Write a program to implement monkey banana problems.

```prolog
on(floor,monkey).
on(floor,chair).
in(room,monkey).
in(room,banana).
in(room,chair).
at(ceiling,banana).
strong(monkey).
grasp(monkey).
climb(Monkey,Chair).
push(Monkey,Chair) :- strong(Monkey).
under(Banana,Chair) :- push(Monkey, Chair).
canreach(Banana, Monkey) :-
      at(Floor,Banana);
      at(Ceiling,Banana),
      under(Banana,Chair),
      climb(Monkey, Chair).
canget(Banana, Monkey) :- canreach(Banana, Monkey), grasp(Monkey).
```

Write a program for river crossing of your choice of problem.

```python
x=['M','L','G','C']
y=[]
print("Before Process")
print("Element in the left side bank",x)
print("Element in the right side bank",y)
while True:
    print(x[1],"",x[2],"",x[3],"")
    i=input("Enter the item :")
    i=i.upper()
    if x[1]==i and x[2]=='G' and x[3]=='C':
        print("Goat will eat cabbage :")
        break
    elif x[2]==i and x[3]!='C':
        y.append(x[2])
        if len(y)==2 and y[0]=='G':
            x[2]=y[0]
            y[0]=y[1]
            y.pop()
    elif x[1]==i and x[2]=='G':
        y.append(x[1])
        x[1]=x[2]
        x[2]=''
        if len(y)==2 and y[0]=='G':
            x[2]=y[0]
            y[0]=y[1]
            y.pop()
    elif x[1]==i and x[2]!='C' and x[2]!='G':
        y.append(x[1])
        y.append('M')
        x[1]=''
        x=[]
        print("Goal is reached")
        break
    if x[2]==i and x[3]=='C':
        y.append(x[2])
        x[2]=x[3]
        x[3]=''
    if x[3]==i:
        print("Lion will eat  goat :")
        break
print("After process")
print("Element in the left side bank",x)
print("Element in the right side bank",y)
```

Write a program for tic tac toe(for defaulters)

```
board = ["-", "-", "-",
         "-", "-", "-",
         "-", "-", "-"]

# Define a function to print the game board
def print_board():
    print(board[0] + " | " + board[1] + " | " + board[2])
    print(board[3] + " | " + board[4] + " | " + board[5])
    print(board[6] + " | " + board[7] + " | " + board[8])

# Define a function to handle a player's turn
def take_turn(player):
    print(player + "'s turn.")
    position = input("Choose a position from 1-9: ")
    while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:
        position = input("Invalid input. Choose a position from 1-9: ")
    position = int(position) - 1
    while board[position] != "-":
        position = int(input("Position already taken. Choose a different
position: ")) - 1
    board[position] = player
    print_board()

# Define a function to check if the game is over
def check_game_over():
    # Check for a win
    if (board[0] == board[1] == board[2] != "-") or \
       (board[3] == board[4] == board[5] != "-") or \
       (board[6] == board[7] == board[8] != "-") or \
       (board[0] == board[3] == board[6] != "-") or \
       (board[1] == board[4] == board[7] != "-") or \
       (board[2] == board[5] == board[8] != "-") or \
       (board[0] == board[4] == board[8] != "-") or \
       (board[2] == board[4] == board[6] != "-"):
        return "win"
    # Check for a tie
    elif "-" not in board:
        return "tie"
    # Game is not over
    else:
        return "play"

# Define the main game loop
def play_game():
    print_board()
    current_player = "X"
    game_over = False
    while not game_over:
        take_turn(current_player)
        game_result = check_game_over()
        if game_result == "win":
            print(current_player + " wins!")
            game_over = True
        elif game_result == "tie":
            print("It's a tie!")
            game_over = True
        else:
            # Switch to the other player
            current_player = "O" if current_player == "X" else "X"

# Start the game
play_game()
```

Design a family tree in prolog assuming your own example.
Write a program which contains three predicates: male, female, parent. Make rules for following family relations:
father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew and niece, cousin.

```
% Define the predicates male and female.
male(john).
male(bob).
male(michael).
male(james).
male(david).
female(jane).
female(lisa).
female(emma).
female(susan).
female(amy).

% Define parent relationships.
parent(john, bob).
parent(john, lisa).
parent(bob, michael).
parent(bob, jane).
parent(lisa, james).
parent(lisa, emma).
parent(jane, susan).
```

```prolog
parent(jane, amy).
parent(michael, david).

% Define rules for various family relations.

% Mother and father relationships.
mother(Mother, Child) :- female(Mother), parent(Mother, Child).
father(Father, Child) :- male(Father), parent(Father, Child).

% Grandparent relationships.
grandfather(Grandfather, Grandchild) :- male(Grandfather), parent(Parent,
Grandchild), parent(Grandfather, Parent).
grandmother(Grandmother, Grandchild) :- female(Grandmother), parent(Parent,
Grandchild), parent(Grandmother, Parent).

% Sibling relationships.
sibling(Sibling1, Sibling2) :- parent(Parent, Sibling1), parent(Parent,
Sibling2), Sibling1 \= Sibling2.

% Brother and sister relationships.
brother(Brother, Sibling) :- male(Brother), sibling(Brother, Sibling).
sister(Sister, Sibling) :- female(Sister), sibling(Sister, Sibling).

% Uncle and aunt relationships.
uncle(Uncle, NieceNephew) :- male(Uncle), sibling(Uncle, Parent),
parent(Parent, NieceNephew).
aunt(Aunt, NieceNephew) :- female(Aunt), sibling(Aunt, Parent),
parent(Parent, NieceNephew).

% Nephew and niece relationships.
nephew(Nephew, UncleAunt) :- male(Nephew), parent(Parent, Nephew),
aunt(UncleAunt, Parent).
niece(Niece, UncleAunt) :- female(Niece), parent(Parent, Niece),
aunt(UncleAunt, Parent).

% Cousin relationships.
cousin(Cousin1, Cousin2) :- parent(Parent1, Cousin1), parent(Parent2,
Cousin2), sibling(Parent1, Parent2), Cousin1 \= Cousin2.
```

Queries:-

```prolog
?- mother(X, james).
% X = lisa.
```

```
?- grandfather(X, amy).
% X = john.

?- sibling(X, susan).
% X = amy.
% X = david.

?- uncle(X, jane).
% X = david.
?- brother(X, amy).
% X = james.

?- sister(X, david).
% X = susan.
```

Illustrate Basic operators in Prolog.
Illustrate Basic operators and comparison operators in Prolog.

```
% Arithmetic Operators
addition(X, Y, Sum) :- Sum is X + Y.
subtraction(X, Y, Difference) :- Difference is X - Y.
multiplication(X, Y, Product) :- Product is X * Y.
division(X, Y, Quotient) :- Y =\= 0, Quotient is X / Y.  % Check for
division by zero

% Comparison Operators
equal(X, Y) :- X =:= Y.
not_equal(X, Y) :- X =\= Y.
greater_than(X, Y) :- X > Y.
less_than(X, Y) :- X < Y.
greater_than_or_equal(X, Y) :- X >= Y.
less_than_or_equal(X, Y) :- X =< Y.

% Logical Operators
logical_and(X, Y) :- X, Y.
logical_or(X, Y) :- X; Y.
logical_not(X) :- \+X.

% Example Usage
% Arithmetic
?- addition(5, 3, Sum).  % Sum = 8
?- subtraction(10, 3, Diff).  % Diff = 7
?- multiplication(4, 5, Prod).  % Prod = 20
?- division(15, 5, Quot).  % Quot = 3
?- division(10, 0, Quot).  % Fails due to division by zero

% Comparison
?- equal(5, 5).  % true
?- not_equal(4, 5).  % true
?- greater_than(8, 3).  % true
?- less_than(4, 9).  % true
?- greater_than_or_equal(7, 7).  % true
?- less_than_or_equal(6, 7).  % true

% Logical
?- logical_and(true, false).  % false
?- logical_or(true, false).  % true
?- logical_not(false).  % true
```

Write a program to solve the Tower of Hanoi problem.

```


def tower_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n - 1, source, target, auxiliary)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n - 1, auxiliary, source, target)

# Example usage:
n = 3  # Number of disks
tower_of_hanoi(n, 'A', 'B', 'C')
```

Demonstrate Uniform cost search in python.

Water Jug Problem

```
Write a program to solve the water jug problem
from collections import deque

def water_jug_puzzle(A, B, C):
    visited = set()
    queue = deque([(0, 0, [])])

    while queue:
        x, y, steps = queue.popleft()

        if x == C or y == C:
            return steps

        visited.add((x, y))

        # Fill jug A
        if x < A:
            new_x = A
            new_y = y
            if (new_x, new_y) not in visited:
                queue.append((new_x, new_y, steps + [f"Fill A"]))

        # Fill jug B
        if y < B:
            new_x = x
            new_y = B
            if (new_x, new_y) not in visited:
                queue.append((new_x, new_y, steps + [f"Fill B"]))

        # Empty jug A
        if x > 0:
            new_x = 0
            new_y = y
            if (new_x, new_y) not in visited:
                queue.append((new_x, new_y, steps + [f"Empty A"]))

        # Empty jug B
        if y > 0:
            new_x = x
            new_y = 0
            if (new_x, new_y) not in visited:
                queue.append((new_x, new_y, steps + [f"Empty B"]))

        # Pour from jug A to jug B
        if x > 0 and y < B:
            pour = min(x, B - y)
            new_x = x - pour
            new_y = y + pour
            if (new_x, new_y) not in visited:
                queue.append((new_x, new_y, steps + [f"Pour A to B"]))

        # Pour from jug B to jug A
        if y > 0 and x < A:
            pour = min(y, A - x)
            new_x = x + pour
            new_y = y - pour
            if (new_x, new_y) not in visited:
                queue.append((new_x, new_y, steps + [f"Pour B to A"]))

    return []

# Example usage:
A = 4  # Jug A capacity
B = 3  # Jug B capacity
C = 2  # Target amount of water
steps = water_jug_puzzle(A, B, C)

if steps:
    print(f"Steps to measure {C} liters of water:")
    for step in steps:
        print(step)
else:
    print(f"It is not possible to measure {C} liters of water.")
```

Write a program to implement Depth Limited Search.

```
#implementing dls
def dls(root , visited , depth , depth_limit , graph):
    if depth > depth_limit :
        return
    visited.add(root)
    print(root , end = ' ')
    for child in graph[root]:
        if child not in visited:
            dls(child , visited , depth+1 , depth_limit , graph)





if __name__ == "__main__":
    #Dictionary to represent the graph
    graph = {
        1:[2,3],
        2:[1,4],
        3:[1,5],
        4:[6,7,8],
        5:[],
        6:[4,9],
        7:[4,10],
        8:[],
        9:[],
        10:[]
    }

    visited = set()
    root=1
    depth = 0
    depth_limit = 3

    dls(root , visited , depth, depth_limit , graph)
```

Write a program to implement Breadth first Search.

```
from queue import Queue
#implementing bfs
def bfs(graph , start_node,goal_node):
    closed_list = set()
```

```python
    opened_list = Queue()
    opened_list.put(start_node)

    while not(opened_list.empty()):

        curr = opened_list.get()
        closed_list.add(curr)
        for child in graph[curr]:
            if child not in closed_list:
                opened_list.put(child)

    print(closed_list)

if __name__ == "__main__":
    #Dictionary to represent the graph
    graph = {
        0:[1,2,3],
        1:[0,2],
        2:[0,1,4],
        3:[0],
        4:[2]
    }
    start=0

    goal = 4
    bfs(graph , start , goal)
```

Write a program to implement DFS

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

if __name__ == "__main__":
    #Dictionary to represent the graph
    graph = {
        0:[1,2,3],
        1:[0,2],
        2:[0,1,4],
        3:[0],
        4:[2]
    }
    start=0

    goal = 4

    visited = set() # Set to keep track of visited nodes.
    dfs(visited, graph, 'A')
```

Write a program to implement a bidirectional search algorithm.

```
from collections import deque

def bidirectional_search(graph, start, goal):
    forward_queue = deque([start])
    forward_visited = set([start])

    backward_queue = deque([goal])
    backward_visited = set([goal])

    while forward_queue and backward_queue:
        # Forward search
        node = forward_queue.popleft()
        print("Forward exploring:", node)

        if node in backward_visited:
            print("Bidirectional search met at:", node)
            return True

        for neighbor in graph[node]:
            if neighbor not in forward_visited:
                forward_visited.add(neighbor)
                forward_queue.append(neighbor)

        # Backward search
        node = backward_queue.popleft()
        print("Backward exploring:", node)

        if node in forward_visited:
            print("Bidirectional search met at:", node)
            return True
```

```python
        for neighbor in graph[node]:
            if neighbor not in backward_visited:
                backward_visited.add(neighbor)
                backward_queue.append(neighbor)

    return False



result = bidirectional_search(graph, start_node, goal_node)

if result:
    print("Bidirectional search successful.")
else:
    print("Bidirectional search did not meet in the middle.")
```

Design the following in prolog
1.burger is a food
2.sandwich is a food
3.pizza is a food
4.sandwich is a lunch
5.pizza is a dinner
6.Every food is a meal OR Anything is a meal if it is a food.
7.Is pizza a food?
8.Which food is meal and lunch?
9.Is sandwich a dinner?

```prolog
% 1. burger is a food
% 2. sandwich is a food
% 3. pizza is a food
food(burger).
food(sandwich).
food(pizza).

% 4. sandwich is a lunch
lunch(sandwich).
```

```
% 5. pizza is a dinner
dinner(pizza).

% 6. Every food is a meal OR Anything is a meal if it is a food.
meal(X) :- food(X) , lunch(X) , dinner(X).


% 7. Is pizza a food?
is_food(pizza) :- food(pizza).

% 8. Which food is meal and lunch?
food_is_meal_and_lunch(X) :- food(X), meal(X), lunch(X).

% 9. Is sandwich a dinner?
is_dinner(sandwich) :- dinner(sandwich).
```

Write a database with European countries specifying the area, the population and the neighbors of the individual countries:

```
% Predicates for country area
area(Austria, 83858).
area(France, 547030).
area(Germany, 357021).
area(Italy, 301230).
area(Liechtenstein, 160).
area(Spain, 504851).
area(Switzerland, 41290).
area(United_kingdom, 244820).


% Predicates for country population
population(Austria, 8169929).
population(France, 63182000).
population(Germany, 83251851).
population(Italy, 59530464).
population(Liechtenstein, 32842).
population(Spain, 47059533).
population(Switzerland, 7507000).
population(United_kingdom, 61100835).
```

```
% Predicates for country neighbors
neighbor(Austria, Switzerland).
neighbor(France, Switzerland).
neighbor(France, Germany).
neighbor(France, Spain).
neighbor(Germany, Switzerland).
neighbor(italy, switzerland).
neighbor(Liechtenstein, Switzerland).


% Example queries:
% - To find the name of a specific country, e.g., Germany:
?- country_name(Germany).

% - To find the size of a specific country, e.g., Germany:
?- country_size(Germany, Size).

% - To find the population of a specific country, e.g., France:
?- country_population(France, Population).

% - To find the neighbor of a specific country, e.g., Germany:
?- country_neighbor(Germany, Neighbor).
```

Enhance the database with deduction rules, e.g.

```
% Predicate for population density
population_density(Country, Density) :-
    population(Country, Population),

    area(Country, Area),
    Density is Population / Area.
```

Queries:-

```
?- population_density(Austria, Density).
% Density will be unified with the population density of Austria.

?- population_density(France, Density).
% Density will be unified with the population density of France.
```

```prolog
?- population_density(Germany, Density).
% Density will be unified with the population density of Germany.

?- population_density(Italy, Density).
% Density will be unified with the population density of Italy.

% ... Query for population density of other countries as needed
```