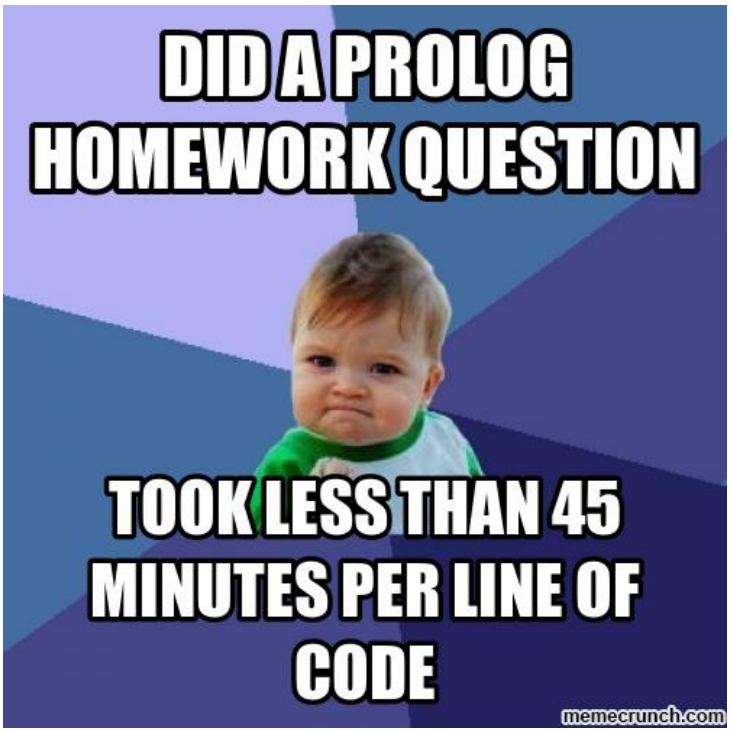
ASSIGNMENT 3

LOGIC PROGRAMMING

Miguel Álvarez Valiente & Rodrigo Juez



Escuela Politécnica Superior
Computer Science Engineering

Comment on the problem of the eight queens.

Describe the eight queens problem. Add a reference.
This is a classical problem in computer science, it consists of placing eight queens, on a chessboard that is also of size 8, so that no two queens are attacking each other, that is, no two queens are in the same row, column or diagonal.

The solution proposed in P-99 problems is generalized for any given N, which receives as an argument, and it returns a list with all the positions of N queens placed in a N sized board. The list consists of the indices per column, that is the third position of the list is the third column and the content of that index is the row, for example [3,2,1] would have a queen in positions column 1 and row 3, column 2 and row 2, column 3 and row 1.

The problem was published in 1848 by *Max Bezzel*[1] and solved by *Franz Nauck* in 1850, the solution proposed by the statement is the simplest form of that solution.

In 1972, Edsger Dijkstra used this problem to illustrate the power of what he called structured programming. He published a highly detailed description of a depth-first backtracking algorithm.

References:

[1] *W. W. Rouse Ball (1960)* "The Eight Queens Problem", in Mathematical Recreations and Essays, Macmillan, New York.

Explain the algorithm that solves the problem (not just a description but a reasoning of the steps).

The problem is splitted into smaller functions:

1. range(A, B, L): It is a very simple recursive function that generates a list by going down on recursion and adding 1 each time and then generating the list on the way up.
2. permu(X, Y): It consists of a predicate that checks if two lists are permutations of each other. We understand, because of the execution which we will later see, that even when we execute it in the terminal it only returns one solution, that is because it is true right away, but when the queens_1 is executed it tries as many times as it can until a solution is found. This is because when tests give false, it backtracks until it can choose another solution, and switches two numbers when generating the permutation.
3. test(Qs): It is the function that checks if the solution is valid, it is done by going down on the recursion checking if each element is in both lists (Cs, Ds). These lists contain the operations (X-Y) and (X+Y), these represent the different diagonals. They use memberchk which we didn't see defined in the code so we assume it's a system function.

```
% queens_1(N,Qs) :- Qs is a solution of the N-queens problem

queens_1(N,Qs) :- range(1,N,Rs), permu(Rs,Qs), test(Qs).

% range(A,B,L) :- L is the list of numbers A..B

range(A,A,[A]).
range(A,B,[A|L]) :- A < B, A1 is A+1, range(A1,B,L).

% permu(Xs,Zs) :- the list Zs is a permutation of the list Xs

permu([],[]).
% va bajando en la recursion eliminando todos los elementos para llegar a comprobar si queda algun
elemento por eliminar
permu(Qs,[Y|Ys]) :- del(Y,Qs,Rs), permu(Rs,Ys).

del(X,[X|Xs],Xs).
del(X,[Y|Ys],[Y|Zs]) :- del(X,Ys,Zs).

% test(Qs) :- the list Qs represents a non-attacking queens solution

test(Qs) :- test(Qs,1,[],[]).

% test(Qs,X,Cs,Ds) :- the queens in Qs, representing columns X to N,
% are not in conflict with the diagonals Cs and Ds

test([],_,_,_).
test([Y|Ys],X,Cs,Ds) :-
    C is X-Y, \+ memberchk(C,Cs),
    D is X+Y, \+ memberchk(D,Ds),
    X1 is X + 1,
    test(Ys,X1,[C|Cs],[D|Ds]).
```

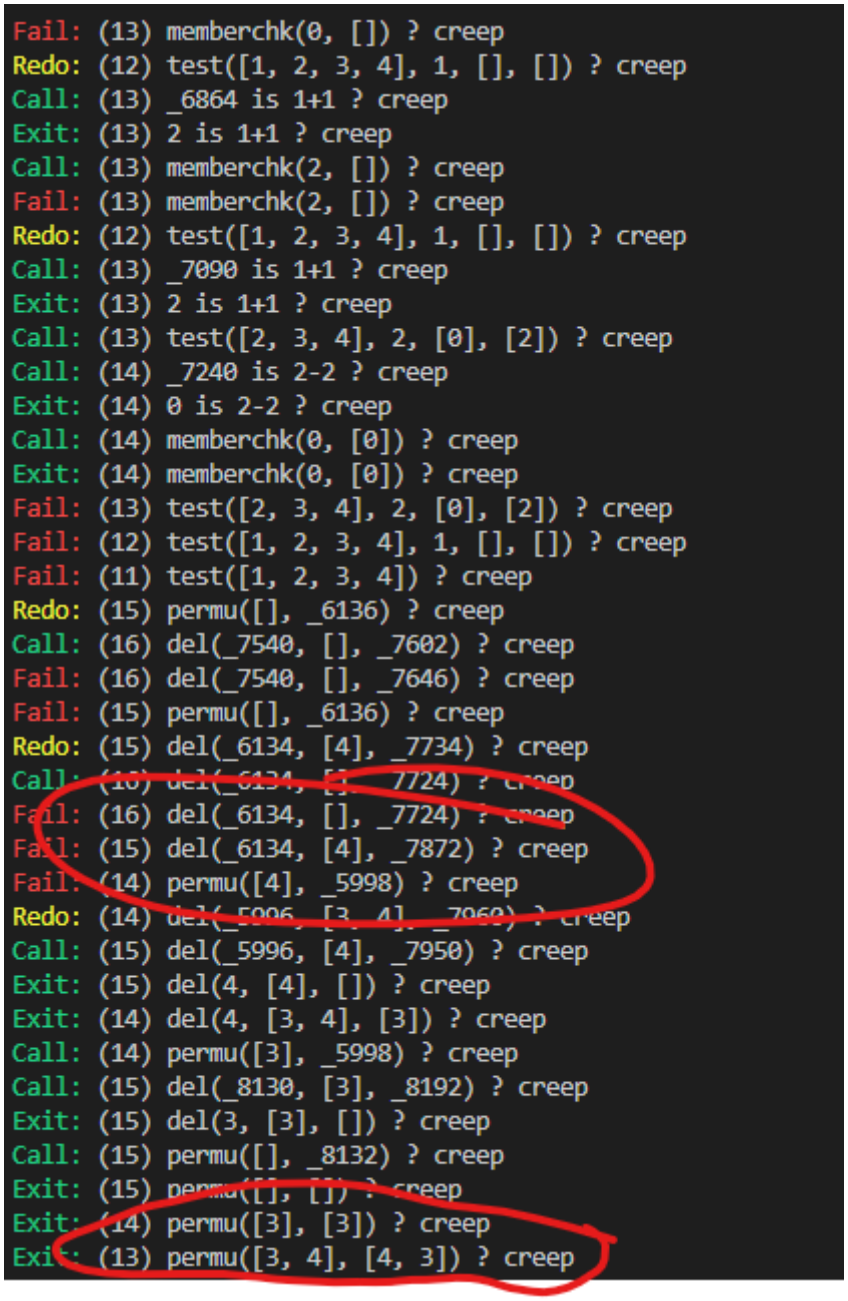
Code Execution:

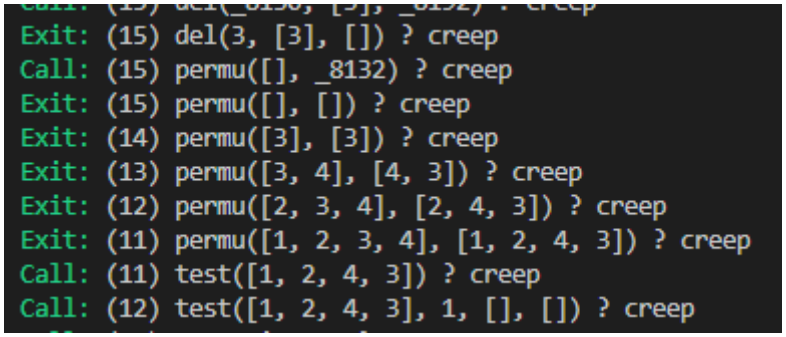
```
queens_1(4, L).
   Call: (10) queens_1(4, _4370) ? creep
   Call: (11) range(1, 4, _4810) ? creep
   Call: (12) 1<4 ? creep
   Exit: (12) 1<4 ? creep
   Call: (12) _4950 is 1+1 ? creep
   Exit: (12) 2 is 1+1 ? creep
   Call: (12) range(2, 4, _4800) ? creep
   Call: (13) 2<4 ? creep
   Exit: (13) 2<4 ? creep
   Call: (13) _5182 is 2+1 ? creep
   Exit: (13) 3 is 2+1 ? creep
   Call: (13) range(3, 4, _5032) ? creep
   Call: (14) 3<4 ? creep
   Exit: (14) 3<4 ? creep
   Call: (14) _5414 is 3+1 ? creep
   Exit: (14) 4 is 3+1 ? creep
   Call: (14) range(4, 4, _5264) ? creep
   Exit: (14) range(4, 4, [4]) ? creep
   Exit: (13) range(3, 4, [3, 4]) ? creep
   Exit: (12) range(2, 4, [2, 3, 4]) ? creep
   Exit: (11) range(1, 4, [1, 2, 3, 4]) ? creep
```

As you can see it first generates a range from 1 to 4 (N=4), simple enough and then continues generating permutations and checking if they're the solution:

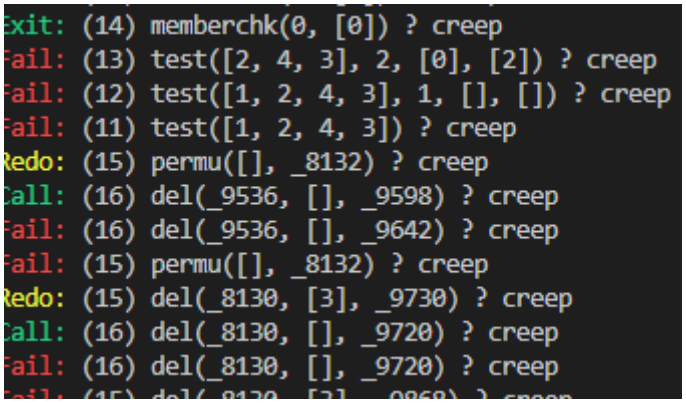
```
   Call: (11) permu([1, 2, 3, 4], _4370) ? creep
   Call: (12) del(_5720, [1, 2, 3, 4], _5782) ? creep
   Exit: (12) del(1, [1, 2, 3, 4], [2, 3, 4]) ? creep
   Call: (12) permu([2, 3, 4], _5722) ? creep
   Call: (13) del(_5858, [2, 3, 4], _5920) ? creep
   Exit: (13) del(2, [2, 3, 4], [3, 4]) ? creep
   Call: (13) permu([3, 4], _5860) ? creep
   Call: (14) del(_5996, [3, 4], _6058) ? creep
   Exit: (14) del(3, [3, 4], [4]) ? creep
   Call: (14) permu([4], _5998) ? creep
   Call: (15) del(_6134, [4], _6196) ? creep
   Exit: (15) del(4, [4], []) ? creep
   Call: (15) permu([], _6136) ? creep
   Exit: (15) permu([], []) ? creep
   Exit: (14) permu([4], [4]) ? creep
   Exit: (13) permu([3, 4], [3, 4]) ? creep
   Exit: (12) permu([2, 3, 4], [2, 3, 4]) ? creep
   Exit: (11) permu([1, 2, 3, 4], [1, 2, 3, 4]) ? creep
```

The first permutation is the same as the original range:





As you can see unfortunately when we do the test it doesn't pass and it fails, then Prolog backtracks until it can take another route (for example when generating another permutation) and tries again, it is clearly seen in the circled steps, when instead of generating again [1,2,3,4] it backtracks and instead generates [1,2,4,3] and if that also gives false it will backtrack even more and try more combinations and so on.



As you can see unfortunately it also fails with [1,2,4,3].

Informal comment of the solution: What do you think about the solution?
We already explained how it works commenting the trace output, our opinion is that is a very basic solution that uses too many resources, taking into account what we've learned in previous practices we can see how it is doing more or less a simple search similar to depth or breadth.

About the implementation we think that Prolog is a very good programming language to solve this kind of problem as with very few lines of code you can program a statement that represents all possible permutations of a list, and the way **perm**/2 was implemented impressed us a lot.

Have you come up with another way of solving it?
We've thought about doing a heuristic to help expand nodes, so that we know which permutations to expand first, as we've seen in theory and other practices choosing the right one can be very difficult but we've been searching around the internet and have found a very interesting paper about how to use Machine Learning to solve this problem:

Kaur, Bhupinder & Rawat, Shashank & Dinesh, Remalli & Ghosh, Sudipta & Puri, Manish & Das, Anamika & Singh, Jitendra & Sengar,. (2013). A Novel Approach to 8-Queen Problem Employing Machine Learning. 10.13140/2.1.5042.3364. .

We haven't found if there is another way to solve this problem that doesn't involve backtracking or any form of it.