part1.txt

```
% Queens example
n_queens(N, Qs) :-
        length(Qs, N),
        Qs ins 1..N,                          % restricts all values in Qs to values from 1 to N (prevents Queens from being not on the board)
        safe_queens(Qs).

safe_queens([]).
safe_queens([Q|Qs]) :- safe_queens(Qs, Q, 1), safe_queens(Qs).

safe_queens([], _, _).
safe_queens([Q|Qs], Q0, D0) :-
        Q0 #\= Q,                                     % Q0 is not equal to Q
        abs(Q0 - Q) #\= D0,                   % absolute value of Q0 minus Q cannot be equal to the current diagonal
        D1 #= D0 + 1,                         % increment the diagonal
        safe_queens(Qs, Q0, D1).        % recursively call on Qs, Q0, and D1 (incremented diagonal)

Safe queens gets called as the last statement of n_queens.
This call starts by going to the safe queens which takes a
list argument and splts the list into its head and tail as
Q and Qs respectively (we are going to label the arguments
in this list [A, B, C, D]). Then safe_queens is called and the
form with 3 arguments, a list ([B, C, D]), a variable (A), and a 1
is called.

From here Qs is broken down further into its head (B) and tail ([C, D]),
A becomes labelled as Q0 and B becomes labelled as Q. the following
constraints are then applied

1. A is not equal to B
2. The absolute value of A - B is not equal to 1

D1 is then set as D0 + 1 (effectively incrementing D0), and safe_queens
with three arguments is recursively called with arguments [C, D], A, and D1 (1)

The following restraints are created by continuing to trace this recursive call

3. A is not equal to C
4. The absolute value of A - C is not equal to 2

5. A is not equal to D
6. The absolute value of A - D is not equal to 3

At this point we have finished the first safe_queens call in the safe_queens with
a single argument. This statement then recursively calls itself on the tail of the
list it received causing safe queens to get called on the tail of the prior list.
This repeats until there are no more items in the tail of the list passed to safe
queens and we hit our base case. These are the resulting constraints of tracing
this algorithm.

7. B is not equal to C
8. The absolute value of B - C is not equal to 1

9. B is not equal to D
10. The absolute value of B - D is not equal to 2

11. C is not equal to D
12. The absolute \vbalue of C - D is not equal to 1
```

```
% Sodoku puzzle solver
sudoku(Rows) :-
        length(Rows, 9), maplist(same_length(Rows), Rows), % bind the rows to have length of 9
                                                           % bind each element in same_length(Rows) to
                                                           % another element in Rows creating a 9x9
                                                           % grid for the sodoku puzzle

        append(Rows, Vs), Vs ins 1..9,      % append Rows and Vs, limit the domain of Vs
                                            % to 1 through 9

        maplist(all_distinct, Rows),        % check that all Rows are distinct

        transpose(Rows, Columns),           % makes list of columns based on list of rows given

        maplist(all_distinct, Columns),     % check that all Columns are distinct

        Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],% define Rows as A - I

        blocks(As, Bs, Cs),                 % check that values in the 3 blocks (3x3 squares)
                                            % found in rows A - C are unique

        blocks(Ds, Es, Fs),                 % check that values in the 3 blocks found in rows
                                            % D - F are unique

        blocks(Gs, Hs, Is).                 % check that values in the 3 blocks found in rows
                                            % G - I are unique

blocks([], [], []).
blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
        all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]), % check that values in a block are unique

        blocks(Ns1, Ns2, Ns3).  % recursively call block on tail (row with 3 elements removed)

problem(1, [[_,_,_,_,_,_,_,_,_],        % definition of problem to be solved
            [_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).

% The first line of soduku binds the rows to have a length of 9 to give us the row size
% of our sodoku grid. An element the same length of the rows is then bound to rows
% providing the 9x9 grid for the sodoku puzzle to be held and solved in. Rows and Vs
% (our values for each spot in the grid) are then appended and Vs is given a constained
% domain of values 1-9 (the valid values that can exist in the grid of a sodoku puzzle).
% all_distinct is then mapped to Rows to make sure that all the Rows are distinct.
% transpose then creates a list of columns based off of a list of rows given and these
% columns are checked to make sure that they are all distainct. We then define rows A - I
% in a list of rows. At this point we have created a sodoku puzzle with distinct rows
% and columns and the last thing to check is the 3x3 blocks existing around the edges
% and center of the puzzle. This is checked by putting rows A-C, D-F, and G-I into the blocks
% statement and check that the first, middle, and last 3 values of each are all distinct
% from the first, middle, and last 3 values of the others (all these values must be
% distinct as they are what constitutes a block). If this check passes then we are left with
% a valid sodoku puzzle.
```

```
% Knights and Knaves

/* You meet two people
Person A says at least one of the following is true:
      that I am a knight or that Person B is a knight
Person B says Person A could say that I am a knave */
:- use_module(library(clpb)).

knights(1, [A,B]) :-
        sat(A =:= ((A =:= 1) + (B =:= 1))),
        sat(B =:= (A =:= ~B)).

% knights(1, [A,B]).
% A = B, B = 0

/* You meet three people
Person A says that Person B is a knave
Person B says that it is false that Person C is a knave
Person C says I am a knight or Person A is a knight */

knights(2, [A,B,C]) :-
        sat(A =:= ~B),
        sat(B =:= C),
        sat(C =:= ((C =:= 1) + (A =:= 1))).

% knights(2, [A,B,C]).
% A = 0,
% B = C, C = 1

/* You meet 4 people
Person A tells you that Person C could say that Person B is a knave
Person B says that Person C is a knave
Person C says only a knave would say that Person D is a knave
Person D says I and Person A are different*/

knights(3, [A,B,C,D]) :-
        sat(A =:= (C =:= ~B)),
        sat(B =:= ~C),
        sat(C =:= (1 =:= ~D)),
        sat(D =:= ~A).

% knights(3, [A,B,C,D]).
% A = C, C = 1,
% B = D, D = 0
```

```
:- use_module(library(clpfd)).

crypt1([H1|L1], [H2|L2], [H3|L3], L4) :-
    L4 ins 0..9,                    % Give constraints on variable values in L4
    H1 #\= 0, H2 #\= 0, H3 #\= 0,   % Heads cannot have value 0
    all_distinct(L4),               % Variable values are distinct in L4
    reverseWord([H1|L1], R1, []),   % Reverse the 3 input words
    reverseWord([H2|L2], R2, []),
    reverseWord([H3|L3], R3, []),
    helper(R1, R2, R3, 0).          % Call to helper function that does the sum with reversed words one iteration at a time


reverseWord([], X, X).          % base case: returns reversed list when entered list is empty
reverseWord([H|T], X, A) :-     % recursive case: takes head and adds it to accumulator list
    reverseWord(T, X, [H|A]).

% first word, second word, third word, num iterations, carry1, carry2
helper([],[],[C], C).
helper([], [H2|T2], [H3|T3], C) :-
    H3 #= (H2 + C) mod 10,
    H2 + C #= H3 + (C1*10),
    helper([], T2, T3, C1).

helper([H1|T1], [], [H3|T3], C) :-
    H3 #= (H2 + C) mod 10,
    H2 + C #= H3 + (C1*10),
    helper(T2, [], T3, C1).

helper([H1|T1], [H2|T2], [H3|T3], C) :-
    H3 #= (H1 + H2 + C) mod 10,
    H1 + H2 + C #= H3 + (C1*10),
    helper(T1, T2, T3, C1).

/*
crypt1([S,E,N,D],[M,O,R,E],[M,O,N,E,Y],[D,N,E,S,R,O,M,Y]), labeling([ff],[D,N,E,S,R,O,M,Y]).
D = 7,
E = 5,
M = 1,
N = 6,
O = 0,
R = 8,
S = 9,
Y = 2
false <- this false shows that there are no other solutions

crypt1([T,H,I,S],[D,O],[W,O,R,K],[T,H,I,S,D,O,E,W,R,K]), labeling([ff],[T,H,I,S,D,O,E,W,R,K]).
D = 2,
E = 8,
H = 0,
I = 4,
K = 5,
O = 9,
R = 7,
S = 6,
T = 1,
W = 3

D = 2,
E = 8,
H = 0,
I = 4,
K = 5,
O = 6,
R = 7,
S = 9,
T = 1,
W = 3

This one had a lot of solutions so here are the first two
*/
```