Prolog Problems:

1. *Problem 1: Ancestor Relation*

Write a Prolog program to define an ancestor relation. If A is the parent of B and B is the parent of C, then A is an ancestor of C. Test it with queries like `ancestor(A, C).`

given relations can be explained as follows:

```
parent(sanjay, rahul).
parent(rahul, priya).
parent(rahul, amit).
parent(priya, diya).
parent(diya, arjun).
ancestor(A, C) :-
    parent(A, C).  % A is a direct parent of C
ancestor(A, C) :-
    parent(A, X),  % A is a parent of X
    ancestor(X, C).  % X is an ancestor of C
% Queries
?- ancestor(sanjay, priya).   % Is sanjay an ancestor of priya?
?- ancestor(sanjay, amit).    % Is sanjay an ancestor of amit?
?- ancestor(sanjay, arjun).   % Is sanjay an ancestor of arjun?
?- ancestor(sanjay, rahul).   % Is sanjay an ancestor of rahul?

true.  % sanjay is an ancestor of priya
true.  % sanjay is an ancestor of amit
true.  % sanjay is an ancestor of arjun
false. % sanjay is not an ancestor of rahul
```

2. *Problem 2: Sibling Relation*

Write a Prolog program to define a sibling relation. Two people are siblings if they have at least one parent in common. Test it with queries like `sibling(X, Y).`

```
% Facts
parent(ram, shashi).
parent(ram, sunita).
parent(suresh, shashi).
parent(suresh, sunita).
parent(mohan, ajay).
parent(mohan, amita).
parent(sunita, ajay).
parent(sunita, amita).

% Rules
sibling(X, Y) :-
    parent(Z, X),
    parent(Z, Y),
    X \= Y.
```

Queries –

```
?- sibling(shashi, sunita).
true.
?- sibling(shashi, ajay).
false.
?- sibling(ajay, amita).
true.
?- sibling(ram, sunita).
```

true.

3. *Problem 3: Fibonacci Sequence*

Write a Prolog program to generate the Fibonacci sequence up to a given number N. The predicate `fibonacci(N, Sequence)` should give the Fibonacci sequence up to N in `Sequence`.

```
fibonacci(0, [0]).

fibonacci(1, [0, 1]).

fibonacci(N, Sequence) :-

    N > 1,

    fibonacci(N, 0, 1, Sequence).


fibonacci(2, A, B, [A, B]).

fibonacci(N, A, B, [B | Rest]) :-

    N > 2,

    Next is A + B,

    N1 is N - 1,

    fibonacci(N1, B, Next, Rest).
```

4. *Problem 4: Tower of Hanoi*

Write a Prolog program to solve the Tower of Hanoi problem. The predicate `hanoi(N)` should give the steps to solve a Tower of Hanoi puzzle with N disks.

```
hanoi(N) :- hanoi(N, left, middle, right).

hanoi(0, _, _, _) :- !.

hanoi(N, A, B, C) :-

    M is N - 1,

    hanoi(M, A, C, B),

    move(A, C),
```

```
    hanoi(M, B, A, C).
move(From, To) :-
    write('Move a disk from '), write(From),
    write(' to '), write(To), nl.
```

Python Problems:

1. *Problem 1: String Reversal*

Write a Python function `reverse_string(input_str)` that takes a string as input and returns the string in reverse order.

```
def reverse(input_str):
    return input_str[::-1]
input_str = input("Enter a string: ")
reversed = reverse(input_str)
print("Reversed string:", reversed)
```

2. *Problem 2: Caesar Cipher*

Write a Python function `caesar_cipher(input_str, shift)` that implements a simple Caesar cipher, which is a type of substitution cipher in which each letter in the plaintext is 'shifted' a certain number of places down the alphabet.

```
def caesar_cipher(input_str, shift):
    result = ""
    for char in input_str:
        if char.isalpha():
            shifted_char = chr(ord(char) + shift)
            result += shifted_char
        else:
            result += char
```

```
    return result
plaintext = input("Enter the plaintext: ")
shift = int(input("Enter the shift: "))
ciphertext = caesar_cipher(plaintext, shift)
print("Ciphertext:", ciphertext)
```

3. *Problem 3: Find the Duplicate*

Write a Python function `find_duplicate(lst)` that takes a list of integers where each integer appears once except for one which appears twice. The function should find and return the integer that appears twice.

```
def find_duplicates(lst):
    duplicates = []
    for i in lst:
        if lst.count(i) > 1 and i not in duplicates:
            duplicates.append(i)
    return duplicates
input_list = input("Enter the list (comma-separated values): ")
lst = list(map(int, input_list.split(",")))
duplicates = find_duplicates(lst)
print("Duplicate values:", duplicates)
```

4. *Problem 4: Tic Tac Toe Checker*

Write a Python function `tic_tac_toe(board)` that checks a Tic Tac Toe board represented as a list of lists and returns whether 'X', 'O', or neither player has won the game yet.

```
def tic_tac_toe(board):
    # Check rows
    for row in board:
        if all(elem == 'X' for elem in row):
```

```
            return 'X'
        elif all(elem == 'O' for elem in row):
            return 'O'
    # Check columns
    for col in range(len(board[0])):
        if all(row[col] == 'X' for row in board):
            return 'X'
        elif all(row[col] == 'O' for row in board):
            return 'O'
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2]:
        return board[0][0]
    elif board[0][2] == board[1][1] == board[2][0]:
        return board[0][2]
    return 'Neither'
# Example board
board = [['X', 'O', 'O'],
         ['X', 'X', 'O'],
         ['O', 'X', 'O']]
winner = tic_tac_toe(board)
print(winner)  # Output: 'O'
```

1. *Problem 5: Breadth-First Search (BFS)*

Write a Python function `bfs(graph, start_node)` that performs a breadth-first search on a given graph, starting from a given node. The graph will be represented as a dictionary where the keys are node identifiers and the values are lists of neighboring nodes.

```
from collections import deque
```

```
def bfs(graph,start):
    visited=set()
    queue=deque([start])
    while queue:
        node=queue.popleft()
        if node not in visited:
            print(node)
            visited.add(node)
            for neighbour in graph[node]:
                queue.append(neighbour)
graph={'0':['1','2','3'],'1':['0','2'],'2':['0','4'],'3':['0'],'4':['2']}
bfs(graph,'0')
```

2. *Problem 6: Depth-First Search (DFS)*

Write a Python function `dfs(graph, start_node)` that performs a depth-first search on a given graph, starting from a given node. As with the previous problem, the graph will be represented as a dictionary.

```
graph={
    'A':['B','C','D'],'B':['E'],'C':['D','E'],'D':[],'E':[]
}
visited=set()
def dfs(visited,graph,root):
    if root not in visited:
        print(root)
        visited.add(root)
        for neighbour in graph[root]:
            dfs(visited,graph,neighbour)
dfs(visited,graph,'A')
```

3. *Problem 7: Best-First Search*

Write a Python function `best_first_search(graph, start_node, goal_node, heuristic)` that performs a best-first search on a given graph, starting from a given node and aiming for a goal node. The heuristic function will be provided and takes two nodes as input, returning an estimate of the cost to reach the goal from the first node.

```
def greedy(graph, heuristics, start_node):
  if(graph is None or heuristics is None or start_node is None):
    return 'Failiure', 0
  paths = {start_node:0}
  current_path = start_node
  selected_key = ''
  selected_value = 0

  while(True):
    if(heuristics[current_path[-1]] == 0):
      return current_path, paths[current_path]
    min_heuristic_value = 10000
    for key, value in graph.items():
      if(key[0] == current_path[-1] and heuristics[key[-1]] <= 
min_heuristic_value):
        selected_key = key
        selected_value = value
        min_heuristic_value = heuristics[key[-1]]
    paths[current_path + selected_key[-1]] = paths[current_path] + 
selected_value
    del paths[current_path]
    current_path += selected_key[-1]
```

```
graph = {'SA':3, 'SB':2, 'AC':4, 'AD':1, 'BE':3, 'BF':1, 'EH':5, 'FI':2, 'FG':3} # SBFG

heuristics = {'S':13, 'A':12, 'B':4, 'C':7, 'D':3, 'E':8, 'F':2, 'G':0, 'H':4, 'I':9}

start_node = 'S'

path, cost=greedy(graph,heuristics,start_node)

print(path, '\t', cost)


def heuristic(node1, node2):

    # Example heuristic function

    return abs(ord(node1) - ord(node2))

found = best_first_search(graph, start_node, goal_node, heuristic)

print(found)  # Output: True
```

4. Problem 8: A Star Search

Write a Python function `a_star_search(graph, start_node, goal_node, heuristic)` that performs an A* search on a given graph, starting from a given node and aiming for a goal node. The heuristic function will be provided, just like in the best-first search problem.

```
def a_star(graph, heuristics, start_node):

  if(graph is None or heuristics is None or start_node is None):

    return 'Failiure', 0

  paths = {start_node:0}

  current_path = start_node


  while(True):

    if(heuristics[current_path[-1]] == 0):

      return current_path, paths[current_path]
```

```
    for key, value in graph.items():
      if(key[0] == current_path[-1]):
        paths[current_path + key[-1]] = paths[current_path] + value
    del paths[current_path]
    min_cost = 10000
    for key, value in paths.items():
      if(value + heuristics[key[-1]] < min_cost):
        min_cost = value + heuristics[key[-1]]
        current_path = key
graph = {'AB':2, 'AE':3, 'BC':1, 'BG':9, 'ED':6, 'DG':1} # expected result = AEDG
heuristics = {'A':11, 'B':6, 'C':99, 'D':1, 'E':7, 'G':0}
start_node = 'A'
path, cost= a_star(graph,heuristics,start_node)
print(path, '\t', cost)
```

5. *Problem 9: Implement Dijkstra's Algorithm*

Write a Python function `dijkstra(graph, start_node, end_node)` that finds the shortest path between two nodes in a graph using Dijkstra's algorithm. The graph will be represented as a dictionary of dictionaries: the outer dictionary's keys are node identifiers, and the inner dictionaries' keys are neighboring nodes with the associated values being the distance to that neighbor.

```
import heapq
def dijkstra(graph, start_node, end_node):
    distances = {node: float('inf') for node in graph}
    distances[start_node] = 0
    previous = {node: None for node in graph}
    priority_queue = [(0, start_node)]
```

```
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        if current_node == end_node:
            return construct_path(previous, end_node)
        if current_distance > distances[current_node]:
            continue
        for neighbor, distance in graph[current_node].items():
            new_distance = current_distance + distance
            if new_distance < distances[neighbor]:
                distances[neighbor] = new_distance
                previous[neighbor] = current_node
                heapq.heappush(priority_queue, (new_distance, neighbor))
    return None
def construct_path(previous, end_node):
    path = []
    current_node = end_node
    while current_node is not None:
        path.append(current_node)
        current_node = previous[current_node]
    path.reverse()
    return path
# Example graph
graph = {
    'A': {'B': 4, 'C': 2},
    'B': {'A': 4, 'C': 1, 'D': 5},
    'C': {'A': 2, 'B': 1, 'D': 8},
```

```
    'D': {'B': 5, 'C': 8}
}
start_node = 'A'
end_node = 'D'
path = dijkstra(graph, start_node, end_node)
print(path)  # Output: ['A', 'C', 'B', 'D']
```