1. WAP to solve Towers of Hanoi problem.

Program:

```
move(1,X,Y,_):- write('Move top disk from '),
        write(X), write(' to '),
        write(Y),nl.
move(N,X,Y,Z):- N>1, M is N-1,
        move(M,X,Z,Y), move(1,X,Y,_),
        move(M,Z,Y,X).
```

2. Given the following facts:
   Steve only likes easy course.
   Science course are hard.
   All the courses in the basket weaving department are easy.
   BK301 is a basket weaving course.

   WAP to find "What course would Steve like?"

   Program:

```
likes(steve,X):- easy(X).
hard(X):- science(X).
easy(X):- baskweav(X).
baskweav(bk301).
```

3. Given the following facts:
   All people who are not poor and are smart are happy.
   Those people who read are not stupid but smart.
   John could read and is wealthy.
   Happy people have exciting lives.

   WAP to prove "John has an exciting life".

   Program:

```
happy(X):-wealthy(X),smart(X).
smart(X):-reads(X).
wealthy(john).
reads(john).
exciting_life(A):-happy(A).
```

4. Given the following facts:
   John likes all food.
   Apples are food.
   Chicken is food.
   Anything anyone eats & isn't killed by is food.
   Bill eats Peanuts and is still alive.
   Sue eats everything Bill eats.

   WAP to prove that "John likes peanuts" and find "what food does Sue eat?"

   Program:

```
likes(john,Y):- food(Y).
food(apple).
food(chicken).
food(Y):- eat(X,Y),alive(X,Y).
alive(bill,peanut).
eat(bill,peanut).
eat(sue,Y):- eat(bill,Y).
```

5. Given the following facts:
   Marcus was a man.
   Marcus was a Pompeian.
   All Pompeian's were Roman.
   Caesar was a ruler.
   All Roman were either loyal to Caesar or hated him.
   Everyone is loyal to someone.
   People only try to assassinate ruler they are not loyal to.
   Marcus try to assassinate Caesar.
   All men are people.

   WAP to find "Is Marcus loyal to Caesar?", "Does Marcus hate Caesar?"

   Program:

```
man(marcus).
pompeian(marcus).
roman(X):- pompeian(X).
ruler(caesar).
hate(X,caesar):- roman(X),not(loyalto(X,caesar)).
loyalto(X,Y):- man(X),man(Y).
loyalto(X,Y):- person(X),ruler(Y),not(tryassasinate(X,Y)).
tryassasinate(marcus,caesar).
person(X):- man(X).
```

6. Given the following facts:
   Anyone passing his history examination & winning the lottery is happy.
   Anyone who studies or is lucky can pass all his exam.
   John didn't study.
   John is lucky.
   Anyone who is lucky wins the lottery.

   WAP to prove "John is happy"

   Program:

```
happy(X):-pass(X,history),win(X,lottery).
pass(X,_):-study(X);lucky(X).
lucky(john).
study(not(john)).
win(X,lottery):-lucky(X).
```

7. WAP to solve the 4-3 Gallon Water Jug Problem.
Water Jug Problem: We have a four-gallon jug of water and a three-gallon jug of water and there is a tap that can be used to fill the jugs with water.
The challenge of the problem is to be able to put exactly two gallons of water in the four-gallon jug, even though there are no markings on the jugs.

Program:

```
solution(P) :-
 path(0, 0, [state(0, 0)], P).
path(2, 0, [state(2, 0)|_], []).
path(0, 2, C, ['Pour 2 gallons from 3-Gallon jug to 4-gallon.'|R] )
:-
not(member(state(2, 0), C)),
path(2, 0, [state(2, 0)|C], R).
path(X, Y, C, ['Fill the 4-Gallon Jug.'|R]) :-
X < 4,
not(member(state(4, Y), C)),
path(4, Y, [state(4, Y)|C], R).
path(X, Y, C, ['Fill the 3-Gallon Jug.'|R]) :-
Y < 3,
not(member(state(X, 3), C)),
path(X, 3, [state(X, 3)|C], R).
path(X, Y, C, ['Empty the 4-Gallon jug on ground.'|R]) :-
X > 0,
not(member(state(0, Y), C)),
path(0, Y, [state(0, Y)|C], R).
path(X, Y, C, ['Empty the 3-Gallon jug on ground.'|R]) :-
Y > 0,
not(member(state(X, 0), C)),
path(X, 0, [state(X, 0)|C], R).
path(X, Y, C, ['Pour water from 3-Gallon jug to 4-gallon until it
is full.'|R]) :-
X + Y >= 4,
X < 4,
Y > 0,
NEW_Y is Y - (4 - X),
```

```
        not(member(state(4, NEW_Y), C)),
        path(4, NEW_Y, [state(4, NEW_Y)|C], R).
        path(X, Y, C, ['Pour water from 4-Gallon jug to 3-gallon until it
is full.'|R]) :-
        X + Y >=3,
        X > 0,
        Y < 3,
        NEW_X is X - (3 - Y),
        not(member(state(NEW_X, 3), C)),
        path(NEW_X, 3, [state(NEW_X, 3)|C], R).
        path(X, Y, C, ['Pour all the water from 3-Gallon jug to 4-
gallon.'|R]) :-
        X + Y =< 4,
        Y > 0,
        NEW_X is X + Y,
        not(member(state(NEW_X, 0), C)),
        path(NEW_X, 0, [state(NEW_X, 0)|C], R).
        path(X, Y, C, ['Pour all the water from 4-Gallon jug to 3-
gallon.'|R]) :-
        X + Y =< 3,
        X > 0,
        NEW_Y is X + Y,
        not(member(state(0, NEW_Y), C)),
        path(0, NEW_Y, [state(0, NEW_Y)|C], R).
```

8. WAP to solve the following Monkey and Banana problem: A hungry monkey finds himself in a room in which a bunch of bananas is hanging from the ceiling. The monkey, unfortunately, cannot reach the banana. However, in the room there are also a chair and a stick. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around, carry other things around, reach for the bananas, and wave a stick in the air. What is the best sequence of actions for the monkey to take to acquire lunch?

Program:

```
perform(grasp,
 state(middle, middle, onbox, hasnot),
 state(middle, middle, onbox, has)).
perform(climb,
 state(MP, BP, onfloor, H),
 state(MP, BP, onbox, H)).
perform(push(P1,P2),
 state(P1, P1, onfloor, H),
 state(P2, P2, onfloor, H)).
perform(walk(P1,P2),
 state(P1, BP, onfloor, H),
 state(P2, BP, onfloor, H)).
getfood(state(_,_,_,has)).
getfood(S1):- perform(Act, S1, S2),
 nl, write('In '), write(S1),
 nl, write(' try '), write(Act),
 getfood(S2).
```

9. WAP to solve the following Missionaries and Cannibals problem: In the missionaries and cannibals' problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board. And, in some variations, one of the cannibals has only one arm and cannot row.

Program:

```
        % Main control block and printing
find :-
 path([3,3,left],[0,0,right],[[3,3,left]],_).
output([]) :- nl, nl.
output([[A,B,String]|T]) :-
output(T),
 write(B), write(' ~~ '), write(A), write(': '), write(String), nl.
% Base case
path([A,B,C],[A,B,C],_,MoveList):-
nl,nl,output(MoveList).
% Recursive call to solve the problem
path([A,B,C],[D,E,F],Traversed,Moves) :-
 move([A,B,C],[I,J,K],Out),
 legal([I,J,K]), % Don't use this move unless it's safe.
 not(member([I,J,K],Traversed)),
 path([I,J,K],[D,E,F],[[I,J,K]|Traversed],[ [[I,J,K],[A,B,C],Out] | Moves ]).
% Move commands and descriptions of the move
move([A,B,left],[C,B,right],'One missionary crosses the river') :-
 A > 0, C is A - 1.
move([A,B,left],[C,B,right],'Two missionaries cross the river') :-
 A > 1, C is A - 2.
move([A,B,left],[C,D,right],'One missionary and One cannibal cross
the river') :-
 A > 0, B > 0, C is A - 1, D is B - 1.
move([A,B,left],[A,D,right],'One cannibal crosses the river') :-
 B > 0, D is B - 1.
move([A,B,left],[A,D,right],'Two cannibals cross the river') :-
```

```
 B > 1, D is B - 2.
move([A,B,right],[C,B,left],'One missionary returns from the other
side') :-
 A < 3, C is A + 1.
move([A,B,right],[C,B,left],'Two missionaries return from the other
side') :-
 A < 2, C is A + 2.
move([A,B,right],[C,D,left],'One missionary and One cannibal return
from the other
side') :-
 A < 3, B < 3, C is A + 1, D is B + 1.
move([A,B,right],[A,D,left],'One cannibal returns from the other side')
:-
 B < 3, D is B + 1.
move([A,B,right],[A,D,left],'Two cannibals return from the other side')
:-
 B < 2, D is B + 2.
% Legal move definition where B is missionaries and A is cannibals:
legal([B,A,_]) :-
 (A =< B ; B = 0),
 C is 3-A, D is 3-B,
 (C =< D; D = 0).
```

10. WAP to show Teacher-Student Relationship tree.

Program:

```
studies(charlie, csc135).
studies(olivia, csc135).
studies(jack, csc131).
studies(arthur, csc134).

teaches(kirke, csc135).
teaches(collins, csc131).
teaches(collins, csc171).
teaches(juniper, csc134).

professor(X, Y) :- teaches(X, C), studies(Y, C).
studies(charlie, What).
professor(kirke, Student).
Students.
```

11. Write a Program to Implement Breadth First Search using Python.

Program:

```
  graph = {
 '5' : ['3','7'],
 '3' : ['2', '4'],
 '7' : ['8'],
 '2' : [],
 '4' : ['8'],
 '8' : []
}

visited = []
queue = []

def bfs(visited, graph, node):
 visited.append(node)
 queue.append(node)

 while queue:
  m = queue.pop(0)
  print (m, end = " ")

  for neighbour in graph[m]:
   if neighbour not in visited:
    visited.append(neighbour)
    queue.append(neighbour)

print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

12. Write a Program to Implement Depth First Search using Python.

Program:

```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}

visited = set()
def dfs(visited, graph, node):
  if node not in visited:
    print (node)
    visited.add(node)
    for neighbour in graph[node]:
      dfs(visited, graph, neighbour)

dfs(visited, graph, 'A')
```