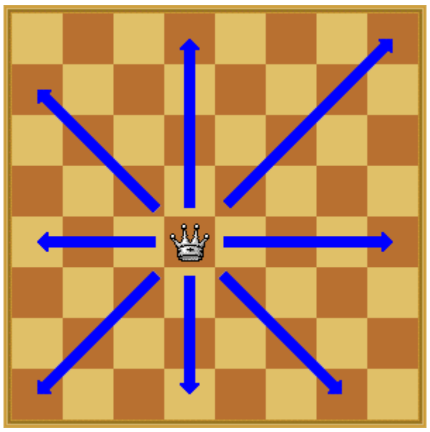
CSE 30
Programming Abstractions: Python
Programming Assignment 2

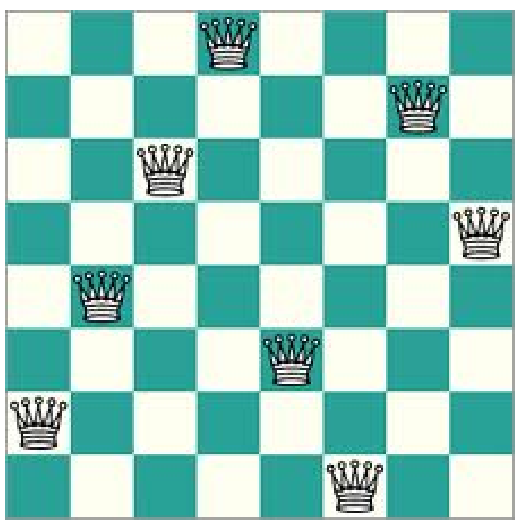
In this project, you will write a Python program that uses recursion to find all solutions to the n-Queens problem, for $1 \le n \le 13$. Begin by reading the Wikipedia article on the Eight Queens puzzle at:

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.

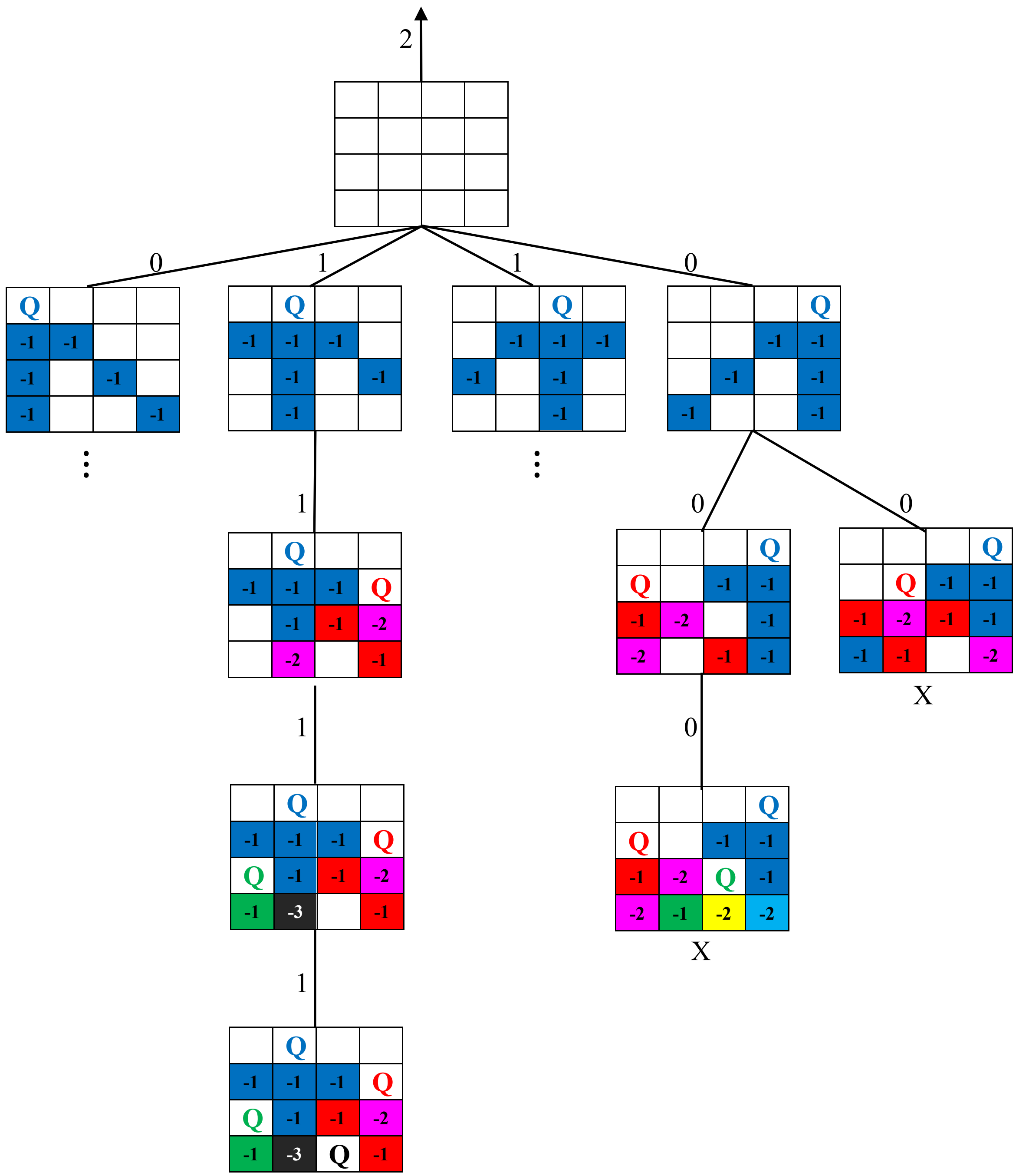


The Eight Queens puzzle is to find a placement of 8 queens on an otherwise empty 8×8 chessboard in such a way that no two queens confront each other. In particular, no two queens lie on the same row, column or diagonal. One solution to this problem is pictured below.



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 × 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:

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

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 thus shaded purple (see https://www.youtube.com/watch?v=90bhBUJRY20).

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

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. Think of the number placed in a square as a measure of how safe that square is, so a negative number means unsafe.

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.

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

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. 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 ⋮ 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 $(1, 1, 0, 0, 2, 10, 4, 40, 92, 352, 724, 2680, 14200, 73712, \ldots)$. See the article https://oeis.org/A000170 for links to some of this work. The Python program you will build in this assignment is practical for values of n up to 13, or 14 at most. Beyond that, it slows down significantly. (I was able to compute $a(13) = 73{,}712$ in about 20 seconds, and $a(14) = 365{,}596$ in under 2 minutes on my Windows machine.)

Program Operation

Your program for this project will be called Queens.py. It 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:

```
$ python3 Queens.py 8
```

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

```
$ python3 Queens.py -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.)

```
$ python3 Queens.py
Usage: python3 Queens.py [-v] number
Option: -v  verbose output, print all solutions
$ python3 Queens.py blah
Usage: python3 Queens.py [-v] number
Option: -v  verbose output, print all solutions
$ python3 Queens.py 4
4-Queens.py has 2 solutions
$ python3 Queens.py -v 4
(2, 4, 1, 3)
(3, 1, 4, 2)
4-Queens has 2 solutions
$ python3 Queens.py -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^{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	

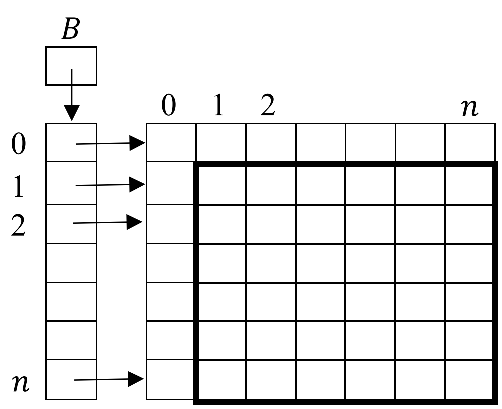
and the 4-tuple `(3, 1, 4, 2)` encodes the other solution:

	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 list of lists of ints. Each list will be of length $n + 1$. We can think of this as a 2-dimensional table with $n + 1$ rows and $n + 1$ columns. Denoting this table as B, we may picture it in memory as



The extra row and column are there so row and column numbers in the table correspond directly with row and column numbers on the chessboard. Specifically, cell $B[i][j]$ corresponds to the square in row i, column j of the chessboard. We use the following encoding of chessboard states. For $1 \leq i \leq n$ and $1 \leq j \leq n$,

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

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 create this list of lists and initialize it to all zero, though this is not required. Your program will contain two functions called

`placeQueen(B, i, j)` and `removeQueen(B, i, j)`

respectively. 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 $i < k \le n$ and $1 \le l \le n$. 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 $i < k \le n$ and $1 \le l \le n$. All of this serves to implement the encoding of states described above. Your program will also contain a function called `printBoard(B)` that prints out a solution to n-queens in the format described above.

Finally, your program will contain a function called `findSolutions(B, i, mode)` that implements the recursive algorithm we've been describing. If the string argument `mode` has the value "`verbose`", then `findSolutions()` will print solutions as it finds them by calling `printBoard()`. If mode has any other value, `findSolutions()` will print nothing. The function call `findSolutions(B, i, mode)` will return an int that is the number of solutions to n-queens which have queens placed on rows 1 to $(i - 1)$, as represented by the current state of the table B. 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 will read the command line arguments, then determine the value of n, and what mode (normal or verbose) to run in. It will create a list of lists (or table) of size $(n + 1) \times (n + 1)$, initialize it to all zeros, call function `findSolutions()` on this table in the correct mode, then print out the number of solutions to n-queens that were found.

Submit the program Queens.py to Gradescope before the due date. The Examples section of the webpage will contain a folder called pa2 that will contain some helpful examples, as well as some sample output for the program. This project is considerably more challenging than pa1, so start early and ask questions in office hours and on Piazza.