Prolog

-Cut and Fail operator

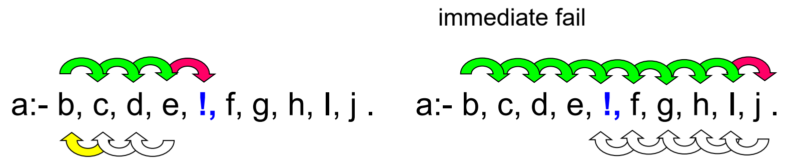
-Problem representation in Prolog

Prolog's Persistence

- In Prolog, a programmer can control execution of a program through the ordering of clauses and goals.
- Prolog will automatically backtrack in order to satisfy a goal.
- Automatic backtracking is a useful programming concept as it relieves the programmer of burden of programming backtracking explicitly.
- However, uncontrolled backtracking may cause inefficiency in a program.
- It becomes important to control, or prevent backtracking.
- We can do this in Prolog using ***cut*** operator ------symbol (!).

Cut !

- If we want to restrict backtracking we can control which sub-goals can be redone using the cut = **!**.
- We use it as a goal within the body of clause.
- It succeeds when `called`, but stops when an attempt is made to `redo` it on backtracking.
- It commits to the choices made so far in the predicate.
  - **unlimited backtracking can occur before and after the cut but no backtracking can go through it.**


immediate fail
a:- b, c, d, e, !, f, g, h, I, j .
a:- b, c, d, e, !, f, g, h, I, j .

Mutually Exclusive Clauses

- We should only use a cut if the clauses are mutually exclusive (if one succeeds the others won't).
- **If the clauses are mutually exclusive then we don't want Prolog to try the other clauses when the first fails**
  = redundant processing.
- By including a cut in the body of a clause we are committing to that clause.
  - Placing a cut at the start of the body commits to the clause as soon as head unification succeeds.
    ```
    a(1,X):- !, b(X), c(X).
    ```
  - Placing a cut somewhere within the body (even at the end) states that we cannot commit to the clause until certain sub-goals have been satisfied.
    ```
    a(_,X):- b(X), c(X), !.
    ```

Intro to Cut and fail Operators

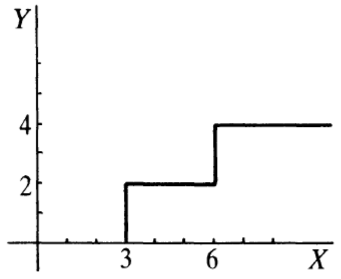


Figure 5.1 A double-step function.

```
f( X, 0)  :-  X < 3.                 % Rule 1
f( X, 2)  :-  3 =< X, X < 6.         % Rule 2
f( X, 4)  :-  6 =< X.                % Rule 3
  ?-  f( 1, Y),    2 < Y.
```

Mutually Exclusive Clauses

```
f(X,0):- X < 3.
f(X,1):- 3 =< X, X < 6.
f(X,4):- 6 =< X.
```

```
|?- trace, f(2,N).
   1      1 Call: f(2,_487) ?
   2      2 Call: 2<3 ?
   2      2 Exit: 2<3 ? ?
   1      1 Exit: f(2,0) ?
N = 0 ? ;
   1      1 Redo: f(2,0) ?
   3      2 Call: 3=<2 ?
   3      2 Fail: 3=<2 ?
   4      2 Call: 6=<2 ?
   4      2 Fail: 6=<2 ?
   1      1 Fail: f(2,_487) ?
no
```

Green Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- 3 =< X, X < 6, !.
f(X,4):- 6 =< X.
```

If you reach this point don't bother trying any other clause.

```
|?- trace, f(2,N).
   1      1 Call: f(2,_487) ?
   2      2 Call: 2<3 ?
   2      2 Exit: 2<3 ? ?
   1      1 Exit: f(2,0) ?
N = 0 ? ;
no
```

- Notice that the answer is still the same, with or without the cut.
  - This is because the cut does not alter the logical behaviour of the program.
  - It only alters the procedural behaviour: specifying which goals get checked when.
- This is called a ***green cut***. It is the correct usage of a cut.
- Be careful to ensure that your clauses are actually mutually exclusive when using green cuts!

Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- 3 =< X, X < 6, !.
f(X,4):- 6 =< X.
```

Redundant?

```
| ?- f(7,N).
   1      1 Call: f(7,_475) ?
   2      2 Call: 7<3 ?
   2      2 Fail: 7<3 ?
   3      2 Call: 3=<7 ?
   3      2 Exit: 3=<7 ?
   4      2 Call: 7<6 ?
   4      2 Fail: 7<6 ?
   5      2 Call: 6=<7 ?
   5      2 Exit: 6=<7 ?
   1      1 Exit: f(7,2) ?
N = 2 ?
yes
```

- Because the clauses are mutually exclusive and ordered we know that once the clause above fails certain conditions must hold.
- We might want to make our code more efficient by removing superfluous tests.

Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- X < 6, !.
f(X,4).
```

```
f(X,0):- X < 3.
f(X,1):- X < 6.
f(X,4).
```

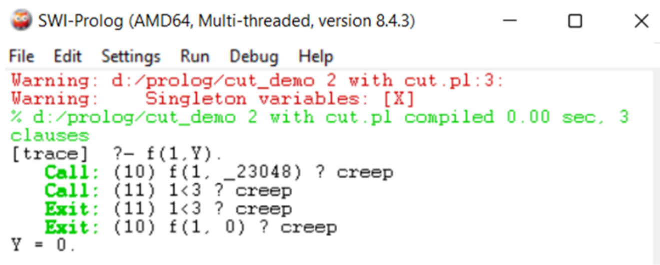
```
| ?- f(7,N).
   1      1 Call: f(7,_475) ?
   2      2 Call: 7<3 ?
   2      2 Fail: 7<3 ?
   3      2 Call: 7<6 ?
   3      2 Fail: 7<6 ?
   1      1 Exit: f(7,2) ?
N = 2 ?
yes
```

```
| ?- f(1,Y).
   1      1 Call: f(1,_475) ?
   2      2 Call: 1<3 ?
   2      2 Exit: 1<3 ? ?
   1      1 Exit: f(1,0) ?
   Y = 0 ? ;
   1      1 Redo: f(1,0) ?
   3      2 Call: 1<6 ?
   3      2 Exit: 1<6 ? ?
   1      1 Exit: f(1,1) ?
   Y = 1 ? ;
   1      1 Redo: f(1,1) ?
   1      1 Exit: f(1,2) ?
   Y = 2 ?
   yes
```

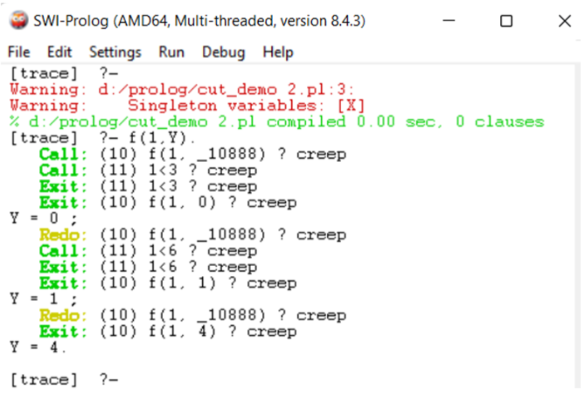
Cuts !

```
f(X,0):- X < 3, !.
f(X,1):- X < 6, !.
f(X,4).
```

```
f(X,0):- X < 3.
f(X,1):- X < 6.
f(X,4).
```


SWI-Prolog (AMD64, Multi-threaded, version 8.4.3)
File Edit Settings Run Debug Help

```
Warning: d:/prolog/cut_demo 2 with cut.pl:3:
Warning:    Singleton variables: [X]
% d:/prolog/cut_demo 2 with cut.pl compiled 0.00 sec, 3
clauses
[trace]  ?- f(1,Y).
   Call: (10) f(1, _23048) ? creep
   Call: (11) 1<3 ? creep
   Exit: (11) 1<3 ? creep
   Exit: (10) f(1, 0) ? creep
Y = 0.
```


SWI-Prolog (AMD64, Multi-threaded, version 8.4.3)
File Edit Settings Run Debug Help

```
[trace]  ?-
Warning: d:/prolog/cut_demo 2.pl:3:
Warning:    Singleton variables: [X]
% d:/prolog/cut_demo 2.pl compiled 0.00 sec, 0 clauses
[trace]  ?- f(1,Y).
   Call: (10) f(1, _10888) ? creep
   Call: (11) 1<3 ? creep
   Exit: (11) 1<3 ? creep
   Exit: (10) f(1, 0) ? creep
Y = 0 ;
   Redo: (10) f(1, _10888) ? creep
   Call: (11) 1<6 ? creep
   Exit: (11) 1<6 ? creep
   Exit: (10) f(1, 1) ? creep
Y = 1 ;
   Redo: (10) f(1, _10888) ? creep
   Exit: (10) f(1, 4) ? creep
Y = 4.

[trace]  ?-
```

Compact version

- The most compact version of this same program would be:

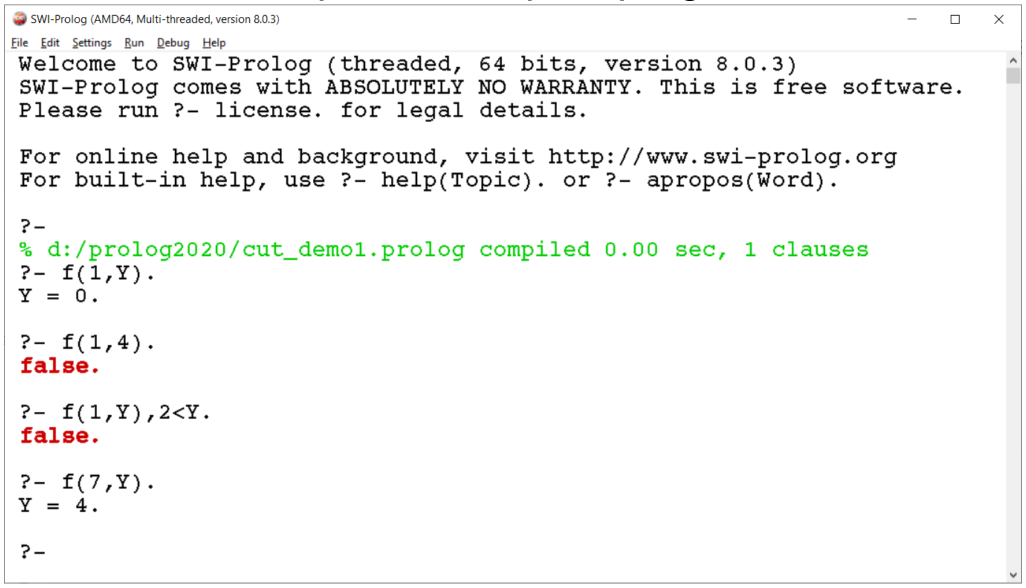
```
f(X,Y):-X<3,!,Y is 0;X<6,!,Y is 2; Y is 4.
```

- Check how all of the queries will be answered by this program:

```
?-f(1, Y), 2 <Y.
?-f(7,Y).
?-f(1,4)
```

- You can now notice that once a case is qualified, other options are not being tried worthlessly due to automatic backtracking.

Output of Compact program



```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% d:/prolog2020/cut_demo1.prolog compiled 0.00 sec, 1 clauses
?- f(1,Y).
Y = 0.

?- f(1,4).
false.

?- f(1,Y),2<Y.
false.

?- f(7,Y).
Y = 4.

?-
```

```
f(X,Y):-X<3,!,Y is 0;X<6,!,Y is 2; Y is 4.
```

Using the cut

- *Red cuts* change the logical behaviour of a predicate.
- TRY NOT TO USE RED CUTS!
- Red cuts make your code hard to read and are dependent on the specific ordering of clauses (which may change once you start writing to the database).
- If you want to improve the efficiency of a program use *green cuts* to control backtracking.
- Do not use cuts in place of tests.

To ensure a logic friendly cut either:

```
p(X):- test1(X), !, call1(X).
p(X):- test2(X), !, call2(X).
p(X):- testN(X), !, callN(X).
```

`testI` predicates are mutually exclusive.

```
p(1,X):- !, call1(X).
p(2,X):- !, call2(X).
p(3,X):- !, callN(X).
```

The mutually exclusive tests are in the head of the clause.

Cut

- Reduces the search space by dynamic pruning of the search tree
- Can be used to prevent the Prolog system from searching parts of the search space that are known to contain no solutions.
- Green cuts do not change the meaning of the program, red cuts do (to be avoided!)

Red Cuts

- Red cuts: explicitly omitting conditions = changing the meaning of a program
- Thus, problematic; also most common source of bugs in Prolog programs

Fail

- This predicate always fails.
- *Cut* and *Fail* combination is used to produce negation.
- Since the LHS of the neck cannot contain any operator, *A* → *~B* is implemented as

neg(A) :- A, !, fail ; true.

% here B is same as ~A

means If A is true then fail otherwise true.

Negation demo

```
/* Method 1: using negation operator \+           */
?- \+(2=4).
true.

likes(mary, X):- \+ snake(X), animal(X).
animal(fido).
snake(kobra).

/*Method 2:/* defining negated clause separately   */
animal(fido).
snake(kobra).
not(snake(X)) :- snake(X), ! , fail ; true.
likes(mary,X) :- not(snake(X)), animal(X).


  4 ?- likes(mary,kobra).
  false.

  5 ?- likes(mary,fido).
  true.
```

Negation as failure

- Consider:
  p :- q, !, r.
  p:- s.

  means roughly:
  p :- q, r.
  p :- not_q, s.

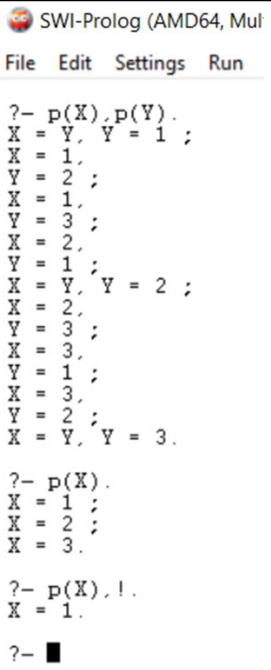
  Define **not_q**:
  not_q :- q, !, fail;true.

- Not very practical. Better:

  not(G) :- G, !, fail.
  not(G).
- **not/1 is** meta-predicate.

Problems on CUT

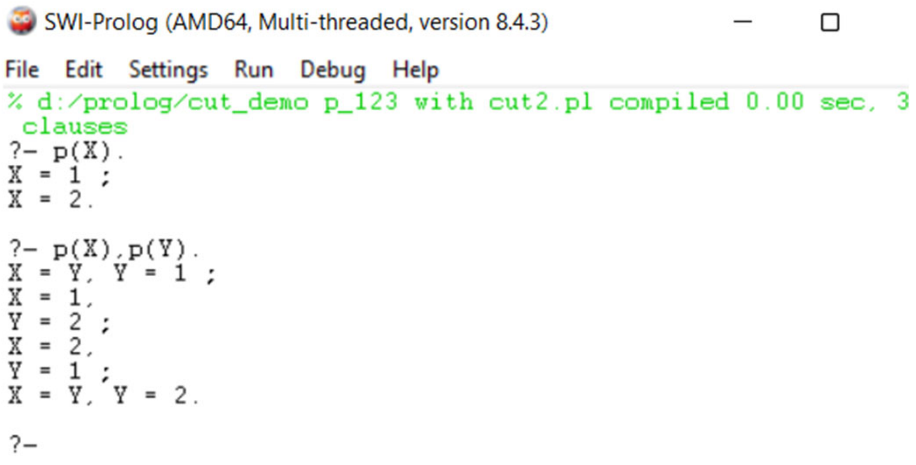
```
p(1).
p(2).
p(3).

?-p(X).
X=1, X=2, X=3

?-p(X), !.
X=1

?-p(X),p(Y).
```



```
?- p(X),p(Y).
X = Y, Y = 1 ;
X = 1,
Y = 2 ;
X = 1,
Y = 3 ;
X = 2,
Y = 1 ;
X = Y, Y = 2 ;
X = 2,
Y = 3 ;
X = 3,
Y = 1 ;
X = 3,
Y = 2 ;
X = Y, Y = 3.

?- p(X).
X = 1 ;
X = 2 ;
X = 3.

?- p(X),!.
X = 1.

?-
```

Problems on CUT

```
p(1).
p(2):-!.
p(3).

?-p(X).
X=1, X=2

?-p(X), !.
X=1

?-p(X),p(Y).
```


SWI-Prolog (AMD64, Multi-threaded, version 8.4.3)
File Edit Settings Run Debug Help

```
% d:/prolog/cut_demo p_123 with cut2.pl compiled 0.00 sec, 3
 clauses
?- p(X).
X = 1 ;
X = 2.

?- p(X),p(Y).
X = Y, Y = 1 ;
X = 1,
Y = 2 ;
X = 2,
Y = 1 ;
X = Y, Y = 2.

?-
```

Vroylqj#DL#Suredhp v#xvlqj#Surorj=

1. Water-jug
2. Monkey-Banana
3. N-queens puzzle
4. Tic-tac-toe puzzle
5. 8-puzzle
6. ROBOT’S world
7. Language parser
8. Cryptarithmetics
9. Sudoku

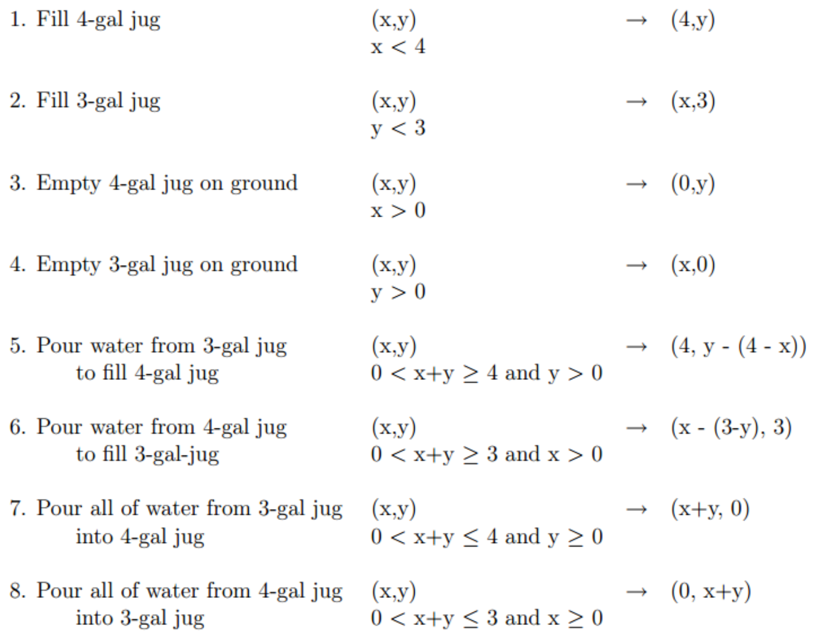
A Water Jug Problem:

- You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to ll the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug.

State Representation and Initial State:

- represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note $0 \leq x \leq 4$, and $0 \leq y \leq 3$.
- Our initial state: (0,0)
- Goal Predicate { state = (2,y) where $0 \leq y \leq 3$.

Operators



1. Fill 4-gal jug	(x,y) $x < 4$	$\rightarrow$	(4,y)
2. Fill 3-gal jug	(x,y) $y < 3$	$\rightarrow$	(x,3)
3. Empty 4-gal jug on ground	(x,y) $x > 0$	$\rightarrow$	(0,y)
4. Empty 3-gal jug on ground	(x,y) $y > 0$	$\rightarrow$	(x,0)
5. Pour water from 3-gal jug to fill 4-gal jug	(x,y) $0 < x+y \geq 4$ and $y > 0$	$\rightarrow$	(4, y - (4 - x))
6. Pour water from 4-gal jug to fill 3-gal-jug	(x,y) $0 < x+y \geq 3$ and $x > 0$	$\rightarrow$	(x - (3-y), 3)
7. Pour all of water from 3-gal jug into 4-gal jug	(x,y) $0 < x+y \leq 4$ and $y \geq 0$	$\rightarrow$	(x+y, 0)
8. Pour all of water from 4-gal jug into 3-gal jug	(x,y) $0 < x+y \leq 3$ and $x \geq 0$	$\rightarrow$	(0, x+y)

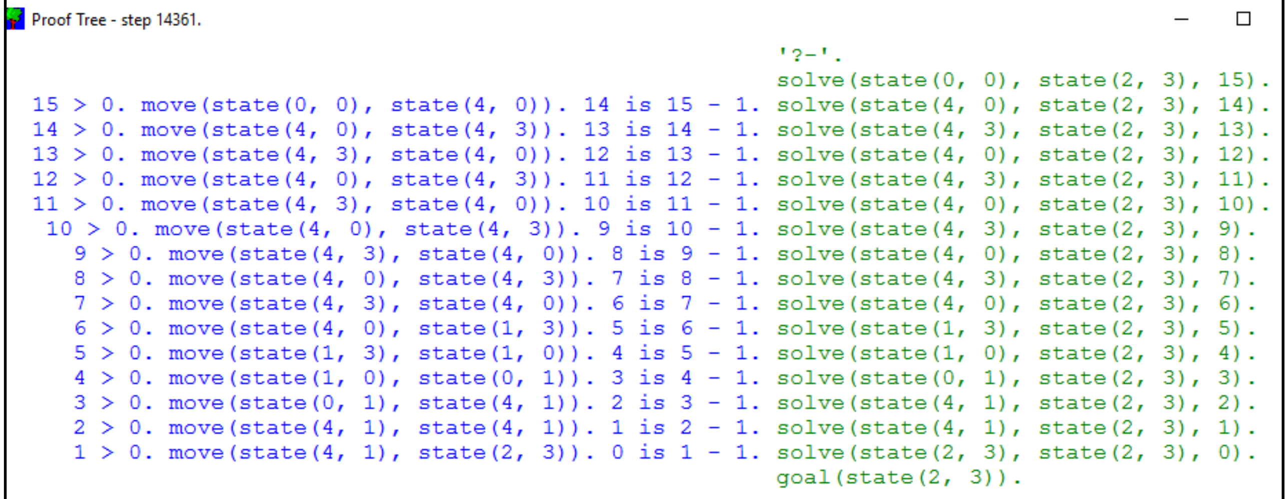
Water-jug problem represented in PROLOG:

```
goal(state(2,_)).
move(state(X,Y),state(4,Y)):- X < 4.
move(state(X,Y),state(X,0)):- Y>0.
move(state(X,Y),state(X,3)):- Y < 3.
move(state(X,Y),state(0,Y)):- X > 0.

move(state(X,Y),state(4,Z)):- T is X+Y, T >= 4,
                             Y > 0,D is 4-X, Z is Y-D.
move(state(X,Y),state(Z,3)):- T is X+Y, T >= 3, X>0,
                              D is 3-Y, Z is X-D.
move(state(X,Y),state(Z,0)):- T is X+Y, T =< 4,
                               Y > 0, Z is X+Y.
move(state(X,Y),state(0,Z)):- T is X+Y, T =< 3,
                               X>0, Z is X+Y.
move(state(X,Y),state(2,0)):- X is 0, Y is 2.
%dfs algo in prolog
solve(N,N):- goal(N).
solve(N,T):- move(N,N1), solve(N1,T).
?-solve(state(0,0),state(2,Y)).
```

Water-jug with depth limited search

```
goal(state(2,_)).
move(state(X,Y),state(4,Y)):- X < 4.
move(state(X,Y),state(X,0)):- Y>0.
move(state(X,Y),state(X,3)):- Y < 3.
move(state(X,Y),state(0,Y)):- X > 0.
move(state(X,Y),state(4,Z)):- T is X+Y, T >= 4, Y > 0,D is 4-X, Z is Y-D.
move(state(X,Y),state(Z,3)):- T is X+Y, T >= 3, X>0, D is 3-Y, Z is X-D.
move(state(X,Y),state(Z,0)):- T is X+Y, T =< 4,   Y > 0, Z is X+Y.
move(state(X,Y),state(0,Z)):- T is X+Y, T =< 3,  X>0, Z is X+Y.

%depth limited algo in prolog

solve(N,N,_):- goal(N).

solve(N,T,MaxDepth):- MaxDepth>0,move(N,N1),
                                      M1 is MaxDepth-1,solve(N1,T,M1).

?-solve(state(0,0),state(2,Y),15).
```

Proof tree



```
                                                     '?-'.
                                                     solve(state(0, 0), state(2, 3), 15).
 15 > 0. move(state(0, 0), state(4, 0)). 14 is 15 - 1. solve(state(4, 0), state(2, 3), 14).
 14 > 0. move(state(4, 0), state(4, 3)). 13 is 14 - 1. solve(state(4, 3), state(2, 3), 13).
 13 > 0. move(state(4, 3), state(4, 0)). 12 is 13 - 1. solve(state(4, 0), state(2, 3), 12).
 12 > 0. move(state(4, 0), state(4, 3)). 11 is 12 - 1. solve(state(4, 3), state(2, 3), 11).
 11 > 0. move(state(4, 3), state(4, 0)). 10 is 11 - 1. solve(state(4, 0), state(2, 3), 10).
  10 > 0. move(state(4, 0), state(4, 3)). 9 is 10 - 1. solve(state(4, 3), state(2, 3), 9).
    9 > 0. move(state(4, 3), state(4, 0)). 8 is 9 - 1. solve(state(4, 0), state(2, 3), 8).
    8 > 0. move(state(4, 0), state(4, 3)). 7 is 8 - 1. solve(state(4, 3), state(2, 3), 7).
    7 > 0. move(state(4, 3), state(4, 0)). 6 is 7 - 1. solve(state(4, 0), state(2, 3), 6).
    6 > 0. move(state(4, 0), state(1, 3)). 5 is 6 - 1. solve(state(1, 3), state(2, 3), 5).
    5 > 0. move(state(1, 3), state(1, 0)). 4 is 5 - 1. solve(state(1, 0), state(2, 3), 4).
    4 > 0. move(state(1, 0), state(0, 1)). 3 is 4 - 1. solve(state(0, 1), state(2, 3), 3).
    3 > 0. move(state(0, 1), state(4, 1)). 2 is 3 - 1. solve(state(4, 1), state(2, 3), 2).
    2 > 0. move(state(4, 1), state(4, 1)). 1 is 2 - 1. solve(state(4, 1), state(2, 3), 1).
    1 > 0. move(state(4, 1), state(2, 3)). 0 is 1 - 1. solve(state(2, 3), state(2, 3), 0).
                                                     goal(state(2, 3)).
```

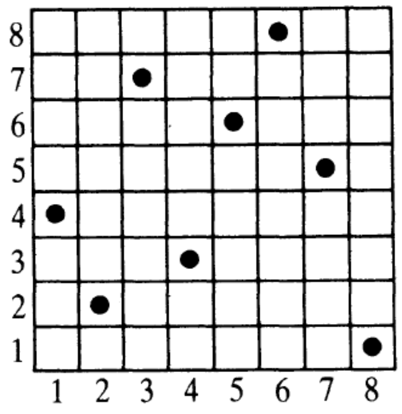
The 8 Queens Problem

- The problem here is to place eight queens on the empty chessboard in such a way that no queen attacks any other queen.
- The solution will be programmed as a unary predicate:
  - solution(**Pos**)
  - which is true if and only if **Pos** represents a position with eight queens that do not attack each other.
- Now choose a representation for board positions.

Problem Representation

- One natural choice is to represent the position by a list of eight items, each of them corresponding to one queen.
- Each square can be specified by a pair of coordinates (X and Y) on the board, where each coordinate is an integer between 1 and 8.
- In the program we can write such a pair as X/Y.
- Having chosen this representation, the problem is to find such a list of the form [X1/Y1, X2/Y2, X3/Y3, ..., X8/Y8] which satisfies the no-attack requirement.

Problem Representation



- As all the queens will have to be in different columns to prevent vertical attacks,
  - we can immediately constrain the choice and so make the search task easier.
- More specific template: [1/Y1,2/Y2,3/Y3,. . . ,8 /Y8]
- Example is shown in figure.

```
[1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1]
```

The solution(List) relation

It can be formulated by considering 2 cases:

Here List= [1/Y1, 2/Y2, 3/Y3,. . . , 8/Y8]

- Case 1: The list of queens is empty i.e. **solution([ ]).**
- Case 2: The list of queens is non-empty, then it looks like: [X/Y I Others], now following conditions must hold:

```
solution( [X/Y | Others] ) :-
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).
```

-There must be no attack between the queens in the list others; i.e., Others itself must also be a solution.

-X and Y must be integers between 1 and 8.

-A queen at square X/Y must not attack any of the queens in the list Others.

noattack(Q, Qlist)

- To specify that a queen at some square does not attack another square is easy:
  - the two squares must not be in the same row, the same column or the same diagonal.
- nottack(Q, Qlist) can have 2 sub cases:
  - If the list Qlist is empty then relation is certainly true because there is no queen to be attacked.

  i.e. **noattack (_, [ ]).**

  - If Qlist is not empty then it has the form [Q1 I Qlist1] and two conditions must be satisfied:

  (a) the queen at Q must not attack the queen at Q1, and

  (b) the queen at Q must not attack any of the queens in Qlist1.

Program for 8-queens problem

```
solution( [] ).

solution( [X/Y | Others] )  :-          % First queen at X/Y, other queens at Others
  solution( Others),
  member( Y, [1,2,3,4,5,6,7,8] ),
  noattack( X/Y, Others).               % First queen does not attack others

noattack( _, [] ).                      % Nothing to attack

noattack( X/Y, [X1/Y1 | Others] )  :-
  Y =\= Y1,                             % Different Y-coordinates
  Y1-Y =\= X1-X,                        % Different diagonals
  Y1-Y =\= X-X1,
  noattack( X/Y, Others).

member( X, [X | L] ).

member( X, [Y | L] )  :-
  member( X, L).

% A solution template

template( [1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8] ).
```

```
?-  template( S), solution( S).
```

4-Queens Demo

```
?- solution ([1/Y1, 2/Y2, 3/Y3, 4/Y4]).
```

```
solution( [ ]).
solution( [X/Y|Others] ) :-
solution( Others),
member(Y , [1,2,3,4]),
noattack( X/Y, Others).

 noattack( _,[  ]).

noattack( X/Y, [X1/Y1|Others]) :-
Y =\= Y1, Y1-Y =\= X1-X , Y1-Y =\=
X-X1, noattack(X/Y,Others).

member( X,[X|L]).
member( X, [Y|L]):-member( X, L).

%?-
solution([1/Y1,2/Y2,3/Y3,4/Y4]).
```

4-Queens Demo

?- solution([1/Y1, 2/Y2, 3/Y3, 4/Y4]).

/

solution([2/Y2, 3/Y3, 4/Y4]), noattack(1/Y1, [2/Y2, 3/Y3, 4/Y4])

```
solution( [ ]).
solution( [X/Y | Others ] ) :-
solution( Others),
member(Y , [1,2,3,4] ),
noattack ( X/Y, Others).

 noattack( _ , [  ] ).

noattack( X/Y, [X1/Y1|Others]) :-
Y =\= Y1, Y1-Y =\= X1-X , Y1-Y
=\= X-X1, noattack (X/Y, Others).

member( X,[X|L]).
member( X, [Y|L]):-member( X, L).

%?-
solution([1/Y1,2/Y2,3/Y3,4/Y4]).
```

4-Queens Demo

?- solution([1/Y1, 2/Y2, 3/Y3, 4/Y4]).

/

solution([2/Y2, 3/Y3, 4/Y4]), noattack(1/Y1, [2/Y2, 3/Y3, 4/Y4])

/

solution([3/Y3, 4/Y4]), noattack(2/Y2, [3/Y3, 4/Y4]).

```
solution( [ ]).
solution( [X/Y|Others] ) :-
solution( Others),
member(Y , [1,2,3,4]),
noattack( X/Y, Others).

 noattack( _,[  ]).

noattack( X/Y, [X1/Y1|Others]) :-
Y =\= Y1, Y1-Y =\= X1-X , Y1-Y
=\= X-X1, noattack(X/Y,Others).

member( X,[X|L]).
member( X, [Y|L]):-member( X, L).

%?-
solution([1/Y1,2/Y2,3/Y3,4/Y4]).
```

4-Queens Demo

?- solution([1/Y1, 2/Y2, 3/Y3, 4/Y4]).

/

solution([2/Y2, 3/Y3, 4/Y4]), noattack(1/Y1, [2/Y2, 3/Y3, 4/Y4])

/

solution([3/Y3, 4/Y4]), noattack(2/Y2, [3/Y3, 4/Y4]).

/

solution([4/Y4]), noattack(3/Y3, [4/Y4]).

```
solution( [ ]).
solution( [X/Y|Others] ) :-
solution( Others),
member(Y , [1,2,3,4]),
noattack( X/Y, Others).

 noattack( _,[  ]).

noattack( X/Y, [X1/Y1|Others]) :-
Y =\= Y1, Y1-Y =\= X1-X , Y1-Y
=\= X-X1, noattack(X/Y,Others).

member( X,[X|L]).
member( X, [Y|L]):-member( X, L).

%?-
solution([1/Y1,2/Y2,3/Y3,4/Y4]).
```

4-Queens Demo

?- solution([1/Y1, 2/Y2, 3/Y3, 4/Y4]).

/

solution([2/Y2, 3/Y3, 4/Y4]), noattack(1/Y1, [2/Y2, 3/Y3, 4/Y4])

/

solution([3/Y3, 4/Y4]), noattack(2/Y2, [3/Y3, 4/Y4]).

/

solution([4/Y4]), noattack(3/Y3, [4/Y4]).

/

solution([]), noattack(4/Y4, []).

/

true

```
solution( [ ]).
solution( [X/Y|Others] ) :-
solution( Others),
member(Y , [1,2,3,4]),
noattack( X/Y, Others).

 noattack( _,[  ]).

noattack( X/Y, [X1/Y1|Others]) :-
Y =\= Y1, Y1-Y =\= X1-X , Y1-Y
=\= X-X1, noattack(X/Y,Others).

member( X,[X|L]).
member( X, [Y|L]):-member( X, L).

%?-
solution([1/Y1,2/Y2,3/Y3,4/Y4]).
```

?- solution([1/Y1, 2/Y2, 3/Y3, 4/Y4]).

/

solution([2/Y2, 3/Y3, 4/Y4]), noattack(1/Y1, [2/Y2, 3/Y3, 4/Y4])

/

solution([3/Y3, 4/Y4]), noattack(2/Y2, [3/Y3, 4/Y4]).

/

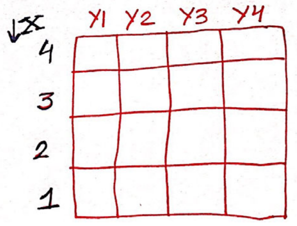
solution([4/Y4]), noattack(3/Y3, [4/Y4]).

/

solution([]), noattack(4/Y4, []).

/

true



```
solution( [ ]).
solution( [X/Y|Others] ) :-
solution( Others),
member(Y , [1,2,3,4]),
noattack( X/Y, Others).

 noattack( _,[  ]).

noattack( X/Y, [X1/Y1|Others]) :-
Y =\= Y1, Y1-Y =\= X1-X , Y1-Y
=\= X-X1, noattack(X/Y,Others).

member( X,[X|L]).
member( X, [Y|L]):-member( X, L).

%?-
solution([1/Y1,2/Y2,3/Y3,4/Y4]).
```

4-Queens Demo

?- solution([1/Y1, 2/Y2, 3/Y3, 4/Y4]).

/

solution([2/Y2, 3/Y3, 4/Y4]), noattack(1/Y1, [2/Y2, 3/Y3, 4/Y4])

/

solution([3/Y3, 4/Y4]), noattack(2/Y2, [3/Y3, 4/Y4]).

/

solution([4/Y4]), noattack(3/Y3, [4/Y4]).

/

solution([]), noattack(4/Y4, []).

/

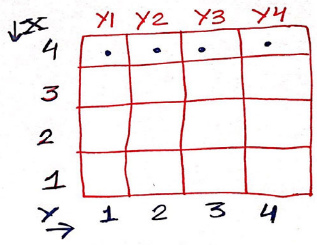
true

for X=4, Y4=1 or 2 or 3 or 4

fail
For Y4=1
Y3 can be only 3 or 4

succeed
For Y4=2, Y3=4 only
Y4=3, Y3=1 only

fail → Y4=4, Y3=1 or 2



```
solution( [ ]).
solution( [X/Y | Others ] ) :-
solution( Others),
member(Y , [1,2,3,4]),
noattack ( X/Y, Others).

 noattack( _,[  ]).

noattack( X/Y, [X1/Y1|Others]) :-
Y =\= Y1, Y1-Y =\= X1-X , Y1-Y
=\= X-X1, noattack(X/Y,Others).

member( X,[X|L]).
member( X, [Y|L]):-member( X, L).

%?-
solution([1/Y1,2/Y2,3/Y3,4/Y4]).
```

4-Queens Demo

4-Queens Demo

```
% d:/prolog2020/4Queen_demo.pl compiled 0.00 sec,
6 clauses
?- solution([1/Y1,2/Y2,3/Y3,4/Y4]).
Y1 = 3,
Y2 = 1,
Y3 = 4,
Y4 = 2 ;
Y1 = 2,
Y2 = 4,
Y3 = 1,
Y4 = 3 ;
false.

?-
```

Just 2 unique answers!

```
solution( [ ]).
solution( [X/Y|Others] ) :-
solution( Others),
member(Y , [1,2,3,4]),
noattack( X/Y, Others).

 noattack( _,[  ]).

noattack( X/Y, [X1/Y1|Others]) :-
Y =\= Y1, Y1-Y =\= X1-X , Y1-Y =\=
X-X1, noattack(X/Y,Others).

member( X,[X|L]).
member( X, [Y|L]):-member( X, L).

%?-solution([1/Y1,2/Y2,3/Y3,4/Y4]).
```

Solutions
(92 unique answers)

S = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1];

S = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1];

S = [1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1];

…

Summary

- Base and recursive cases
- Using focused recursion to stop infinite loops.
- List processing through recursion: member/2
- Introduced the Prolog tracer.
- Showed three techniques for collecting results:
  - Recursively find a result, then revise it at each level.
    - listlength/3
  - Use an accumulator to build up result during recursion.
    - reverse/3
  - Build result in the head of the clause during backtracking.
    - append/3