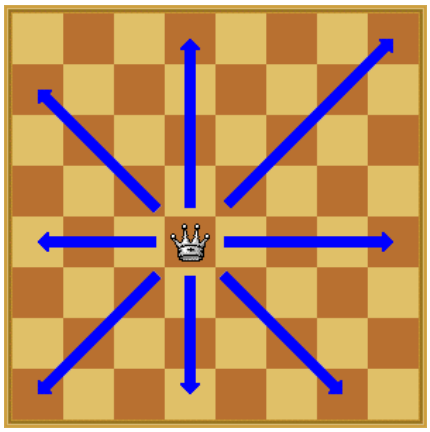
CMPS 12A
Introduction to Programming
Programming Assignment 5

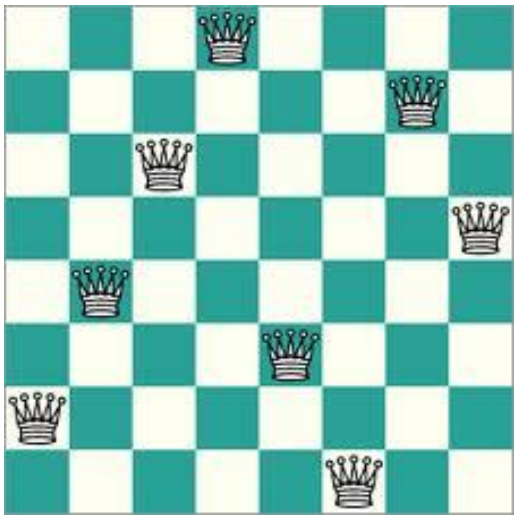
In this assignment you will write a Java program that finds all solutions to the *n*-Queens problem, for $1 \le n \le 13$. Begin by reading the Wikipedia article on the Eight Queens puzzle at:

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

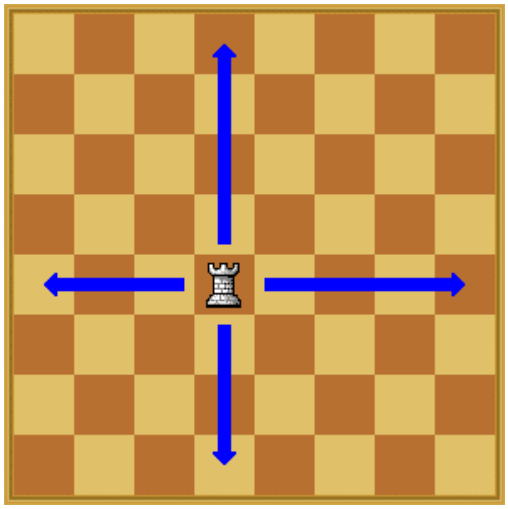
In the game of Chess a queen can move any number of spaces in any linear direction: horizontally, vertically, or along a diagonal.



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. One solution to this problem is pictured below.



The *n*-Queens problem is the natural generalization of placing *n* queens on an $n \times n$ chessboard so that no two queens lie on the same row, column or diagonal. There are many ways of solving this problem. Our approach will be to start with a solution to the *n*-rooks problem (i.e. place *n* Rooks on an $n \times n$ chessboard so that no two rooks attack each other) then check if that arrangement is also a solution to *n*-Queens. The rook move in chess is similar to the queen's move except that it cannot move diagonally.



Solutions to the *n*-Rooks problem are easy to find since one need only position the rooks so that no two are on the same row and no two are on the same column. Since there are *n* rows and *n* columns to choose from, solutions abound.

A natural way to encode solutions to *n*-Rooks is as permutations of the integers {1, 2, 3, …, *n*}. A *permutation* of a set is an ordered arrangement of the elements in that set. If we number the rows and columns of the $n \times n$ chessboard using the integers 1 through *n*, each square is then labeled by a unique pair of coordinates (*i*, *j*) indicating the square in row *i* and column *j*. The permutation $(a_1, a_2, a_3, \ldots, a_n)$ corresponds to the placement of a piece on the square with coordinates (a_j, j) for $1 \le j \le n$. For instance, the permutations $(2, 7, 8, 5, 1, 4, 6, 3)$ and $(2, 4, 6, 8, 3, 1, 7, 5)$ correspond to two the 8-Rooks solutions pictured below.

8			R					
7		R						
6							R	
5				R				
4						R		
3								R
2	R							
1					R			
	1	2	3	4	5	6	7	8

8				R				
7							R	
6			R					
5								R
4		R						
3					R			
2	R							
1						R		
	1	2	3	4	5	6	7	8

Observe that the solution on the right is also a solution to 8-Queens, while the one on the left is not, since certain pairs of rooks lie on the same diagonal (such as the rooks on (7, 2) and (5, 4).) In fact the solution on the right is the 8-Queens solution pictured on the previous page. In general, any solution to the *n*-Queens problem is also a solution to *n*-Rooks, but the converse is false. Not every solution to *n*-Rooks is a solution to *n*-Queens.

Your program will generate all solutions to *n*-Rooks by systematically producing all permutations of the set {1, 2, 3, …, *n*}. It will check each *n*-Rooks solution for diagonal attacks to see if the given permutation

also solves *n*-Queens. Whenever an *n*-Queens solution is found, your program will print out the permutation, then move on to the next permutation. Thus when $n=8$, $(2, 4, 6, 8, 3, 1, 7, 5)$ would be printed while $(2, 7, 8, 5, 1, 4, 6, 3)$ would not. Two major problems must therefore be solved to complete the project: (1) how can you produce all permutations of the set {1, 2, 3, …, *n*}, and (2) given one such permutation how can you determine if two pieces lie on the same diagonal.

Permutations

There are $n!$ permutations of a finite set containing *n* elements. To see this, observe that there are *n* ways to choose the first element in the arrangement, $n-1$ ways to choose the second element, $n-2$ ways to choose the third, … , 2 ways to choose the $(n-1)^{\text{th}}$ element, and finally 1 way to choose the n^{th} and last element in the ordered arrangement. The number of ways of making all these choices in succession is therefore $n(n-1)(n-2)\cdots 3\cdot 2\cdot 1 = n!$. For instance there are $3!=6$ permutations of the set {1, 2, 3}: 123, 132, 213, 231, 312, 321. The permutations of a finite set {1, 2, 3, …, *n*} have a natural ordering called the *lexicographic order*, or alphabetic order. The $4!=24$ permutations of {1, 2, 3, 4} are listed in order as follows.

1234	2134	3124	4123
1243	2143	3142	4132
1324	2314	3214	4213
1342	2341	3241	4231
1423	2413	3412	4312
1432	2431	3421	4321

As an exercise list the $5!=120$ permutations of {1, 2, 3, 4, 5} in lexicographic order. After finishing this long exercise, you will see the need for an algorithm that systematically produces all permutations of a finite set. We will represent a permutation $(a_1, a_2, a_3, \ldots, a_n)$ by an array $A[\,]$ of length $n+1$, where $A[j]=a_j$ for $1\le j\le n$, and the element $A[0]$ is simply not used. Your program will include a function with the following heading.

```
static void nextPermutation(int[] A){. . .}
```

This method will alter its argument *A* by advancing (*A*[1], *A*[2], *A*[3], …, *A*[*n*]) to the next permutation in the lexicographic ordering. If (*A*[1], *A*[2], *A*[3], …, *A*[*n*]) is already at the end of the sequence, the function will reset *A* to the initial permutation $(1, 2, 3, \ldots, n)$ in the lexicographic order. The pseudo-code below gives an outline for the body of `nextPermutation()`.

```
scan the array from right-to-left
    if the current element is less than its right-hand neighbor
        call the current element the pivot
        stop scanning
if the left end was reached without finding a pivot
    reverse the array (permutation was lexicographically last, so start over)
    return
scan the array from right-to-left again
    if the current element is larger than the pivot
        call the current element the successor
        stop scanning
swap the pivot and the successor
reverse the portion of the array to the right of where the pivot was found
return
```

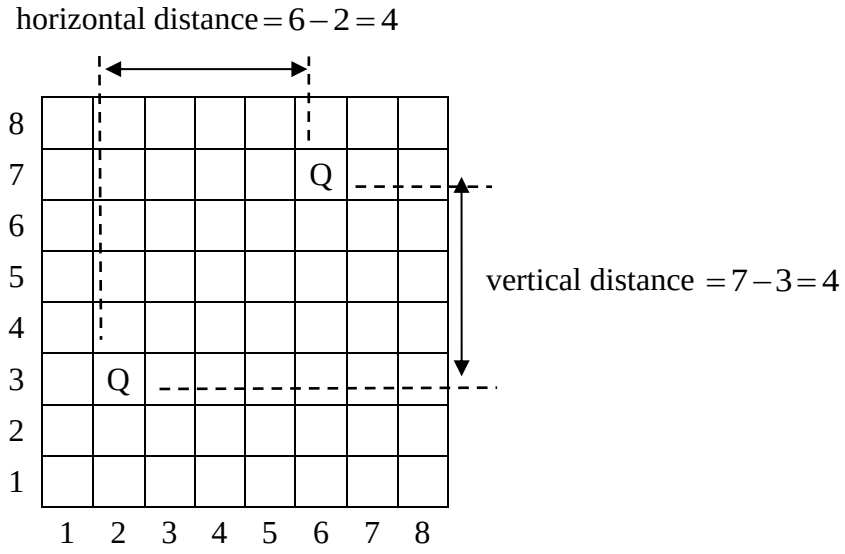
Run the above procedure by hand on the initial permutation (1, 2, 3, 4) and see that it does indeed produce all 24 permutations of the set {1, 2, 3, 4} in lexicographic order. Also check that (4, 3, 2, 1) which is the final permutation in lexicographic order, is returned to the initial state (1, 2, 3, 4).

Finding Diagonal Attacks

Your program will also include another function with the following heading.

```
static boolean isSolution(int[] A){. . .}
```

This method will return true if the permutation represented by $(A[1], A[2], A[3], \ldots, A[n])$ places no two queens on the same diagonal, and will return false otherwise. To check if two queens at $(A[i], i)$ and $(A[j], j)$ lie on the same diagonal, it is sufficient to check whether their horizontal distance apart is the same as their vertical distance apart, as illustrated in the diagram below.



Function `isSolution()` should compare each pair of queens at most once. If a pair is found on the same diagonal, do no further comparisons and return false. If all $n(n-1)/2$ comparisons are performed without finding a diagonal attack, return true. (Question: why is the number of 2-element subsets of an *n* element set exactly $n(n-1)/2$?)

Program Operation

Your program for this project will be called Queens.java. You will include a Makefile that creates an executable Jar 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 permutations representing solutions to *n*-Queens will be printed in lexicographic order, 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.

```
% Queens
Usage: Queens [-v] number
Option:  -v   verbose output, print all solutions
% Queens x
Usage: Queens [-v] number
Option:  -v   verbose output, print all solutions
% Queens 5
5-Queens has 10 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
% Queens -v 6
(2, 4, 6, 1, 3, 5)
(3, 6, 2, 5, 1, 4)
(4, 1, 5, 2, 6, 3)
(5, 3, 1, 6, 4, 2)
6-Queens has 4 solutions
% Queens -v 4
(2, 4, 1, 3)
(3, 1, 4, 2)
4-Queens has 2 solutions
% Queens -v 3
3-Queens has 0 solutions
%
```

It is recommended that you write helper functions to perform basic subtasks such as: print the usage message and quit, calculate the factorial of *n*, print out a formatted array as above, swap two elements in an array, and reverse the elements in a subarray. Some of these methods have already been posted on the website as examples.

There are more efficient procedures for solving *n*-queens that use programming techniques beyond the scope of this course (such as recursion). Even if you know how to use these techniques, you are required to solve the problem as indicated in this project description. Your program should work very quickly on problem sizes up to 12 or 13. Beyond that, you should expect your program to slow down considerably.

What to turn in

Write a Makefile for this project that creates an executable Jar file called `Queens`, and that includes a `clean` target (as in lab4). Submit the files Makefile and Queens.java to the assignment name pa5. As always start early and ask questions of myself, the TAs and on Piazza.