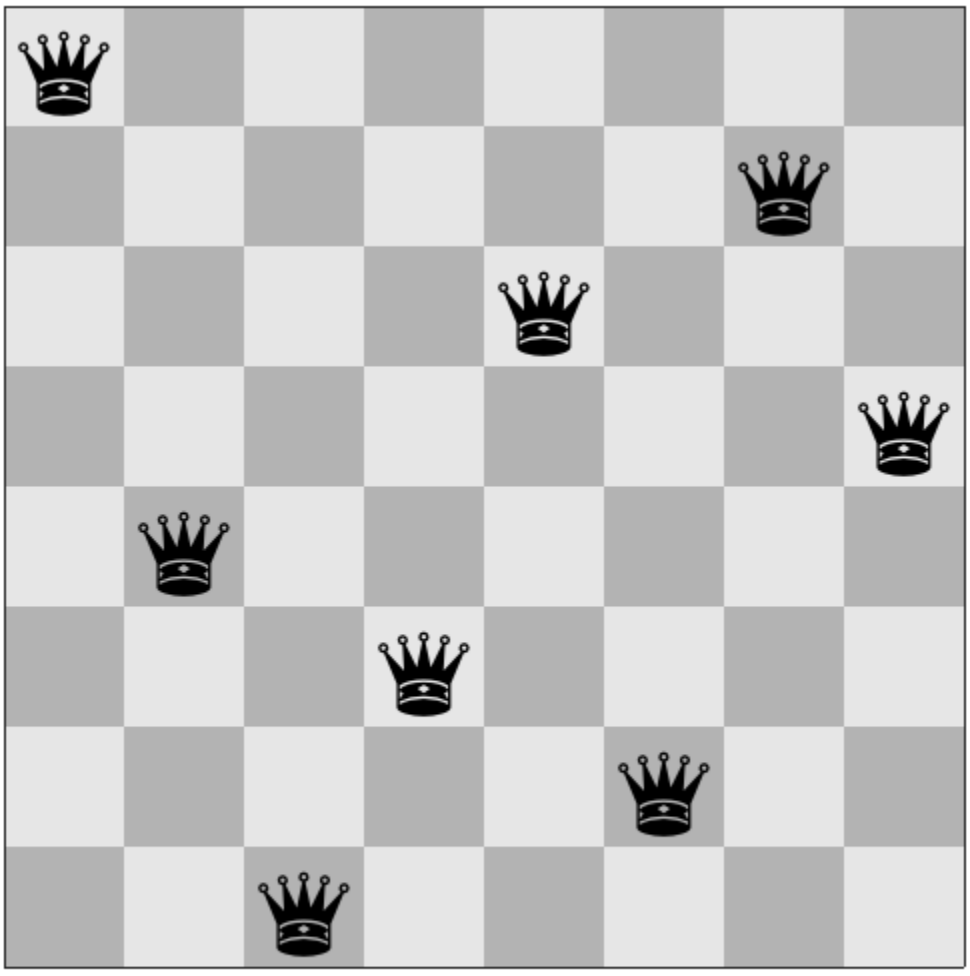
Assignment 3: Eight Queens problem.

Authors: Kevin de la Coba Malam, Juan Luis Sanz Calvar

Pair 01. Group 2392.

In this problem we have a N number of queens in a NxN board. Each queen must be placed so it cannot be taken from another queen, queens cannot be in any of the diagonals, columns, or rows of any other else.

The solution is represented in the code as a list, in this list we have N elements with values going from 1 to N from bottom to top, and also from 1 to N from left to right. This means that if the first value has 6, the first queen is placed in [1,6], if the second value is 3, the second queen is in [2,3].

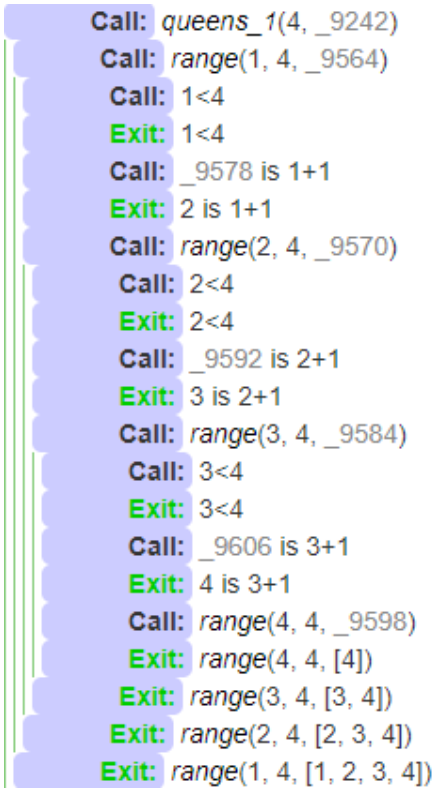


Eddins, S. (2017, 5 mayo). The Eight Queens Problem – Generating All Solutions. LaptrinhX. https://laptrinhx.com/the-eight-queens-problem-generating-all-solutions-627809323/

This is one example of a solution, the list would have the following values: **[8, 4, 1, 3, 6, 2, 7, 5]**.

The code given contains 2 solutions. The first one is divided in 3 predicates and the second one is divided in 2 predicates. Both solutions use permutations, this is useful because it fulfils the condition where none of the queens are in the same columns or rows.

Let's start by analyzing the first solution. In this solution the first predicate is **`range(1, N, Rs)`**, where N is the width/heigh of the board and Rs is an empty variable. The goal of this predicate is to build a list in Rs which goes from 1 to N.



As if it was a loop, the execution starts by checking if an "index" is less than N, while it is, the predicate is going to call itself until it reaches the end. Once the end is reached the list is built.

The second predicate is **`permu(Rs, Qs)`**. The previous predicated generated a list, this list is the first argument **Rs**, the second argument is a permutation of the previous list. Every time this predicate is called a new permutation is given; in the trace it could be seen that the following order was followed:

1. The fourth element is swapped with third one.
2. The third element with the second one and then the "new" third one with the fourth one.
3. The same movement as in [1].
4. The same movement as in [2].
5. The same movement as in [1, 3].
6. The second element with the first one and then the "new" first one with the last one.
7. The same movement as in [1, 3, 5].
8. The same movement as in [2, 4].
9. The same movement as in [1, 3, 5, 7].
10. …

```
Exit: permu([1, 2, 3, 4], [1, 4, 3, 2])
Call: test([1, 4, 3, 2])
Fail: test([1, 4, 3, 2])
Exit: permu([1, 2, 3, 4], [2, 1, 3, 4])
Call: test([2, 1, 3, 4])
Fail: test([2, 1, 3, 4])
Exit: permu([1, 2, 3, 4], [2, 1, 4, 3])
```

Each time that **`permu(Rs, Qs)`** is called we have a different permutation.

What we conclude is that first it does all the possible permutations of the 2 last elements (2 permutations) by doing all the needed swaps between those elements, in this case just one. After all the permutations with the last 2 elements are done, we start doing the permutations with the last 3 elements, we already did 2 of 6. Once these were done, we start with the last 4 elements, we need to do 24. We can see that we the goal is to do all the N! possible permutations. But how are they generated in the code? To answer this question, we need to analyze the final predicate.

The final predicate is **`test(Qs)`**. This predicate uses the generated permutation and checks whether the permutation is valid. The predicate stores the diagonal cells of each queen and stores the coordinates in 2 lists, when the predicate is checking the next queen, it checks if the coordinates of the position of this queen are in any of the 2 lists, if they are we need to check another permutation.

```
Redo: test([1, 2, 3, 4], 1, [], [])
Call: _9648 is 1+1
Exit: 2 is 1+1
Call: memberchk(2, [])
Fail: memberchk(2, [])
Redo: test([1, 2, 3, 4], 1, [], [])
Call: _9662 is 1+1
Exit: 2 is 1+1
Call: test([2, 3, 4], 2, [0], [2])
Call: _9676 is 2-2
Exit: 0 is 2-2
Call: memberchk(0, [0])
Exit: memberchk(0, [0])
Fail: test([2, 3, 4], 2, [0], [2])
Fail: test([1, 2, 3, 4], 1, [], [])
Fail: test([1, 2, 3, 4])
```

In this example the permutation [1, 2, 3, 4] is not valid. First the 1 is checked, we get the diagonals [0], [2]. It's the turn to check the 2, it is seen that the value generated is 0, which is already in the previous lists. It is needed to check another permutation; this one is not valid.

Then, the second solution is an optimization of the first one. At the beginning, we start as the first solution by building a list in Rs which goes from 1 to N (using the same code definitions of the first solution, **`range(1, N, Rs)`**). After that, the second and last predicate **`permu_test(Qs, [Y|Ys], X, Cs, Ds)`** will do the job of **`permu(Rs, Qs)`** and **`test(Qs)`** at the same time, having as arguments the list returned by function range and the permutation itself. This way the code is reduced and it keeps reaching the same solution.

Informal comment

Using a permutation is definitively the correct way to start, with a permutation, the solution already fulfils 2 of the 3 conditions. From this point there are more than one solution, trying permutations until we fulfil all the conditions is the one used in the algorithm.