PART 1

1.

/*

When we run N=4 in the n-Queens problem, the safe_queens function generates a total of 12 constraints. The safe_queens method accepts three arguments, and we can see two types of constraints in the order listed below:

1. Q0 #= Q: This constraint ensures that the queen Q1 (or in position 1) does not share the same column as the queen Q2 (the column or row index is represented by Q0).

2. The constraint abs(Q0 - Q) #= D0 prevents queens from being placed on the same diagonal (D0 specifies the difference in row locations between the queens).

The third argument, D0, in safe_queens is responsible for maintaining and implementing diagonal constraints between the queens.

We have four unbound variables for N=4: Qs= [A, B, C, D]. Within the safe_queens function, each variable goes through a series of loops. These are the constraints that were applied in each loop:

1. Constraints are added to the first variable, A, with respect to the remaining variables, B, C, and D, in the first loop. As a result, each pair has two constraints: (A, B), (A, C), and (A, D). In the first loop for variable A, 6 restrictions are therefore applied.

2. The second loop sets constraints on variable B in relation to the remaining variables, C and D. We eliminate A because it was already considered in the first loop. This loop adds two constraints to each pair: (B, C) and (B, D), for a total of four constraints for variable B.

3. Constraints are added to variable C with the only remaining variable, D, in the third loop. This loop imposes two constraints on the pair (C, D).

Following these loops, no additional constraints are required because all variables have been paired and covered. Finally, we have 12 constraints for N=4 queens: 6 constraints added to variable A, 4 constraints added to variable B, and 2 constraints added to variable C. These constraints ensure that no two queens can be positioned in the same row, column, or diagonal.

*/

safe_queens(Qs, Q, 1) where Qs = [Q2..Q8]

Q = Q1 (or in position 1)

D0 = 1

/*

Iteration 1:

Q0 #\= Q, % Q1 <> Q2, domain for Q2 = [2..8], making sure that no 2 queen is on the same side of the row

abs(Q0 – Q) #\= D0), %(Q1 – Q2) <> 1, domain for Q2 = [3..8], making sure no 2 queens are present on same diagonal

D1 #= D0 + 1, % D1 = 1+1 = 2, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([3..8],Q1,2)

Iteration 2:

Q0 #\= Q, % Q1 <> Q3, domain for Q3 = [3..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q1 – Q3) <> 2, making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 2+1 = 3, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([4..8],Q1,3)

Iteration 3:

Q0 #\= Q, % Q1 <> Q4, domain for Q4 = [4..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q1 – Q4) <> 3, making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 3+1 = 4, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([5..8],Q1,4)

Iteration 4:

Q0 #\= Q, % Q1 <> Q5, domain for Q5 = [5..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q1 – Q5) <> 4, making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 4+1 = 5, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([6..8],Q1,5)

Iteration 5:

Q0 #\= Q, % Q1 <> Q6, domain for Q6 = [6..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q1 – Q6) <> 5, making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 5+1 = 6, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([4..8],Q1,6)

Iteration 6:

Q0 #\= Q, % Q1 <> Q7, domain for Q7 = [7..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q1 – Q7) <> 6, making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 6+1 = 7, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([8],Q1,7)

Iteration 7:

Q0 #\= Q, % Q1 <> Q8, domain for Q8 = [8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q1 – Q8) <> 7, making sure no 2 queens are on same diagonal D1 #= D0 + 1, % D1 = 7+1 = 8, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([],Q1,8)

*/

Base Case: safe_queens([] , _ , _).

Loop 2: Call 1:

safe_queens(Qs, Q, 1) Qs = [Q3..Q8]

Q = Q2

D0 = 1

/*

Iteration 1:

Q0 #\= Q, % Q2 <> Q3, domain for Q2 = [3..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q2 – Q3) <> 1, domain for Q3 = [4..8], making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 1+1 = 2, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([4..8],Q2,2)

Iteration 2:

Q0 #\= Q, % Q2 <> Q4, domain for Q4 = [4..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q2 – Q4) <> 2, making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 2+1 = 3, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([5..8],Q2,3)

Iteration 3:

Q0 #\= Q, % Q2 <> Q5, domain for Q5 = [5..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q2 – Q5) <> 3, making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 3+1 = 4, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([6..8],Q2,4)

Iteration 4:

Q0 #\= Q, % Q2 <> Q6, domain for Q6 = [6..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q2 – Q6) <> 4, making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 4+1 = 5, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([7..8],Q2,5)

Iteration 5:

Q0 #\= Q, % Q2 <> Q7, domain for Q7 = [7..8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q2 – Q7) <> 5, making sure no 2 queens are on same diagonal

D1 #= D0 + 1, % D1 = 5+1 = 6, updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([8],Q2,6)

Iteration 6:

Q0 #\= Q, % Q2 <> Q8, domain for Q8 = [8], making sure that no 2 queens are on the same row

abs(Q0 – Q) #\= D0), %(Q2 – Q8) <> 6, making sure no 2 queens are on same diagonal D1 #= D0 + 1, % D1 = 6+1 = 7 , updating D1 for checking diagonal with next Q safe_queens(Qs, Q0, D1). % safe_queens([],Q2,7)

Base case : safe_queens([] , _ , _).

The first argument N-queens problem gives us the dimensions for the number of queens. Label1 returns values for the first query “n_queens(8, Qs), label(Qs)”. The first argument is a positive integer and the second argument is a list of the lengths of the first argument. In this first query, 8 is the 8*8 boards. The first two full loops of safe_queens([Q|Qs], Q0, D0) affect the constraints of column values for the queens Q2...Q7 (assuming 8 queens, and Q0 = 1, Q1 = 5) because they show in a 8*8 board where each queen is

assigned to each of the columns since it is impossible to place two queens in the same column. We need to find which rows the queens are placed in.

When Q0=1, the first queen is placed at the first square of row 0. In Q1 = 5, the second queen is placed at the fifth square of row 1. On running two loops, it shows the different combinations of the queens are placed in the search space from Q2 to Q7. Constraints are determined by how the set of values and variables are chosen so the queens can attack each other depending on their different locations when they are placed in different rows.

*/

2.

```
% render solutions nicely. :- use_rendering(sudoku).

:- use_module(library(clpfd)).

% Example by Markus Triska, taken from the SWI-Prolog manual.

sudoku(Rows) :-                                  %1
   length(Rows, 9),                              %2
   maplist(same_length(Rows), Rows),             %3
   append(Rows, Vs), Vs ins 1..9,                %4
   maplist(all_distinct, Rows),                          %5
   transpose(Rows, Columns),                             %6
   maplist(all_distinct, Columns),                       %7
   Rows = [A,B,C,D,E,F,G,H,I],                           %8
   blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).    %9
blocks([], [], []).                                      %10
blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-         %11
all_distinct([A,B,C,D,E,F,G,H,I]),                       %12
blocks(Bs1, Bs2, Bs3).                                   %13
```

```
/*
problem(1, [[_,_,_, _,_,_, _,_,_],
[_,_,_, _,_,3, _,8,5],
[_,_,1, _,2,_, _,_,_],
[_,_,_, 5,_,7, _,_,_],
[_,_,4, _,_,_, 1,_,_],
[_,9,_, _,_,_, _,_,_],
[5,_,_, _,_,_, _,7,3],
[_,_,2, _,1,_, _,_,_],
[_,_,_, _,4,_, _,_,9]]).
```

Explaination:

Line 1: This is the main predicate for determining whether a Sudoku puzzle is valid. It accepts a list of lists Rows as input, with each inner list representing a puzzle row.

Line 2: This clause ensures that the Rows list has length 9 and that the puzzle is a 9x9 grid.

Line 3: This clause ensures that each puzzle row is the same length. It does this by applying the same_length predicate to the Rows list, followed by the maplist predicate on the result. The same_length predicate accepts two lists as input and returns true if their lengths are the same.

Line 4: This clause adds the condition that each digit from 1 to 9 appear exactly once in the puzzle. This is accomplished by ensuring that the set of all numbers in the puzzle is exactly identical to the set of all numbers ranging from 1 to 9. The append predicate joins two lists, and the ins predicate determines whether a number belongs to a set.

Line 5: This clause ensures that no duplicates appear in any of the puzzle's rows. It does this by using the all_distinct predicate on the rows list. The all_distinct predicate accepts a list as input and returns true if there are no duplicates in the list.

Line 6: This clause converts the puzzle's rows into columns. This is done because the blocks predicate operates on the puzzle's columns. The transposition predicate takes two lists as input and returns a new list with the ith element being the list of elements at the original lists' ith position.

Line 7: This clause ensures that no duplicates appear in any of the puzzle's columns. It does this by using the all_distinct predicate on the Columns list. The all_distinct predicate accepts a list as input and returns true if there are no duplicates in the list.

Line 8: This clause gives the puzzle rows the names A, B, C,..., I. This is done for a simple definition of the block's predicate.

Line 9: These clauses define the blocks predicate, which is used to ensure that there are no duplicate 3x3 subgrids in the puzzle. The predicate receives three lists, each of which represents a 3x3 subgrid of the puzzle.

Line 10: This clause defines the blocks predicate's base case, which is when all three subgrids are empty.

Line 11: The recursive case for the blocks predicate is defined in this clause. It confirms that there are no duplicates in the current subgrid before recursively calling the predicate to examine the next two subgrids.

Line 12: This clause ensures that there are no duplicates in the current subgrid. It accomplishes this by applying the all_distinct predicate to the subgrid's list of numbers.

Line 13: This clause calls the blocks predicate recursively to confirm the next two subgrids.

*/

3.

/*

% You are on an island where every inhabitant is either a knight or a knave. Knights always tell the truth, and knaves always lie. The

% following examples show how various cases can be solved with CLP(B), Constraint Logic Programming over Boolean variables.

% These examples appear in Raymond Smullyan's "What Is the Name of % this Book" and Maurice Kraitchik's "Mathematical Recreations".

*/

:- use_module(library(clpb)).

- % We use Boolean variables A, B and C to represent the inhabitants.
- % Each variable is true iff the respective inhabitant is a knight.
- % Notice that no search is required for any of these examples.

```
% Example 1: You meet two inhabitants, A and B.
% A says: "Either I am a knave or B is a knight."
example_knights(1, [A,B]) :- sat(A=:=(~A + B)).

% Example 2: A says: "I am a knave, but B isn't."
example_knights(2, [A,B]) :- sat(A=:=(~A * B)).

% Example 3: A says: "At least one of us is a knave."
example_knights(3, [A,B]) :- sat(A=:=card([1,2],[~A,~B])).

% Example 4: You meet 3 inhabitants. A says: "All of us are knaves."
% B says: "Exactly one of us is a knight."
example_knights(4, Ks) :- Ks = [A,B,C], sat(A=:=(~A * ~B * ~C)), sat(B=:=card([1],Ks)).

% Example 5: A says: "B is a knave."
% B says:"AandCareofthesamekind."
% WhatisC?
example_knights(5, [A,B,C]) :- sat(A=:= ~B), sat(B=:=(A=:=C)).

% Example 6: B says: "I am a knight, but A isn't."
% A says: "At least one of us is a knave."
example_knights(6, [A,B,C]) :- sat(B=:=(~B + A)). sat(B=:= ~A), sat(B=:=(A=:=C)).
```

Output:

```
A = B, B = 1.

% Example 7: A says: "At least one of us are knights."
% B says: "Either I am a knight or A is a knave."
example_knights(7, Ks) :- Ks = [A,B,C], sat(A=:=(A * B * C)), sat(B=:=(~B + A)).
‘
```

Output:

Ks = [1, 1, 1].

% Example 8: B says: "All of us are knights."

% A says. "Exactly one of us is a knave."

example_knights(8, Ks) :- Ks = [A,B,C], sat(B=:=(~A * ~B * ~C)), sat(A=:=card([1],Ks)).

Output:

Ks = [1, 0, 0].

PART 2

```
:- use_module(library(clpfd)).
reverse([],[]).
reverse([H|T], RevList):-
   reverse(T, RevT), append(RevT, [H], RevList).

SendHelp([],[],C,[T3]):- T3 #= C.
SendHelp([H1|T1],[H2|T2],Carry2,[H3|T3]):-
      H3 #= (H1+H2+Carry2) mod 10
      Carry #= (H1+H2+Carry2) div 10
      SendHelp(T1,T2,Carry,T3).

crypt1([H1|L1],[H2|L2],[H3|L3],L4) :-
   % Give constraints on variable values in L4
   % Heads cannot have value 0
   % Variable values are distinct in L4
   % Reverse the 3 input words
```

```
   % Call to helper function that does the sum with reversed words one iteration at a time
   L4 ins 0..9,
   H1 #\= 0, H2 #\= 0,
   all_different(L4),
   reverse([H1|L1], Out1), reverse([H2|L2], Out2), reverse([H3|L3], Out3),
  SendHelp(Out1,Out2,0,Out3)

/*
examples:
1.
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]).
S = 9
E = 5
N = 6
D = 7
M = 1
O = 0
R = 8
Y = 2


2.
crypt1([A,B,C,D],[M,O,R,E],[O,N,E,Y],[D,N,E,S,R,O,M,Y]), labeling([ff],[D,N,E,S,R,O,M,Y]).
A = 4
B = 2
C = 1
D = 5
M = 9
O = 0
```

R = 6

E = 3

Y = 8

3.

crypt1([D,O,O,R],[C,A,T],[S,I,X],[D,O,O,G,C,A,T,S,I,X]), labeling([ff],[D,O,O,G,C,A,T,S,I,X]).

D = 4

O = 7

O = 2

R = 8

C = 0

A = 3

T = 6

S = 1

I = 9

X = 5

*/