Q1: Write a program in prolog to implement TowerOfHanoi(N) where N represents the number of disks.

move(0,_).

move (1, A,,C):- inform (A,C), 1. & base condition

move (N, A, B, C): Mis N-1,

move (M, A, C, B),

inform (A,C),

move (M, B, A, C).

inform(A,B): write ('Move one disk from tower '), write(A), write(' to tower '), write(B), nl.

OUTPUT:

```
?- move(3, left, middle, right).
Move one disk from tower left to tower right
Move one disk from tower left to tower middle
Move one disk from tower right to tower middle
Move one disk from tower left to tower right
Move one disk from tower middle to tower left
Move one disk from tower middle to tower right
Move one disk from tower left to tower right
true.

?-
```

Q2: Write a program to implement the Hill climbing search algorithm in prolog.

%Define the hill climbing algorithm

hill_climb (List, Max) :- select_max (List, Max, Rest), hill_climb (Rest, Max, Max).

hill_climb ([), Max, Max).

hill_climb (List, CurrentMax, Max) :- select_max (List, NewMax, Rest), NewMax > CurrentMax,

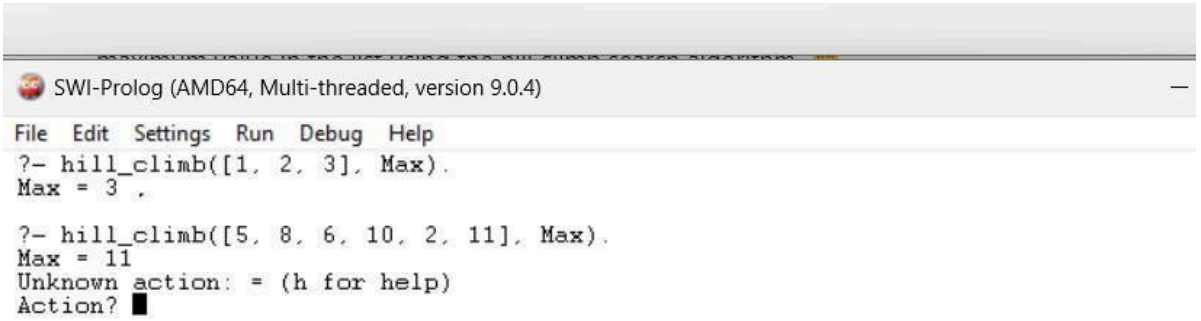
hill climb (Rest, NewMax, Max). hill_climb (List, CurrentMax, Max) :- select_max (List, NewMax, Rest), NewMax =< CurrentMax, hill climb (Rest, CurrentMax, Max).

%Select the maximum value in a list

select_max([X], x, []).

select_max([X|Xs], Max, [X|Rest]) :- select max (Xs, Max, Rest), Max > X.

select_max([X/Xs), X, Xs) :- select max (Xs, Max, _), X >= Max.



Q3: Write a program to implement the Best first search algorithm in prolog.

%Define heuristic values for nodes

heuristic(a, 5).

heuristic(b, 3). heuristic(c, 8).

heuristic(d, 2).

heuristic(e, 6).

```
% Define edges between nodes

edge (a, b, 2).

edge (a, c, 4).

edge (b, d, 3). edge (c, e, 5).

edge (d, e, 1).

% IDDFS algorithm

iddfs (Start, Goal, Solution) :-

max_depth (MaxDepth), between (1, MaxDepth, Depth),

iddfs_search (Start, Goal, [Start], Depth, Solution).

iddfs_search (Goal, Goal,

iddfs_search (Current, Goal, Visited, Depth, [Current/Path]) :- Depth > 0,
```

Depthi is Depth - 1, move (Current, Next),

A .

+ member (Next, Visited), iddfs search (Next, Goal, [Next Visited), Depthl, Path)

max_depth (10) % Set the maximum depth for IDDES

OUTPUT:

```
.

?- iddfs(a, e, Path).
Path = [a, b, c, d]
```

Q4: Write a program to implement A* search algorithm in Prolog.

```
% A search algorithm /

astar (Start, Final,, Tp) :-
      estimation (Start, Final, E),
      astarl([(E, E, 0, [Start])], Final, _, Tp).

astarl([(_,_, Tp, [Final | R]) |  _], Final, [Final | R], Tp) :-
      reverse ([Final | R), L3),
     write('Path = '), write (L3).

astarl([(_,_, P, [X | R1]) | R2), Final, C, Tp) :-
     findall ((NewSum, E1, NP, [Z, X | R1)), (street(X, Z, V), not (member
(Z,R1)),
     NP is P+ V,estimation (Z, Final, E1), NewSum is E1+ NP,L),

 append (R2, L, R3),
 sort (R3, R4),
astar1 (R4, Final, C, TP).

estimation (C1, C2, Est):-
```

area (C1, X1, Y1),

area (C2, X2, Y2),

DX is X1 - X2,

DY is Y1-Y2,

Est is sqrt (DX* DX + DY *DY).

OUTPUT:

```
?- astar(1, 10, Path, _).
Path = [1,5,6,10]
true .

?- astar(1, 9, Path, _).
Path = [1,5,6,9]
true .
```

Q5: Write a program to implement the min-max search algorithm in Prolog.

%Define the game state

state([_,_,_,_,_,_,_,_,_]).

%Define the players

```
player(x).

player(o).

%Define the moves

move(state([_, B, C, D, E, F, G, H, I]), 1, state([x,B,C,D,E,F,G,H,I])) :-player(x).
move(state([A, _, C, D, E, F, G, H, I]), 1, state([A,x,C,D,E,F,G,H,I])) :-player(x).
move(state([A, B, _, D, E, F, G, H, I]), 1, state([B,B,x,D,E,F,G,H,I])) :-player(x).
move(state([A, B, C,  _, E, F, G, H, I]), 1, state([C,B,C,x,E,F,G,H,I])) :-player(x).
move(state([A, B, C, D,  _, F, G, H, I]), 1, state([D,B,C,D,x,F,G,H,I])) :-player(x).
move(state([A, B, C, D, E,  _, G, H, I]), 1, state([E,B,C,D,E,x,G,H,I])) :-player(x).
move(state([A, B, C, D, E, F,  _, H, I]), 1, state([F,B,C,D,E,F,x,H,I])) :-player(x).
move(state([A, B, C, D, E, F, G,  _, I]), 1, state([G,B,C,D,E,F,G,x,I])) :-player(x).
move(state([A, B, C, D, E, F, G, H,  _]), 1, state([H,B,C,D,E,F,G,H,x])) :-player(x).

move(state([_, B, C, D, E, F, G, H, I]), 1, state([o,B,C,D,E,F,G,H,I])) :-player(x).
move(state([A, _, C, D, E, F, G, H, I]), 1, state([A,o,C,D,E,F,G,H,I])) :-player(x).
move(state([A, B, _, D, E, F, G, H, I]), 1, state([B,B,o,D,E,F,G,H,I])) :-player(x).
move(state([A, B, C,  _, E, F, G, H, I]), 1, state([C,B,C,o,E,F,G,H,I])) :-player(x).
move(state([A, B, C, D,  _, F, G, H, I]), 1, state([D,B,C,D,o,F,G,H,I])) :-player(x).
move(state([A, B, C, D, E,  _, G, H, I]), 1, state([E,B,C,D,E,o,G,H,I])) :-player(x).
```

```
move(state([A, B, C, D, E, F,  _, H, I]), 1, state([F,B,C,D,E,F,o,H,I])) :-player(x).
move(state([A, B, C, D, E, F, G,  _, I]), 1, state([G,B,C,D,E,F,G,o,I])) :-player(x).
move(state([A, B, C, D, E, F, G, H,  _]), 1, state([H,B,C,D,E,F,G,H,o])) :-player(x).


%define win conditions
win(state([x,x,x,_,_,_,_,_,_]),x).
win(state([_,_,_,x,x,x,_,_,_]),x).
win(state([_,_,_,_,_,_,x,x,x]),x).
win(state([x,_,_,x,_,_,x,_,_]),x).
win(state([_,x,_,_,x,_,_,x,_]),x).
win(state([_,_,x,_,_,x,_,_,x]),x).
win(state([x,_,_,_,x,_,_,_,x]),x).
win(state([_,_,x,_,x,_,x,_,_]),x).

%Define the draw condition

draw(State) :-

    \+ win(State, x),

    \+ win(State, o),
```

```
    \+ move(State, _, _).

%Define the utility function

utility(state([A, B, C, D, E, F, G, H, I]), Value) :-
    (win(state([A, B, C, D, E, F, G, H, I]), x) -> Value = 1;
     win(state([A, B, C, D, E, F, G, H, I]), o) -> Value = -1;
    draw(state([A, B, C, D, E, F, G, H, I])) -> Value = 0).

%Define the minimax algorithm

minimax(State, Value) :-
    utility(State, Value).
minimax(State, Value) :-
    findall(NewState, move(State,_, NewState), NewStates),
    best(NewStates, Value).

best([State], Value) :-
    minimax(State, Value).

best([State|States], Value) :-
    minimax(State, V1),
    best(States, V2),
```

(player(x) -> max(V1, V2, Value); min(V1, V2, Value)).

OUTPUT:

```
?- minimax(state([x,o,x,o,x,o,_,_,_]),Value).
Value = 1 .

?- minimax(state([o,x,o,x,_,_,_,_,_]),Value).
Value = 1 .
```

Q6: Write a program to solve the Water-Jug Problem in Prolog.

```
%Define the initial state and the goal state
initial_state((0, 0)).
goal_state((4,_)).

%Define the actions possible in the problem
action((Jug1, Jug2), fill_jugl, (5, Jug2)) :-
        Jugl < 5.

action((Jug1, Jug2), fill_jug2, (Jug1, 3)) :-
       Jug2 < 3.

action((Jug1, Jug2), empty_jugl, (0, Jug2)) :-
        Jug1 > 0.

action((Jug1, Jug2), empty_jug2, (Jug1, 0)) :-
       Jug2 > 0.

action((Jug1, Jug2), pour_jugl_to_jug2, (NewJugl, NewJug2)) :-
       Jug1 > 0,

Total is Jug1 + Jug2,
```

```
NewJug2 is min(Total, 3),

NewJugl is Jugl (NewJug2 -Jug2).

action((Jugl, Jug2), pour_jug2_to_jugl, (NewJugl, NewJug2)) :- Jug2 > 0,

Total is Jugl + Jug2,

NewJugl is min(Total, 5),

NewJug2 is Jug2 (NewJug1 -Jug1).

%Define the predicate to solve the problem using depth-first search

solve (State,_,[]) :- goal_state (State).
solve (State, Visited, [Action|Rest)) :-
        action (State, Action, NextState),
+ member (NextState, Visited),

solve (NextState, [NextState|Visited), Rest).
```

OUTPUT:

```
?- initial_state(InitialState), solve(InitialState, [InitialState], Actions).
InitialState = (0, 0),
Actions = [fill_jug1, fill_jug2, empty_jug1, pour_jug2_to_jug1, fill_jug2, pour_jug2_to_jug1, empty_jug1, pour_jug2_to_jug1, fill_jug2|...]
```

Q7: Implement sudoku problem (minimum 9X9 size) using constraint satisfaction in Prolog.

```
sudoku (Rows) :-

    length (Rows, 9), maplist (same_length (Rows), Rows),

    append (Rows, Vs), Vs ins 1..9,

    maplist (all_distinct, Rows),

    transpose (Rows, Columns),

    maplist (all_distinct, Columns),

    Rows= [As, Bs, Cs, Ds, Es, Fs, Gs, Hs, Is],

    blocks (As, Bs, Cs),

    blocks (Ds, Es, Fs),
```

```
    blocks (Gs, Hs, Is).


blocks([], [], []).


blocks ([N1, N2, N3/Ns1], [N4, N5, N6 NS2], [N7, NB, N9 (Ns3]):-
    all_distinct([N1, N2, N3,N4, N5, N6, N7, NB, N9]),

    blocks (Ns1, Ns2, Ns3).




problem (1, [_,_,_,_,_,_,_,_,_],

            [_,_,_,_,_,3,_,8,5],

            [_,_,1,_,2,_,_,_,_],

            [_,_,_,5,_,7,_,_,_],

            [_,_,4,_,_,_,1,_,_],

            [_,9,_,_,_,_,_,_,_],

            [5,_,_,_,_,_,_,7,3],

            [_,_,2,_,1,_,_,_,_],

            [_,_,_,_,4,_,_,_,9]]).
```

OUTPUT:

```
?- problem(1, Grid), sudoku(Grid).
Grid = [[9, 8, 7, 6, 5, 4, 3, 2|...], [2, 4, 6, 1, 7, 3, 9|...], [3, 5, 1, 9, 2, 8|...], [1, 2, 8, 5, 3|...], [6, 3, 4, 8|...], [7, 9, 5|...], [5, 1|...],
[4|...], [...|...]].

?-
```

Q8: Write a Prolog program to implement the family tree and demonstrate the family relationship.

```
male(shantanu).
male(bheeshma).
male(chitrangada).
male(vichitravirya).
male(pandu).
male(yudhishtra).
male(bheema).
male(arjuna).
male(nakula).
male(sahadeva).
male(dhritarashtra).
male(duryodhana).
male(dushasana).

female(ganga).
female(satyavati).
female(ambika).
female(ambalika).
female(gandhari).
female(kunti).
female(madri).
female(duhsala).

parent_of(shantanu,bheeshma).
parent_of(shantanu,chitrangada).
parent_of(shantanu,vichitravirya).
parent_of(ganga,bheeshma).
parent_of(satyavati,chitrangada).
parent_of(satyavati,vichitravirya).
parent_of(vichitravirya,dhritarashtra).
parent_of(vichitravirya,pandu).
parent_of(ambika,dhritarashtra).
parent_of(ambalika,pandu).
parent_of(dhritarashtra,duryodhana).
parent_of(dhritarashtra,dushasana).
parent_of(dhritarashtra,duhsala).
parent_of(gandhari,duryodhana).

parent_of(gandhari,dushasana).
parent_of(gandhari,duhsala).
parent_of(pandu,yudhishtra).
parent_of(pandu,bheema).
parent_of(pandu,arjuna).
parent_of(pandu,nakula).
parent_of(pandu,sahadeva).
parent_of(kunti,yudhishtra).
parent_of(kunti,bheema).
parent_of(kunti,arjuna).
parent_of(madri,nakula).
parent_of(madri,sahadeva).
```

```
?- father_of(X,arjuna).
X = pandu ;

?- mother_of(X,duhsala).
X = gandhari ;

?- brother_of(X,bheema).
X = yudhishtra ;

?- sister_of(X,duryodhana)
|
|
X = duhsala
```

/*RULES*/

father_of (X, Y):-male (X),parent_of (X, Y).

```
mother_of (X, Y):-female (X), parent_of (X, Y).

grandfather_of (X, Y): male (X), parent_of (X,Z),parent_of(Z,Y).

sister_of (X, Y):-% (X, Y or Y,X)%
        female (X),
        father_of (F, Y), father_of (F,X),X\=Y.

sister_of (X, Y): female (X),
    mother_of (M, Y), mother_of (M,X),X\=Y.

aunt_of (X, Y): female (X),
    parent_of (Z, Y), sister_of (Z,X),!.

brother_of (X, Y):- %(X, Y or Y,X)%
    male (X),
    father_of (F, Y), father_of (F,X),X \ =Y.

brother_of (X,Y): male(X),
    mother_of (M, Y),
    mother_of (M,X),X \= Y.

uncle_of (X, Y):-
     parent_of (Z, Y), brother_of (Z,X).

ancestor_of (X, Y): parent_of (X, Y).
```

```
ancestor_of (X, Y): parent_of (X, Z),
        ancestor of (Z,Y)
```

Q9: Write a prolog program to implement knowledge representation using frames with appropriate examples.

```
frame (john, [
        [name, 'John Doe'],
        [age, 30],
        [gender, male],
        [occupation, engineer]
]).

frame (jane, [
        [name, Jane Smith'],
        [age, 28],
        [gender, female],
        [occupation, doctor]

]).

frame (car1, [
        [make, toyota],
```

```
    [model, 'Corolla'],
    [year, 2018),
    [color, blue]

]).

frame(car2, [
    [make, honda),
    [model, 'Civic'],
    [year, 2020],
    [color, red]
]).
get_frame_property(Frame, Property, Value) :-
     frame (Frame, Properties),
    member ([Property, Value], Properties).
```

OUTPUT:

```
?- get_frame_property(john, name, Name).
Name = 'John Doe' ,

?- get_frame_property(car2, color, Color).
Color = red.

?-
```

Q10: Write a Prolog program to implement conc (L1, L2, L3) where appended with L1 to get the resulted list L3.

```
conc(L1, L2, L3) :-
```

```
    append (L1, L2, L3).


% Example query

%?- conc([1, 2], [3, 4], L3).

% L3 = [1, 2, 3, 4] A
```

OUTPUT:

```
?- conc([1, 2], [3, 4], L3).
L3 = [1, 2, 3, 4].

?- conc([1, 2], [3, 4,5,67,7,8], L3).
L3 = [1, 2, 3, 4, 5, 67, 7, 8].

?-
```

Q11: Write a Prolog program to implement reverse (L, R) where List L is original and List R is reversed list.

```
reverse([], []).

reverse([HIT], R) :-

    reverse (T, RevT),

    append (RevT, [H], R).


% Example query

%?- reverse ([1, 2, 3, 4], R).

% R = [4, 3, 2, 1]
```

OUTPUT:

```
?- reverse([1, 2, 3, 4], R).
R = [4, 3, 2, 1].

?-
```

Q12: Write a prolog program to generate a parse tree of a given sentence in English language assuming the grammar required for parsing.

% Define the CFG rules

s(Tree) --> np (NP), vp (VP), { Tree = s(NP, VP) }.

np(Tree) --> det (Det), n(N), (Tree = np (Det, N)).

vp (Tree) --> v (V), np (NP), { Tree = vp (V, NP) }.

%Lexicon (terminal rules)

det (the) --> [the].

det (a) --> [a].

n(man) --> [man].

n(dog) --> [dog].

(bites) --> [bites].

v(likes) --> [likes].

OUTPUT:

```
?- s(ParseTree, [the, man, bites, the, dog], []).
ParseTree = s(np(the, man), vp(bites, np(the, dog))).

?-
```

Q13: Write a Prolog program to recognize context free grammar a^nb^n.

%Define the context-free grammar rules

s--> [].

s--> [a], s, [b].

% Define the predicate to recognize the string

recognize (String) :-

phrase (s, String).

OUTPUT:

```
.

?- recognize([a,a,b,b]).
true .

?- recognize([a,a,b,b,a]).
false.

?-
```