ASSIGNMENT 1

1. GRAPH

```
% File Name: findpath.pl

% findpath(X,Y,Weight,Path).

% where X, Y = name of the node

% Weight = weight of the path taken

% Path = path taken in the form of a list


% Sample run

% ?- findpath(a,e,Weight,Path).

% Path = [a,b,e]

% Weight = 2;

% ......

% Path = [a,c,d,e]

% Weight = 8 ;

% ......


% defining edges in a graph along with weight associated with the edge.


edge(a,b,1).

edge(a,c,6).

edge(b,c,4).

edge(b,d,3).

edge(c,d,1).

edge(d,e,1).

edge(e,b,1).
```

```
% To represent that the edges are bi-directional.

connected(X,Y,W) :- edge(X,Y,W) ; edge(Y,X,W).

% A- Source , B- Destination node

% Path- path from src node to destination node

% Weight - Weight of the edges traversed


findpath(Src_Vertex,Dest_Vertex,Weight,Path) :-
travel_graph(Src_Vertex,Dest_Vertex,[Src_Vertex],Q,Weight),

   reverse(Q,Path).


travel_graph(Src_Vertex,Dest_Vertex,P,[Dest_Vertex|P],Weight) :-
connected(Src_Vertex,Dest_Vertex,Weight).


% starting from source vertex, visits each node and then visited nodes are stored in a list so that
it doesn't have cycles.

% if the vertex is a member of the visited list then that path is not considered.


travel_graph(Src_Vertex,Dest_Vertex,Visited,Path,Weight) :- connected(Src_Vertex,C,W1),

          C \==Dest_Vertex,

          \+member(C,Visited),

          travel_graph(C,Dest_Vertex,[C|Visited],Path,W2),

          Weight is W1 + W2.
```

<u>SAMPLE RUN</u>

```
findpath(a,e,Weight,Path).

WeightPath

2         [a, b, e]

7         [a, b, c, d, e]
```

5	[a, b, d, e]
8	[a, c, d, e]
11	[a, c, d, b, e]
11	[a, c, b, e]
14	[a, c, b, d, e]

2. <u>TOWER OF HANOI</u>

```
% Tower of hanoi

% hanoi( no_of_disks , src, dest, temp )

hanoi(1,Source,Destination,_):-write('Move disk from '),write(Source),write(' to
'),write(Destination),nl.

% for each recursive call, No of disks are reduced by 1 which is equal to M.

% recursively moves the disk from source to destination using a temporary tower.

hanoi(No_of_disks,Source,Destination,Temp):-No_of_disks>1,M is No_of_disks-1,

        hanoi(M,Source,Temp,Destination),

        hanoi(1,Source,Destination,_),

        hanoi(M,Temp,Destination,Source).
```

<u>SAMPLE RUN</u>

hanoi(3,a,b,c).

Move disk from a to b

Move disk from a to c

Move disk from b to c

Move disk from a to b

Move disk from c to a

Move disk from c to b

Move disk from a to b

true

3. NUMBERS IN WORDS.

```
% Converting numbers to words.
% Sample run
% ?- full_words(283).
% two-eight-three

 % numbers with respective words are written.

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

```
full_words(Nums) :-          % This is top-level predicate.
   NDiv is Nums // 10,       %  prints the first digit unconditionally,
   num_words(NDiv),                  % lets you handle the case when the number is zero.
   NMod is Nums mod 10,
   num(NMod).

num_words(0).                % When we reach zero, we stop printing.

num_words(Nums) :-           % Otherwise, we follow this algorithm
   Nums > 0,                 % with one modification- the dash is printed
   NDiv is Nums // 10,       % unconditionally before printing the digit.
   num_words(NDiv),          % recursive call
   NMod is Nums mod 10,
   num(NMod),
   write('-').               % writes hyphen after each number written in words.
```

SAMPLE RUN

full_words(283).

two-eight-three

4. COMBINATIONS

```
% Generate the combinations of K distinct objects chosen from the N elements of a list:
% Sample run :
% ?- combination(3,[a,b,c,d,e,f],L).
% L = [a,b,c] ;
```

% L = [a,b,d] ;

% L = [a,b,e] ;

% ......

% combination(No_of_ways, List_of_numbers/chars , L-o/p of_lists_of_possible_combinations).

combination(1, [H|_], [H]).

combination(N, [H|T], [H|Com]) :-

integer(N), N1 is N - 1, % checks if it is an integer and reduces N by 1.

N1 > 0, % checks if N1 is greater than 0.

combination(N1, T, Com).

% recursively calls top level predicate to find different combinations.

combination(N, [_|T], Com) :- combination(N, T, Com).

<u>SAMPLE RUN</u>

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

L = [a,b,c] ;

L = [a,b,d] ;

L = [a,b,e] ;

5. MAP COLORING.

% color_map1(LIST_MAP_NO, LIST_OF_COLORS, COLORING_LIST) takes a list of area names/numbers associated with the areas/MAP and list of

% colors , and returns COLORING to be a list of pairs [AREA, COLOR], where every area is assigned a color, and no

```
% adjacent areas are assigned the same color.

adjacent(1,2).
adjacent(1,3).
adjacent(1,4).
adjacent(1,6).
adjacent(2,3).
adjacent(2,5).
adjacent(3,4).
adjacent(3,5).
adjacent(3,6).
adjacent(4,5).
adjacent(4,6).

% checks for adjacency.

adjacent1(X,Y) :-adjacent(X,Y).
adjacent1(X,Y) :-adjacent(Y,X).

% Top level predicate.

color_map(L):-color_map1([1,2,3,4,5,6],[red,blue,green,yellow], L).

% base case : if the map list is empty then no coloring is done.

color_map1([],_, []).
```

```
% Recursive case: color the tail of the MAP, then add a color for the head
% in a way that does not conflict.

color_map1([HEAD | TAIL], COLORS, [[HEAD, HCOLOR] | TAIL_COLORING]) :-
  color_map1(TAIL, COLORS, TAIL_COLORING),
  member(HCOLOR, COLORS),
 \+conflicts(HEAD, HCOLOR, TAIL_COLORING).    % checks if there is conflict in   coloring the
                                              adjacent sides.

conflicts(AREA1, COLOR, [[AREA2, COLOR] | _]) :-
   adjacent1(AREA1, AREA2).

% Recursive case: continue to search down the list COLORING.
conflicts(AREA, COLOR, [_|COLORING]) :-
   conflicts(AREA, COLOR, COLORING).

% member as usual

member(X,[X|_]).
member(X,[_|TAIL]) :- member(X,TAIL).
```

SAMPLE RUN

```
color_map(L).
[[1,yellow],[2,blue],[3,green],[4,blue],[5,red],[6,red]]
[[1,green],[2,blue],[3,yellow],[4,blue],[5,red],[6,red]]
```

6. N-QUEENS PROBLEM

```
% goal predicate will take the form:
% queens (N, Qs).
% where N = the number of queens
% Qs = solution to the problem

n_queens(N, Qs) :-
  range(1, N, Ns),
  n_queens(Ns, [], Qs).

% To generate list of N numbered list[number of queens in chess board N*N]

range(N, N, [N]) :- !.                  % Base Case : when reaches N , cuts from this fuction.

range(M, N, [M|Ns]) :-
  M < N,
  M1 is M+1,
  range(M1, N, Ns).

n_queens([], Qs, Qs).                   % base case

n_queens(UnplacedQs, SafeQs, Qs) :-
  select(UnplacedQs, UnplacedQs1,Q),
  chck_not_attacked(SafeQs, Q),
  n_queens(UnplacedQs1, [Q|SafeQs], Qs).
```

% checks if the queen is safe by checking its not attacked

% by other queen diagonally or any rows or columns.

chck_not_attacked(As, A) :- chck_not_attacked(As, A, 1).

chck_not_attacked([], _, _) :- !.   % base case

chck_not_attacked([B|Bs], A, N) :-   % to check if the the queen is not attacked by any other queen

% which is already placed on the chess board.

A =\= B+N,

A =\= B-N,

N1 is N+1,

chck_not_attacked(Bs, A, N1).

select([A|As], As, A).

select([B|Bs], [B|Cs], A) :- select(Bs, Cs, A).

SAMPLE RUN

n_queens(4,Qs).

Qs

[3,1,4,2]

[2,4,1,2]

7. GOLDBACH

% prime number should not be an even number

% it shuld not have factors other than 1 and itself

% Goldbach = Every even integer greater than 2 can be expressed as the sum of two primes.

```
is_prime(2).                                              % declaring 2 as prime no.

is_prime(N) :- integer(N), N mod 2 =\= 0, \+ has_factor(N,3).


has_factor(N,L) :- N mod L =:= 0.

has_factor(N,L) :- L * L < N, L1 is L + 2, has_factor(N,L1).          % checks if the
                                                          number has a factor!

% case if even number is 4, then 4= 2+2.

goldbach(4,[2,2]) :- !.

% checks if N(i/p) is even number, N >4 , calls goldbach(even_no,List,3(prime_no)).

goldbach(N,L) :- integer(N), N mod 2 =:= 0, N > 4, goldbach(N,L,3).

goldbach(N,[P,Q],P) :- Q is N - P, is_prime(Q), Q>1.

% Q is another no after subtracting 3 from N. If Q obtained is prime_no then other choices
are discarded.

goldbach(N,L,P) :- P < N, gen_next_prime(P,P1), goldbach(N,L,P1).


gen_next_prime(P,P1) :- P1 is P + 2, is_prime(P1), !.

% To generated next set of prime numbers. Begins by adding 2 to the number obtained.

gen_next_prime(P,P1) :- P2 is P + 2, gen_next_prime(P2,P1).
```

<u>SAMPLE RUN</u>

goldbach(24,L).

L

[5,19]

[7,17]

[11,13]

[13,11]

[17,7]

[19,5]