Assignment - 1
Submitted By: Sarvansh Prasher

1. **Prolog Code:**

```
% author - Sarvansh Prasher
% version 1.0
% date - 10th February 2020
% findpath(X,Y,Weight,Path) - this relation will represent
% the paths of going from X to Y and weight of those individual paths.

% path(node1,node2,distance) represents the relation of connection of one node to
% another node
path(a,b,1).
path(a,c,6).
path(b,e,1).
path(b,d,3).
path(b,c,4).
path(d,e,1).
path(c,d,1).

findpath(Z1,Z2,N) :- path(Z2,Z1,N).
findpath(Z1,Z2,N) :- path(Z1,Z2,N).

findpath(Z1,Z2,N,L) :- findpath(Z1,Z2,N,L,0).

findpath(Z1, Z2, N, [Z1,Z2],_) :- path(Z1, Z2, N).

findpath(Z1, Z2, N, [Z1|P], V) :- \+ member(Z1, V), path(Z1, Z, N1), findpath(Z, Z2, N2, P,
[Z1|V]), N is N1 + N2.
```

Output (Sample run):

?- findpath(a,e,Weight,Path).
Path = [a, b, e],
Weight = 2
Path = [a, b, d, e],
Weight = 5
Path = [a, b, c, d, e],
Weight = 7
Path = [a, c, d, e],
Weight = 8

2. **Prolog Code:**

```
% author - Sarvansh Prasher
% version 1.0

% hanoi (X, a, c, b) defines relation where number of discs are X
% ,a is starting peg,c is auxillary peg,b is final peg.
```

```
% Base case when we have only one disc and we have to put disc from a to c
hanoi(1,LEFT,CENTER,_):-
   write('Move '),
   write(LEFT),
   write(' to '),
   write(CENTER),
   nl.

% Recursive case which handles the moving of discs from peg.
hanoi(X,LEFT,CENTER,RIGHT):-
   X>1,
   X1 is X-1,
   hanoi(X1,LEFT,RIGHT,CENTER),
   hanoi(1,LEFT,CENTER,_),
   hanoi(X1,RIGHT,CENTER,LEFT).
```

Output (Sample run):

```
?- hanoi(3,a,c,b).
Move a to c
Move a to b
Move c to b
Move a to c
Move b to a
Move b to c
Move a to c
```

3. **Prolog Code:**

```
% author - Sarvansh Prasher
% version 1.0

% full_words(Z) will define the number Z in words. 0-9 number
% will be defined first in word(N) which will work as helper in
% forming words.

word(0):- print(zero).
word(1):- print(one).
word(2):- print(two).
word(3):- print(three).
word(4):- print(four).
word(5):- print(five).
word(6):- print(six).
word(7):- print(seven).
word(8):- print(eight).
word(9):- print(nine).

full_words(Z):-Zmod is Z mod 10, Zdiv is Z // 10,digit(Zdiv),
```

```
    word(Zmod).

digit(0).

digit(Z):- Z > 0,Zmod is Z mod 10,
    Zdiv is Z // 10,
    digit(Zdiv),
    word(Zmod),
    print(-).
```

Output (Sample run):

```
?- full_words(283).
two-eight-three
```

4. **Prolog Code:**

```
% author - Sarvansh Prasher
% version 1.0

% combination(N,T,L) defines relation where L will be the list formed by T by
% N combinations.

combination(0, _, []).
combination(N, [H|T], [H|L]) :- N1 is (N - 1),  combination(N1, T, L).
combination(N, [_|T], L) :- N > 0, combination(N, T, L).
```

Output (Sample run):

?- combination(3,[a,b,c,d,e,f],L).
L = [a, b, c]
L = [a, b, d]
L = [a, b, e]
L = [a, b, f]
L = [a, c, d]
L = [a, c, e]
L = [a, c, f]
L = [a, d, e]
L = [a, d, f]
L = [a, e, f]
L = [b, c, d]
L = [b, c, e]
L = [b, c, f]
L = [b, d, e]
L = [b, d, f]
L = [b, e, f]
L = [c, d, e]
L = [c, d, f]
L = [c, e, f]
L = [d, e, f]

5. **Prolog Code:**

```
% author - Sarvansh Prasher
% version 1.0

% color_map(L) defines the relation where L contains all the vertices with
% colors associated with them.

% Rules which represent the vertexes
vertex(1).
vertex(2).
vertex(3).
vertex(4).
vertex(5).
vertex(6).

% Rule for colors
color(red).
color(green).
color(yellow).
color(blue).

% Rules which represent edge between vertices.
edge(2,1).
edge(2,3).
edge(2,5).
edge(1,6).
edge(1,4).
edge(1,3).
edge(5,3).
edge(5,4).
edge(4,6).
edge(4,3).
edge(3,6).

% Predicate for connecting edges to two nodes
adjacent(X,Y) :- edge(Y,X);edge(X,Y).

% Rules for defining color predicate which will color the vertices
colorVertex([]).
colorVertex([colors(_,C)|Vertex]) :- color(C),colorVertex(Vertex).

% Rules for defining two vertexes having different color
linkedVertextColor([],_).
linkedVertextColor([(Vertex1,Vertex2)|RemainingList],FinalList):-
member(colors(Vertex1,C1),FinalList),
  member(colors(Vertex2,C2),FinalList),dif(C1,C2),
  linkedVertextColor(RemainingList,FinalList).
```

```
  % Rules for getting colored map after giving list and finding all objects in the given edges pair
  color_map(L) :-
    findall((Vertex1, Vertex2), edge(Vertex1, Vertex2), E),
    findall(Vertex1, vertex(Vertex1), Vertexes),
    findall(colors(Vertex2, _), member(Vertex1, Vertexes), L),
    linkedVertextColor(E,L),
    colorVertex(L).
```

Output (Sample run):

?- color_map(L).
L = [*colors*(2, red), *colors*(1, green), *colors*(3, yellow), *colors*(5, green), *colors*(6, red), *colors*(4, blue)]
L = [*colors*(2, red), *colors*(1, green), *colors*(3, yellow), *colors*(5, green), *colors*(6, blue), *colors*(4, red)]
L = [*colors*(2, red), *colors*(1, green), *colors*(3, yellow), *colors*(5, blue), *colors*(6, blue), *colors*(4, red)]
L = [*colors*(2, red), *colors*(1, green), *colors*(3, blue), *colors*(5, green), *colors*(6, red), *colors*(4, yellow)]
L = [*colors*(2, red), *colors*(1, green), *colors*(3, blue), *colors*(5, green), *colors*(6, yellow), *colors*(4, red)]
.
.
.
.

6. **Prolog Code:**

```
% author - Sarvansh Prasher
% version 1.0

% queens(N,Qs) gives the solution where N represents how many queens need to be there
% on board and Qs will give you the solution of where will it be kept on board.

:- use_module(library(clpfd)).

queens(N,Solution) :- generateRowList(N,Rows), nQueens(Rows,[],Solution).

% Predcicate generateRowList(N,Rows) is for generating a list
% of N elements which will help in filling rows.
generateRowList(N,Rows) :- generateRowList(1,N,Rows).
generateRowList(N,N,[N]) :-!.
generateRowList(Rows,N,[Rows|List]) :- N >Rows, N >1, R1 is Rows+1,
  generateRowList(R1,N,List).

% select(Element,List1,List2) predicate will be used for selecting the row
% from main rows list
select([L|L1],L1,L).
```

```
select([R|R1],[R|R2],L):-
        select(R1,R2,L).

% Predicate nQueens(Rows,ChessBoard,Solution) relation for solving problem of
% where to put queen on chessboard.
nQueens([],Solution,Solution).
nQueens(Rows,ChessBoard,Solution) :- select(Rows,R1,RemainingRows) ,
                            checkIfValid(ChessBoard,RemainingRows),
                            nQueens(R1,[RemainingRows|ChessBoard],Solution).

% checkIfValid(ChessBoard,RemainingRows) for checking row,column,diagonal wise
% whether it is a valid move.
checkIfValid(ChessBoard,RemainingRows):-
checkIfValid(ChessBoard,RemainingRows,1).
checkIfValid([],_,_):-!.
checkIfValid([RemainingList|R], Rows, Distane0) :-
      Rows #\= RemainingList,
      abs(Rows - RemainingList) #\= Distane0,
      Distance1 #= Distane0 + 1,
      checkIfValid(R, Rows, Distance1).
```

Output (Sample run):

?- queens(6, Qs).
Qs = [5, 3, 1, 6, 4, 2]
Qs = [4, 1, 5, 2, 6, 3]
Qs = [3, 6, 2, 5, 1, 4]
Qs = [2, 4, 6, 1, 3, 5]

7. **Prolog Code:**

```
% author - Sarvansh Prasher
% version 1.0

% goldbach(N,L) :- N is even number and L is the list of the two
% prime numbers that when added sums up to N.

% Relation for finding whether a number is prime number
% (Submitted in Mini Assignment 2)
div(X, Y, Z) :- Z is X / Y.
greater(X, Y) :- X < Y.
divisible(X, Y) :- div(X, Y, Z), integer(Z).
notPrime(X, Y) :- Y > 1, divisible(X, Y).
notPrime(X, Y) :- greater(Y, X / 2), notPrime(X, Y+1).
notPrime(Z) :- Z > 2, notPrime(Z, 2).
prime(Z) :- not(notPrime(Z)).

% Relation for finding list of prime numbers.
```

```
% Condition for checking prime number combinations for given number N.
primes(N,N1) :- N1 is N + 2, prime(N1),!.
primes(N,N1) :- N2 is N + 2, primes(N2,N1).

% After one number has been found, checking whether N-SolutionFound(Z) is also
% a prime number and making sure it is greater than Z.
goldbach(N,[Z,Z1],Z) :- Z1 is N - Z, prime(Z1), Z<Z1.
goldbach(N,L,Z) :- Z < N, primes(Z,Z1), goldbach(N,L,Z1).

% Converting goldbach/2 to goldbach/3 by taking extra variable 3.
goldbach(N,L) :- N mod 2 =:= 0, N > 4, goldbach(N,L,3).
```

<u>Output (Sample run):</u>

?- goldbach(30, L).
L = [7, 23]
L = [11, 19]
L = [13, 17]