above" means "having a smaller row number". Furthermore, since column 0 does not correspond to anything on the chessboard, we will use it to encode a solution in the required format. For $1 \le i \le n$ and $j = 0$,

$$B[i][0] = \begin{cases} 0 & \text{if row } i \text{ is contains no queen} \\ j & \text{if square } (i, j) \text{ contains a queen} \end{cases}$$

Thus, to print out a solution, enter a loop that prints: $(B[1][0], B[2][0], B[3][0], \dots, B[n][0])$. The contents of row 0 are not specified, so you can put anything you like in $B[0][0 \cdots n]$.

You may wish to write a helper function to initialize this array to all zeros, though this is not required. Your program will contain two functions with the following headings:

```
void placeQueen(int** B, int n, int i, int j)
```

and

```
void removeQueen(int** B, int n, int i, int j)
```

The first function increments $B[i][j]$ from its initial value of 0 to 1, and sets $B[i][0]$ to $j$, thus indicating the existence of a queen on square $(i, j)$. It will also decrement $B[k][l]$ for every square $(k, l)$ under attack from the new queen at $(i, j)$, where $i < k \leq n$ and $1 \leq l \leq n$. The second function undoes everything the first function does. So, it decrements $B[i][j]$ from 1 to 0, resets $B[i][0]$ from $j$ to 0, and increments $B[k][l]$ for every square $(k, l)$ no longer under attack from the now absent queen at $(i, j)$, where $i < k \leq n$ and $1 \leq l \leq n$. All of this serves to implement the encoding of states described above. Your program will also contain a function with the heading

```
void printBoard(int** B, int n)
```

that prints out a solution to $n$-queens in the format described above. Finally, your program will contain a function with the heading

```
int findSolutions(int** B, int n, int i, char* mode)
```

that implements the recursive algorithm we have been describing. If the C string argument `mode` has the value "`verbose`", then `findSolutions()` will print out solutions as it finds them by calling the function `printBoard()`. If mode has any other value, `findSolutions()` will print nothing. The int returned by `findSolutions()` is the number of solutions to $n$-queens that have queens placed on rows 1 to $(i - 1)$, as represented by the current state of array `B[][]`. A top level call to `findSolutions()` that does not print solutions would be `findSolutions(B, 1, "")`. High level pseudo-code for this function is given below.

```
1.  if $i > n$ (a queen was placed on row $n$, and hence a solution was found)
2.      if we are in verbose mode
3.          print that solution
4.      return 1
5.  else
6.      for each square on row $i$
7.          if that square is safe
8.              place a queen on that square
9.              recur on row $(i + 1)$, then add the returned value to an accumulating sum
10.             remove the queen from that square
11. return the accumulated sum
```

Your program's function `main()` will read the command line arguments, then determine the value of $n$, and what mode (normal or verbose) to run in. It will then allocate an int array from heap memory of size $(n + 1) \times (n + 1)$ and initialize it to all zeros. (See lab assignment 2 for a discussion of dynamic memory allocation in C.) Function `main()` will then call `findSolutions()` on this array in the correct mode, and print out the number of solutions to $n$-queens that were found.

**What to turn in**

Write a Makefile for this project that creates an executable file called `Queens`, and that includes a `clean` target (as in lab1), and a `memcheck` target that runs your program under valgrind to check for memory leaks (see lab2). Submit the files Makefile and Queens.c to the assignment name pa2. As always start early and ask questions in lab sessions and on Piazza.