List of programs:-

1 Write the python program to solve 8-Puzzle problem
2 Write the python program to solve 8-Queen problem
3 Write the python program for Water Jug Problem
4 Write the python program for Cript-Arithmetic problem
5 Write the python program for Missionaries Cannibal problem
6 Write the python program for Vacuum Cleaner problem
7 Write the python program to implement BFS.
8 Write the python program to implement DFS.
9 Write the python to implement Travelling Salesman Problem
10 Write the python program to implement A* algorithm
11 Write the python program for Map Coloring to implement CSP.
12 Write the python program for Tic Tac Toe game
13 Write the python program to implement Minimax algorithm for gaming
14 Write the python program to implement Apha & Beta pruning algorithm for gaming
15 Write the python program to implement Decision Tree
16 Write the python program to implement Feed forward neural Network
17 Write a Prolog Program to Sum the Integers from 1 to n.
18 Write a Prolog Program for A DB WITH NAME, DOB.
19 Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.
20 Write a Prolog Program for PLANETS DB.
21 Write a Prolog Program to implement Towers of Hanoi.
22 Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.
23 Write the prolog program to implement family tree.
24 Write a Prolog Program to suggest Dieting System based on Disease.
25 Write a Prolog program to implement Monkey Banana Problem
26 Write a Prolog Program for fruit and its color using Back Tracking.
27 Write a Prolog Program to implement Best First Search algorithm
28 Write the prolog program for Medical Diagnosis
29 Write a Prolog Program for forward Chaining. Incorporate required queries.
30 Write a Prolog Program for backward Chaining. Incorporate required queries.
31 Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc.
----

1 Write the python program to solve 8-Puzzle problem

```
from collections import deque
def bfs(start, goal):
    queue = deque([start])
    visited = set()
    while queue:
        state = queue.popleft()
        if state == goal:
            print("Reached goal:", state)
            return
```

```
        visited.add(tuple(state))

        for i in range(len(state)-1):
            new = state[:]
            new[i], new[i+1] = new[i+1], new[i]
            if tuple(new) not in visited:
                queue.append(new)
bfs([1, 2, 3, 4, 5, 6, 7, 0, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0])
```

2 Write the python program to solve 8-Queen problem

```
def solve(n=8, y=0, board=[]):
    if y == n: return [board]
    solutions = []
    for x in range(n):
        if all(x != c and abs(x - c) != y - r for r, c in enumerate(board)):
            solutions += solve(n, y + 1, board + [x])
    return solutions

def print_solutions():
    for sol in solve():
        for row in sol:
            print(" ".join(["_" if i != row else "Q" for i in range(8)]))
        print("\n")

print_solutions()
```

3 Write the python program for Water Jug Problem

```
def solve(jug1, jug2, target):
    q, visited, path = [(0, 0)], set(), []
    while q:
        a, b = q.pop(0)
        if (a, b) in visited:
            continue
        visited.add((a, b))
        path.append((a, b))
        if a == target or b == target:
            return path
        q += [
            (jug1, b), (a, jug2), (0, b), (a, 0),
            (a - min(a, jug2 - b), b + min(a, jug2 - b)),
            (a + min(b, jug1 - a), b - min(b, jug1 - a))
        ]
    return "No Solution"
```

```
def print_solution(jug1, jug2, target):
    solution = solve(jug1, jug2, target)
    if solution == "No Solution":
        print(solution)
    else:
        for step in solution:
            print(step)

print_solution(4, 3, 2)
```

4 Write the python program for Cript-Arithmetic problem

```
import itertools
def solve():
    for p in itertools.permutations(range(10), 8):
        s, e, n, d, m, o, r, y = p
        if s == 0 or m == 0: continue
        send = s*1000 + e*100 + n*10 + d
        more = m*1000 + o*100 + r*10 + e
        money = m*10000 + o*1000 + n*100 + e*10 + y
        if send + more == money:
            print(f"SEND={send}, MORE={more}, MONEY={money}")
solve()
```

5 Write the python program for Missionaries Cannibal problem

```
def valid(state):
    m1, c1, boat, m2, c2 = state
    return (0 <= m1 <= 3 and 0 <= c1 <= 3 and 0 <= m2 <= 3 and 0 <= c2 <= 3 and
            (m1 == 0 or m1 >= c1) and (m2 == 0 or m2 >= c2))

def successors(state):
    m1, c1, boat, m2, c2 = state
    moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
    next_states = []
    for m, c in moves:
        if boat:
            next_state = (m1 - m, c1 - c, 0, m2 + m, c2 + c)
        else:
            next_state = (m1 + m, c1 + c, 1, m2 - m, c2 - c)
        if valid(next_state):
            next_states.append(next_state)
    return next_states

def solve():
```

```
    start, goal = (3, 3, 1, 0, 0), (0, 0, 0, 3, 3)
    queue, visited = [(start, [])], set()
    while queue:
        state, path = queue.pop(0)
        if state == goal:
            return path + [state]
        if state not in visited:
            visited.add(state)
            for next_state in successors(state):
                queue.append((next_state, path + [state]))
    return "No solution"

solution = solve()
if solution == "No solution":
    print(solution)
else:
    print("Steps to solve the Missionaries and Cannibals problem:")
    for i, step in enumerate(solution):
        m1, c1, boat, m2, c2 = step
        print(f"Step {i + 1}: Left -> M:{m1} C:{c1} | Boat: {'Left' if boat else 'Right'} | Right -> M:{m2} C:{c2}")s
```

6 Write the python program for Vacuum Cleaner problem

```
def vacuum(env):
    for room in env:
        if env[room] == 'dirty':
            print(f"Cleaning {room}")
            env[room] = 'clean'
        else:
            print(f"{room} already clean")
env = {'A': 'dirty', 'B': 'clean'}
vacuum(env)
```

7 Write the python program to implement BFS.

```
def bfs(graph, start):
    visited, queue = set(), [start]
```

```
    while queue:
        node = queue.pop(0)
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(sorted(graph[node] - visited))
graph = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F', 'G'},
    'D': {'B'},
    'E': {'B', 'H'},
    'F': {'C'},
    'G': {'C'},
    'H': {'E'}
}

bfs(graph, 'A')
```

8 Write the python program to implement DFS.

```
def dfs(graph, node, visited):
    if node not in visited:
        print(node, end=" ")    # Print the visited node
        visited.add(node)    # Mark node as visited
        for neighbor in graph.get(node, []):    # Visit all neighbors
            dfs(graph, neighbor, visited)

# Example graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

# Run DFS
visited = set()
print("DFS Traversal:")
dfs(graph, 'A', visited)
```

9 Write the python to implement Travelling Salesman Problem

```python
def td(g, p):
    d = sum(g[p[i]][p[i+1]] for i in range(len(p)-1))
    return d + g[p[-1]][p[0]]    # Return to start

# Generate all paths (backtracking)
def gt(c, s, p, v, bp, bd, g):
    if len(p) == len(c):
        d = td(g, p)
        if d < bd[0]: bd[0], bp[:] = d, p[:]
        return
    for city in c:
        if city not in v:
            v.add(city); p.append(city)
            gt(c, s, p, v, bp, bd, g)
            p.pop(); v.remove(city)

# TSP function
def tsp(g):
    c, s = list(g.keys()), list(g.keys())[0]
    bp, bd = [], [float('inf')]
    gt(c, s, [s], {s}, bp, bd, g)
    return bp, bd[0]

# Graph (Adjacency matrix)
g = {
    'A': {'A': 0, 'B': 10, 'C': 15, 'D': 20},
    'B': {'A': 10, 'B': 0, 'C': 35, 'D': 25},
    'C': {'A': 15, 'B': 35, 'C': 0, 'D': 30},
    'D': {'A': 20, 'B': 25, 'C': 30, 'D': 0}
}

# Run TSP
bp, bd = tsp(g)
print("Best Path:", bp)
print("Min Distance:", bd)
```

10 Write the python program to implement A* algorithm

```python
    class N:
    def __init__(s, n, g=0, h=0):
        s.n, s.g, s.h, s.f, s.p = n, g, h, g + h, None

def a_star(g, h, s, e):
    o, c = {s: N(s, 0, h[s])}, {}
```

```
    while o:
        cur = min(o.values(), key=lambda x: x.f)
        del o[cur.n]; c[cur.n] = cur

        if cur.n == e:
            p = []
            while cur: p.append(cur.n); cur = cur.p
            return p[::-1]

        for nb, cost in g[cur.n].items():
            if nb in c: continue
            g_new = cur.g + cost
            if nb not in o or g_new < o[nb].g:
                o[nb] = N(nb, g_new, h[nb])
                o[nb].p = cur

    return None

g = {'A': {'B': 4, 'C': 3}, 'B': {'D': 5, 'E': 12}, 'C': {'E': 10}, 'D': {'F': 8}, 'E': {'F': 6}, 'F': {}}
h = {'A': 14, 'B': 12, 'C': 11, 'D': 6, 'E': 4, 'F': 0}

print("Shortest Path:", a_star(g, h, 'A', 'F'))
```

11 Write the python program for Map Coloring to implement CSP.

```
neighbors = {
    "A": ["B", "C"],
    "B": ["A", "C", "D"],
    "C": ["A", "B", "D", "E"],
    "D": ["B", "C", "E"],
    "E": ["C", "D"]
}

# Available colors
colors = ["Red", "Green", "Blue"]

# Dictionary to store the assigned colors
color_assignment = {}

def is_valid(region, color):
    """Check if the color assignment is valid for the given region."""
    for neighbor in neighbors.get(region, []):
```

```
            if neighbor in color_assignment and color_assignment[neighbor] == color:
                return False
        return True

def solve(region_list):
        """Backtracking function to assign colors."""
        if not region_list:    # If all regions are assigned colors, return True
            return True

        region = region_list[0]

        for color in colors:
            if is_valid(region, color):
                color_assignment[region] = color    # Assign color

                if solve(region_list[1:]):    # Recur for the remaining regions
                    return True

                del color_assignment[region]    # Backtrack if assignment fails

        return False    # No valid assignment found

# Start solving
if solve(list(neighbors.keys())):
        print("Color Assignment:", color_assignment)
else:
        print("No solution found.")
```

12 Write the python program for Tic Tac Toe game

```
board = [' ']*9

def show():
        for i in range(0, 9, 3):
            print(board[i] + '|' + board[i+1] + '|' + board[i+2])
            if i < 6: print('-+-+-')

def win(p):
        combos = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]
        return any(board[a]==board[b]==board[c]==p for a,b,c in combos)

turn = 'X'
```

```
for _ in range(9):
    show()
    move = int(input(f"{turn}'s move (1-9): ")) - 1
    if board[move] == ' ':
        board[move] = turn
        if win(turn):
            show()
            print(f'{turn} wins!')
            break
        turn = 'O' if turn == 'X' else 'X'
    else:
        print("Invalid move.")
else:
    show()
    print("Draw!")
```

13 Write the python program to implement Minimax algorithm for gaming

```
def eval_bd(bd):
    for r in range(3):
        if bd[r][0] == bd[r][1] == bd[r][2] != '_': return 10 if bd[r][0] == 'X' else -10
    for c in range(3):
        if bd[0][c] == bd[1][c] == bd[2][c] != '_': return 10 if bd[0][c] == 'X' else -10
    if bd[0][0] == bd[1][1] == bd[2][2] != '_' or bd[0][2] == bd[1][1] == bd[2][0] != '_':
        return 10 if bd[1][1] == 'X' else -10
    return 0

def minimax(bd, is_max):
    score = eval_bd(bd)
    if score: return score
    if not any('_' in row for row in bd): return 0

    best = -1000 if is_max else 1000
    for i in range(3):
        for j in range(3):
            if bd[i][j] == '_':
                bd[i][j] = 'X' if is_max else 'O'
                best = max(best, minimax(bd, not is_max)) if is_max else min(best,
minimax(bd, not is_max))
                bd[i][j] = '_'
    return best
```

```
def best_move(bd):
    move, best_val = (-1, -1), -1000
    for i in range(3):
        for j in range(3):
            if bd[i][j] == '_':
                bd[i][j] = 'X'
                val = minimax(bd, False)
                bd[i][j] = '_'
                if val > best_val: move, best_val = (i, j), val
    return move

grid = [['X', 'O', 'X'], ['O', 'O', 'X'], ['_', '_', '_']]
print("Best Move:", best_move(grid))
```

14 Write the python program to implement Apha & Beta pruning algorithm for gaming

```
def alpha_beta(node, d, a, b, max_p):
    if d == 0 or isinstance(node, int): return node
    val = -999999 if max_p else 999999
    for c in node:
        v = alpha_beta(c, d - 1, a, b, not max_p)
        val = max(val, v) if max_p else min(val, v)
        a, b = (max(a, val), b) if max_p else (a, min(b, val))
        if b <= a: break
    return val

tree = [[3, 5, 6], [2, 9, -1], [4, 7, 8]]
print("Best outcome:", alpha_beta(tree, 3, -999999, 999999, True))
```

15 Write the python program to implement Decision Tree

```
class Node:
    def __init__(self, question=None, left=None, right=None, label=None):
        self.question = question
        self.left = left
        self.right = right
        self.label = label

def build_tree():
    return Node("Is it raining?",
                Node("Do you have an umbrella?",
                     Node(label="Go outside"),
                     Node(label="Stay inside")),
                Node(label="Go outside"))
```

```
def classify(node):
    while node.label is None:
        ans = input(node.question + " (yes/no): ").strip().lower()
        node = node.left if ans == "yes" else node.right
    return node.label

tree = build_tree()
print("Decision:", classify(tree))
```

16 Write the python program to implement Feed forward neural Network

```
import random, math

def sigmoid(x): return 1 / (1 + math.exp(-x))
def d_sigmoid(x): return x * (1 - x)

def init(n, h, o):
    return [[random.uniform(-1, 1) for _ in range(h)] for _ in range(n)], [[random.uniform(-1, 1)
for _ in range(o)] for _ in range(h)], [random.uniform(-1, 1) for _ in range(h)], [random.uniform(-1,
1) for _ in range(o)]

def forward(inp, w1, w2, b1, b2):
    h = [sigmoid(sum(i * w + b for i, w, b in zip(inp, ws, b1))) for ws in zip(*w1)]
    o = [sigmoid(sum(h[i] * w + b for i, w, b in zip(range(len(h)), ws, b2))) for ws in zip(*w2)]
    return h, o

def train(data, tar, e=1000, lr=0.5):
    w1, w2, b1, b2 = init(len(data[0]), 2, len(tar[0]))
    for _ in range(e):
        for i, inp in enumerate(data):
            h, o = forward(inp, w1, w2, b1, b2)
            d_o = [(tar[i][j] - o[j]) * d_sigmoid(o[j]) for j in range(len(o))]
            d_h = [sum(w2[k][j] * d_o[j] for j in range(len(o))) * d_sigmoid(h[k]) for k in
range(len(h))]
            for j in range(len(o)): b2[j] += lr * d_o[j]
            for k in range(len(h)): b1[k] += lr * d_h[k]
            for k in range(len(h)): w2[k] = [w + lr * h[k] * d_o[j] for j, w in enumerate(w2[k])]
            for j in range(len(inp)): w1[j] = [w + lr * inp[j] * d_h[k] for k, w in
enumerate(w1[j])]
    return w1, w2, b1, b2

X, Y = [[0,0], [0,1], [1,0], [1,1]], [[0], [1], [1], [0]]
w1, w2, b1, b2 = train(X, Y)
for x in X: print(f"Input: {x}, Output: {forward(x, w1, w2, b1, b2)[1]}")
```

----

17 Write a Prolog Program to Sum the Integers from 1 to n.

```
sum of integers
sum(0, 0).
sum(N, S) :-
	N > 0,
	N1 is N - 1,
	sum(N1, S1),
	S is N + S1.


#sum(5, Result).
```

18 Write a Prolog Program for A DB WITH NAME, DOB.

```
NAME, DOB.
dob(john, '1995-06-15').
dob(alice, '2000-12-01').
dob(bob, '1988-03-23').
dob(eve, '1992-07-19').
find_dob(Name, DOB) :- dob(Name, DOB).

#find_dob(john, DOB).
```

19 Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.

```
teaches(mr_smith, math, 101).
teaches(ms_jones, physics, 102).
student(john, math, 101).
student(alice, physics, 102).
find_teacher(Student, Teacher) :-
	student(Student, Subject, Code),
	teaches(Teacher, Subject, Code).


#find_teacher(john, Teacher).
```

20 Write a Prolog Program for PLANETS DB.

```
planet(mercury, terrestrial, 57).
planet(venus, terrestrial, 108).
```

```
planet(uranus, ice_giant, 2871).
planet(neptune, ice_giant, 4495).
find_planet_info(Name, Type, Distance) :- planet(Name, Type, Distance).


#find_planet_info(mercury, Type, Distance).
```

21 Write a Prolog Program to implement Towers of Hanoi.

```
hanoi(1, Source, Target, _) :-
    write('Move disk 1 from '), write(Source), write(' to '), write(Target), nl.
hanoi(N, Source, Target, Auxiliary) :-
    N > 1,
    N1 is N - 1,
    hanoi(N1, Source, Auxiliary, Target),
    write('Move disk '), write(N), write(' from '), write(Source), write(' to '), write(Target), nl,
    hanoi(N1, Auxiliary, Target, Source).


#hanoi(3, 'A', 'C', 'B').
```

22 Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.

```
can_fly(sparrow).
can_fly(eagle).
cannot_fly(penguin).
cannot_fly(kiwi).
bird_flight(Bird, 'can fly') :- can_fly(Bird).
bird_flight(Bird, 'cannot fly') :- cannot_fly(Bird).


#bird_flight(sparrow, Result).
```

23 Write the prolog program to implement family tree.

```
parent(john, mary).
parent(john, mike).
parent(susan, mary).
parent(susan, mike).
```

```
parent(mary, alice).
parent(mary, bob).
parent(mike, charlie).
father(X, Y) :- parent(X, Y), male(X).
mother(X, Y) :- parent(X, Y), female(X).
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
grandchild(X, Y) :- grandparent(Y, X).
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
male(john).
male(mike).
male(bob).
male(charlie).
female(susan).
female(mary).
female(alice).


#sibling(mary, Sibling).
#grandparent(Grandfather, alice), male(Grandfather).
```

24 Write a Prolog Program to suggest Dieting System based on Disease.

```
% Facts: disease and corresponding diet
diet(diabetes, 'Low sugar, High fiber diet').
diet(hypertension, 'Low salt, Low fat diet').
diet(obesity, 'High protein, Low carb diet').
diet(anemia, 'Iron-rich diet with leafy greens').
diet(ulcer, 'Bland diet, avoid spicy food').

% Rule to suggest diet
suggest_diet(Disease) :-
    diet(Disease, Diet),
    format('Suggested diet for ~w: ~w~n', [Disease, Diet]).

#suggest_diet(diabetes).
```

25 Write a Prolog program to implement Monkey Banana Problem

```
% Initial state: (MonkeyPos, ChairPos, HasBanana)
```

```
state(at_door, on_floor, no).
state(at_window, on_floor, no).
state(at_banana, on_floor, no).
state(at_banana, on_chair, yes).    % Goal state

% Actions
move(state(M, on_floor, no), walk(M, NewM), state(NewM, on_floor, no)).
move(state(M, on_floor, no), push_chair(M, NewM), state(NewM, on_floor, no)).
move(state(M, on_floor, no), climb_chair, state(M, on_chair, no)).
move(state(M, on_chair, no), grab_banana, state(M, on_chair, yes)).

% Solve the problem
solve(State, []) :- State = state(_, _, yes).    % Goal state
solve(State, [Action | Actions]) :-
    move(State, Action, NewState),
    solve(NewState, Actions).

% Example Query:
% ?- solve(state(at_door, on_floor, no), Actions).
% Output: Actions = [walk(at_door, at_banana), climb_chair, grab_banana].
```

26 Write a Prolog Program for fruit and its color using Back Tracking.

```
% Facts: Fruit and its corresponding color
fruit_color(apple, red).
fruit_color(banana, yellow).
fruit_color(grape, purple).
fruit_color(orange, orange).
fruit_color(lemon, yellow).
fruit_color(blueberry, blue).
fruit_color(strawberry, red).
fruit_color(kiwi, green).

% Query Examples:
% Find all fruits with a specific color
% ?- fruit_color(Fruit, red).

% Find the color of a specific fruit
% ?- fruit_color(apple, Color).
```

27 Write a Prolog Program to implement Best First Search algorithm

```prolog
% Define the graph with heuristic values
edge(a, b, 4).
edge(a, c, 3).
edge(b, d, 5).
edge(b, e, 12).
edge(c, f, 10).
edge(c, g, 8).
edge(e, h, 7).
edge(f, i, 6).
edge(g, j, 9).

% Define heuristic values for nodes
heuristic(a, 7).
heuristic(b, 6).
heuristic(c, 4).
heuristic(d, 5).
heuristic(e, 3).
heuristic(f, 2).
heuristic(g, 6).
heuristic(h, 5).
heuristic(i, 1).
heuristic(j, 4).

% Best First Search Algorithm
best_first_search(Start, Goal, Path) :-
    best_first([[Start]], Goal, Path).

best_first([[Goal | Path] | _], Goal, [Goal | Path]).
best_first([CurrentPath | OtherPaths], Goal, Solution) :-
    CurrentPath = [CurrentNode | _],
    findall([Next, CurrentNode | CurrentPath],
        (edge(CurrentNode, Next, _), \+ member(Next, CurrentPath)),
        NewPaths),
    sort_by_heuristic(NewPaths, SortedPaths),
    append(SortedPaths, OtherPaths, UpdatedQueue),
    best_first(UpdatedQueue, Goal, Solution).

% Sorting paths based on heuristic values
sort_by_heuristic(Paths, SortedPaths) :-
    map_list_to_pairs(evaluate_path, Paths, Paired),
    keysort(Paired, SortedPaired),
    pairs_values(SortedPaired, SortedPaths).

evaluate_path([Node | _], H) :-
```

```
    heuristic(Node, H).
#best_first_search_ordered(a, i, Path).
```

28 Write the prolog program for Medical Diagnosis

```
% Symptoms and diagnosis rules
symptom(john, fever).
symptom(john, cough).
symptom(john, sore_throat).

disease(john, flu) :-
    symptom(john, fever),
    symptom(john, cough),
    symptom(john, sore_throat).

diagnose(Patient) :-
    disease(Patient, Disease),
    format('~w is diagnosed with ~w.~n', [Patient, Disease]).
input - diagnose(john).
```

29 Write a Prolog Program for forward Chaining. Incorporate required queries.

```
% Facts
fact(sun_is_shining).
fact(weather_is_good) :- fact(sun_is_shining).
fact(go_for_walk) :- fact(weather_is_good).

% Forward chaining rule
forward :-
    fact(go_for_walk),
    write('You can go for a walk!'), nl.
```

30 Write a Prolog Program for backward Chaining. Incorporate required queries.

```
% Rules for backward reasoning
can_go_out :- weather_is_good.
weather_is_good :- sun_is_shining.
sun_is_shining.

% Backward chaining query
?- can_go_out.
```

31 Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc.

```
<!-- Anchor Tag -->
<a href="https://example.com">Visit Example</a>

<!-- Title Tag -->
<title>My Health Blog</title>

<!-- Heading and Paragraph -->
<h1>Healthy Living Tips</h1>
<p>Welcome to my blog on healthy living.</p>
```

32: write a prolog for pattern matching

```
% match(Pattern, Target) succeeds if Pattern matches the Target list

% Base Case: Empty pattern matches empty list
match([], []).

% Recursive Case: Head of pattern matches head of list, rest matches recursively
match([H1|T1], [H2|T2]) :-
    H1 = H2,
    match(T1, T2).

% Example with variables:
% match([X, b, X], [a, b, a]) will succeed with X = a
% match([X, b, X], [a, b, c]) will fail (a ≠ c)

?- match([a, b, c], [a, b, c]).
```

33: number of vowels

```
% vowel(Char) is true if Char is a vowel
vowel(a).
vowel(e).
vowel(i).
vowel(o).
vowel(u).

% count_vowels(InputList, Count) counts the number of vowels in InputList
```

```
% Base case: empty list has 0 vowels
count_vowels([], 0).

% If head is a vowel, increment count and recurse
count_vowels([H|T], Count) :-
    vowel(H),
    count_vowels(T, RestCount),
    Count is RestCount + 1.

% If head is not a vowel, skip it and recurse
count_vowels([H|T], Count) :-
    \+ vowel(H),
    count_vowels(T, Count).
?- count_vowels([h, e, l, l, o], C).
```