EX NO: 1 **NAME:**

DATE: **REG NO:**

Program 1: Solving Problems Using AI

Aim:

To design and implement a simple **AI-based rule-based problem solving system** that diagnoses common computer issues based on user-provided symptoms such as power status, display availability, and beep sounds.

Code:

```
# Simple AI Problem Solving using Rule-Based System

print("=== Simple AI Problem Solver ===")

# Collect symptoms from user

power = input("Is the computer turning ON? (yes/no): ").strip().lower()

screen = input("Is there any display on the screen? (yes/no): ").strip().lower()

beeps = input("Do you hear any beep sounds? (yes/no): ").strip().lower()

# Rule-based reasoning

if power == "no":

    print("\nProblem: POWER ISSUE")

    print("Suggested Solution: Check power cable, SMPS, or wall socket.")

elif power == "yes" and screen == "no":

    print("\nProblem: DISPLAY ISSUE")

    print("Suggested Solution: Check monitor, display cable, or graphics card.")

elif power == "yes" and screen == "yes" and beeps == "yes":

    print("\nProblem: HARDWARE ERROR")

    print("Suggested Solution: RAM or motherboard issue. Re-seat RAM.")

else:
```

```
    print("\nProblem: SOFTWARE ISSUE")
    print("Suggested Solution: Try restarting, run antivirus, or reinstall OS.")
```

Sample Input and Output

Sample Run 1

Input

Is the computer turning ON? (yes/no): no

Is there any display on the screen? (yes/no): no

Do you hear any beep sounds? (yes/no): no

Output

Problem: POWER ISSUE

Suggested Solution: Check power cable, SMPS, or wall socket.

Sample Run 2

Input

Is the computer turning ON? (yes/no): yes

Is there any display on the screen? (yes/no): no

Do you hear any beep sounds? (yes/no): no

Output

Problem: DISPLAY ISSUE

Suggested Solution: Check monitor, display cable, or graphics card.

Sample Run 3

Input

Is the computer turning ON? (yes/no): yes

Is there any display on the screen? (yes/no): yes

Do you hear any beep sounds? (yes/no): yes

Output

Problem: HARDWARE ERROR

Suggested Solution: RAM or motherboard issue. Re-seat RAM.

Result:

Thus, the above program is verified and executed successfully.

EX NO: 2 **NAME:**

DATE: **REG NO:**

Program 2: Solving Problems Using AI

Aim:

To develop a simple AI-based rule-based system that suggests suitable daily activities based on the current weather condition and temperature provided by the user.

Code:

```
# AI Weather Activity Advisor (Simple Rule-Based AI)

print("=== AI Weather Activity Advisor ===")

# Taking inputs

weather = input("How is the weather? (sunny/rainy/cloudy/cold): ").strip().lower()

temperature = int(input("Enter the temperature in Celsius: "))

# Decision making using rules

if weather == "sunny" and temperature > 25:

    print("\nSuggested Activity: Go for a walk or outdoor sports 😊")

elif weather == "cloudy" and 20 <= temperature <= 25:

    print("\nSuggested Activity: A perfect day for a picnic 🧺")

elif weather == "rainy":

    print("\nSuggested Activity: Stay indoors and read a book 📚")

elif temperature < 15:

    print("\nSuggested Activity: Drink something hot and stay warm ☕")

else:

    print("\nSuggested Activity: You can choose any light indoor activity!")
```

Sample Run 1

Input

How is the weather? (sunny/rainy/cloudy/cold): sunny

Enter the temperature in Celsius: 30

Output

Suggested Activity: Go for a walk or outdoor sports.

Sample Run 2

Input

How is the weather? (sunny/rainy/cloudy/cold): cloudy

Enter the temperature in Celsius: 22

Output

Suggested Activity: A perfect day for a picnic.

Result:

Thus, the above program is verified and executed successfully.

EX NO: 3 **NAME:**

DATE: **REG NO:**

Program 3: Propositional Logic and Reasoning

Aim:

To design and implement a Propositional Logic Evaluator using Python that performs logical operations such as AND, OR, NOT, Implication, and Biconditional based on the truth values of given propositions.

Code:

```
# Simple Propositional Logic and Reasoning Program

print("=== Propositional Logic Evaluator ===")

# Taking truth values for propositions

P = input("Enter truth value for P (true/false): ").strip().lower()

Q = input("Enter truth value for Q (true/false): ").strip().lower()

# Convert input to boolean

P = True if P == "true" else False

Q = True if Q == "true" else False

# Logical reasoning

print("\n---- Results ----")

print("P AND Q:", P and Q)

print("P OR Q:", P or Q)

print("NOT P:", not P)

print("P → Q (Implication):", (not P) or Q)

print("P ↔ Q (Biconditional):", P == Q)
```

Sample Run 1

Input

Enter truth value for P (true/false): true

Enter truth value for Q (true/false): true

Output

---- Results ----

P AND Q: True

P OR Q: True

NOT P: False

P → Q (Implication): True

P ↔ Q (Biconditional): True

Result:

Thus, the above program is verified and executed successfully.

EX NO: 4 **NAME:**

DATE: **REG NO:**

Program 4: Expert System in Prolog

Aim:

To develop a simple medical expert system in Prolog that diagnoses possible diseases based on user-reported symptoms.

Code:

```
% --- Knowledge Base ---
disease(cold) :-
    symptom(runny_nose),
    symptom(sneezing),
    symptom(sore_throat).
disease(flu) :-
    symptom(fever),
    symptom(body_ache),
    symptom(fatigue).
disease(allergy) :-
    symptom(sneezing),
    symptom(itchy_eyes),
    symptom(runny_nose).
% --- User Interaction ---
ask(Question) :-
    write('Do you have '), write(Question), write('? (yes/no) '),
    read(Response),
    (Response == yes -> assert(symptom(Question)) ; fail).
```

```
start :-
    write('--- Simple Medical Expert System ---'), nl,
    (ask(runny_nose) ; true),
    (ask(sneezing) ; true),
    (ask(sore_throat) ; true),
    (ask(fever) ; true),
    (ask(body_ache) ; true),
    (ask(fatigue) ; true),
    (ask(itchy_eyes) ; true),
    (disease(D) ->
        write('Based on your symptoms, you may have: '), write(D), nl
    ;   write('No diagnosis could be made.'), nl).
```

Output:

```
?- start.
--- Simple Medical Expert System ---
Do you have runny_nose? (yes/no) yes.
Do you have sneezing? (yes/no) yes.
Do you have sore_throat? (yes/no) no.
Do you have fever? (yes/no) no.
Do you have body_ache? (yes/no) no.
Do you have fatigue? (yes/no) no.
Do you have itchy_eyes? (yes/no) no.
Based on your symptoms, you may have: cold
true.
```

Result:

Thus, the above program is verified and executed successfully.

EX NO: 5 **NAME:**

DATE: **REG NO:**

Program 5: Working on Uninformed Search

Aim:

To implement Depth-First Search (DFS) in Prolog for traversing a graph and finding a path from a start node to a goal node.

Code:

```
% --- Define the graph ---

edge(a, b).

edge(a, c).

edge(b, d).

edge(b, e).

edge(c, f).

edge(e, g).

% --- Depth-First Search (DFS) ---

dfs(Start, Goal, Path) :-

    dfs_helper(Start, Goal, [Start], RevPath),

    reverse(RevPath, Path).

dfs_helper(Goal, Goal, Path, Path).

dfs_helper(Current, Goal, Visited, Path) :-

    edge(Current, Next),

    \+ member(Next, Visited),

    dfs_helper(Next, Goal, [Next|Visited], Path).
```

Output:

?- dfs(a, g, Path).

Path = [a, b, e, g]

true.

This shows DFS found a path from a to g by exploring depth-first order: a → b → d (dead end), backtrack → e → g (goal found).

Result:

Thus, the above program is verified and executed successfully.

EX NO: 6 **NAME:**

DATE: **REG NO:**

Program 6: Working on Uninformed Search - II

Aim:

To implement Breadth-First Search (BFS) in Prolog for traversing a graph and finding a path from a start node to a goal node.

Code:

```
% --- Define the graph ---
edge(a, b).
edge(a, c).
edge(b, d).
edge(b, e).
edge(c, f).
edge(e, g).
% --- Breadth-First Search (BFS) ---
bfs(Start, Goal, Path) :-
    bfs_helper([[Start]], Goal, RevPath),
    reverse(RevPath, Path).
bfs_helper([[Goal|RestPath]|_], Goal, [Goal|RestPath]).
bfs_helper([CurrentPath|OtherPaths], Goal, Path) :-
    CurrentPath = [CurrentNode|_],
    findall([NextNode|CurrentPath],
        (edge(CurrentNode, NextNode), \+ member(NextNode, CurrentPath)),
        NewPaths),
    append(OtherPaths, NewPaths, UpdatedPaths),
```

```
    bfs_helper(UpdatedPaths, Goal, Path).
```

Output:

```
?- bfs(a, g, Path).

Path = [a, b, e, g]

true.
```

Result:

Thus, the above program is verified and executed successfully.

EX NO: 7 **NAME:**

DATE: **REG NO:**

Program 7: Working on Informed Search -I

Aim:

To implement an Informed Search algorithm that uses heuristic values to efficiently find a path from a start state to a goal state.

Code:

```
% ---------- Informed Search ----------

informed_search(Start, Goal, Path) :-
    search([node(Start, [])], Goal, RevPath),
    reverse(RevPath, Path).

search([node(Goal, Path)|_], Goal, [Goal|Path]).

search([node(State, Path)|Rest], Goal, Sol) :-
    findall(
        node(Next, [State|Path]),
        ( move(State, Next),
          \+ member(Next, Path)
        ),
        Children
    ),
        append(Rest, Children, Open),
```

```
    sort_open(Open, Sorted),
    search(Sorted, Goal, Sol).
sort_open(Open, Sorted) :-
    map_list_to_pairs(hval, Open, P),
    keysort(P, SP),
    pairs_values(SP, Sorted).
hval(node(State,_), H) :-
    h(State, H).
% ---------- Problem Definition ----------
move(a,b).
move(b,c).
move(a,c).
h(a,2).
h(b,1).
h(c,0).
```

Execution Query:

```
informed_search(a, c, Path).
```

Output:

Path = [a, c]

Result:

Thus, the above program is verified and executed successfully.

EX NO: 8 **NAME:**

DATE: **REG NO:**

Program 8: Working on Informed Search-II

Aim:

To implement the Hill Climbing Search algorithm and demonstrate how heuristic-based local search works.

Code:

```
% ---------- Hill Climbing Search ----------

hill_climb(Start, Goal, Path) :-
    climb(Start, Goal, [Start], RevPath),
    reverse(RevPath, Path).

climb(Goal, Goal, Path, Path).

climb(State, Goal, Visited, Path) :-
    findall(
        Next,
        ( move(State, Next),
          \+ member(Next, Visited)
        ),
        Neighbours
    ),
    best(Neighbours, Best),
    climb(Best, Goal, [Best|Visited], Path).

% select node with minimum heuristic value

best([X], X).
```

```
best([X,Y|Rest], Best) :-
   h(X, HX),
   h(Y, HY),
   ( HX =< HY ->
      best([X|Rest], Best)
   ;
      best([Y|Rest], Best)
   ).
% ---------- Problem Definition ----------
move(a,b).
move(a,c).
move(b,d).
move(c,d).
h(a,3).
h(b,2).
h(c,1).
h(d,0).
```

Execution Query:

```
hill_climb(a, d, Path).
```

Output:

Path = [a, c, d]

Result:

Thus, the above program is verified and executed successfully.

EX NO: 9 **NAME:**

DATE: **REG NO:**

Program 9: Working with Prolog

Aim:

To implement the Minimax algorithm for a simple game tree and demonstrate decision-making in adversarial (two-player) games.

Code:

```
/* Simple Minimax Program */

% Leaf nodes with values

leaf(l1, 3).

leaf(l2, 5).

leaf(l3, 2).

leaf(l4, 9).

% Tree structure

child(n1, l1).

child(n1, l2).

child(n2, l3).

child(n2, l4).

child(root, n1).

child(root, n2).

% Minimax definition

% Base case: leaf node

minimax(Node, _, Value) :-
```

```
    leaf(Node, Value).
% MAX player
minimax(Node, max, Value) :-
    findall(V,
        ( child(Node, C),
          minimax(C, min, V)
        ),
        Values),
    max_list(Values, Value).
% MIN player
minimax(Node, min, Value) :-
    findall(V,
        ( child(Node, C),
          minimax(C, max, V)
        ),
        Values),
    min_list(Values, Value).
```

Execute Query:

```
minimax(root, max, Value).
```

Output:

Value = 3.

Result:

Thus, the above program is verified and executed successfully.

EX NO: 10 **NAME:**

DATE: **REG NO:**

Program 10: Working on Control Structures in Prolog

Aim:

To demonstrate and illustrate the use of basic control structures and logical constructs in Prolog programming.

Code:

```
/* Simple Control Structures Demo */
 /* -------- Facts -------- */
rich(alice).
rich(bob).
healthy(alice).
sick(bob).
 bird(tweety).
bat(bruce).
 /* -------- AND (,) -------- */
/* happy if rich AND healthy */
happy(X) :-
    rich(X),
    healthy(X).
  /* -------- OR (;) -------- */
/* can fly if bird OR bat */
can_fly(X) :-
    bird(X) ;
    bat(X).
 /* -------- IF-THEN (->) -------- */
/* grade pass if >= 50 */
grade(Marks, pass) :-
```

```
    Marks >= 50, !.


grade(Marks, fail) :-
    Marks < 50.
  /* -------- IF-THEN-ELSE (-> ;) -------- */
/* check age */
check_age(Age, Status) :-
    ( Age >= 18 ->
        Status = adult
    ;
        Status = minor
    ).
  /* -------- Negation (\+) -------- */
/* not happy */
not_happy(X) :-
    \+ happy(X).
  /* -------- CUT (!) -------- */
/* max of two numbers */
max(X, Y, X) :-
    X >= Y, !.
 max(X, Y, Y) :-
    X < Y.
 /* -------- FAIL and Backtracking -------- */
/* print all list elements */
print_all(L) :-
    member(X, L),
    write(X),
    nl,
    fail.
 print_all(_).
```

```
 /* -------- Recursion -------- */
/* factorial */
factorial(0, 1).
 factorial(N, F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F is N * F1.
```

Execution Query:

```
happy(alice),
can_fly(tweety),
grade(70, G),
check_age(20, A),
not_happy(bob),
max(10,5,M),
print_all([1,2,3]),
factorial(5,F).
```

Output:

```
1
2
3
G = pass,
A = adult,
M = 10,
F = 120
```

Result:

Thus, the above program is verified and executed successfully.

EX NO: 11 **NAME:**

DATE: **REG NO:**

Program 11: Working on Recursion in Prolog

Aim:

To implement factorial program using recursive method in prolog.

Code:

```
/* Recursive factorial program */
factorial(0, 1).          % Base case
factorial(N, F) :-        % Recursive case
   N > 0,
   N1 is N - 1,
   factorial(N1, F1),
   F is N * F1.
```

Execution Query:

```
factorial(5, F).
```

Output:

F = 120

Result:

Thus, the above program is verified and executed successfully.

EX NO: 12 **NAME:**

DATE: **REG NO:**

Program 12: Supervised Learning

Aim:

To implement a Supervised learning in Python to predict students marks based on study hours.

Code:

```
/*Problem: Predict marks based on study hours. This is supervised learning (regression) because data is labeled. */

/* Supervised Learning using Linear Regression */

from sklearn.linear_model import LinearRegression

import numpy as np

# Training data (Hours studied -> Marks obtained)

X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)   # input

Y = np.array([35, 40, 50, 60, 75])            # output (labels)

# Create model

model = LinearRegression()

# Train model

model.fit(X, Y)

# Predict marks for 6 hours of study

hours = np.array([[6]])

prediction = model.predict(hours)

print("Predicted marks for 6 hours study:", prediction[0])
```

Output:

Predicted marks for 6 hours study: 82.0

Result:

Thus, the above program is verified and executed successfully.

EX NO: 13 **NAME:**

DATE: **REG NO:**

Program 13: Simple Bayesian Learning Program

Aim:

To implement a simple Bayesian Learning program in Python that uses Bayes' Theorem to predict whether to "Play" or "No Play" based on the given weather condition.

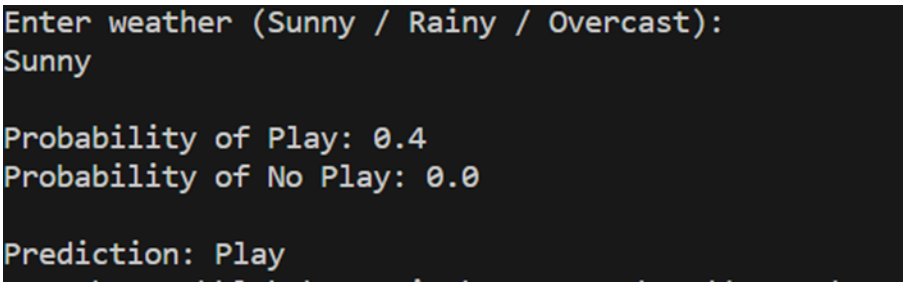
Code:

```
# Weather Prediction Example
# Training data
data = [
    ("Sunny", "Play"),
    ("Sunny", "Play"),
    ("Rainy", "No Play"),
    ("Overcast", "Play"),
    ("Rainy", "No Play")
]
 # Count classes
play_count = 0
noplay_count = 0
 for item in data:
    if item[1] == "Play":
        play_count += 1
    else:
        noplay_count += 1
 total = len(data)
```

```
 # Calculate prior probabilities
p_play = play_count / total
p_noplay = noplay_count / total
 # User input
print("Enter weather (Sunny / Rainy / Overcast):")
weather = input()
 # Count feature occurrences
play_weather = 0
noplay_weather = 0
 for item in data:
    if item[0] == weather and item[1] == "Play":
        play_weather += 1
    if item[0] == weather and item[1] == "No Play":
        noplay_weather += 1
 # Likelihoods
if play_count != 0:
    p_weather_play = play_weather / play_count
else:
    p_weather_play = 0
 if noplay_count != 0:
    p_weather_noplay = noplay_weather / noplay_count
else:
    p_weather_noplay = 0
 # Bayes calculation
final_play = p_play * p_weather_play
final_noplay = p_noplay * p_weather_noplay
# Output
print("\nProbability of Play:", round(final_play, 4))
```

```
print("Probability of No Play:", round(final_noplay, 4))
 # Decision
if final_play > final_noplay:
    print("\nPrediction: Play")
else:
    print("\nPrediction: No Play")
```

Output:



Result:

Thus, the above program is verified and executed successfully.

EX NO: 14 **NAME:**

DATE: **REG NO:**

Program 14: Linear models for clustering (k-means)

Aim:

To implement a simple linear clustering program using the K-Means algorithm in Python to group data points into clusters based on their similarity.

Code:

```
# Simple K-Means Clustering Program
# Beginner Friendly - No Libraries Needed
 # Sample Data Points (1D values)
data = [2, 4, 10, 12, 3, 20, 30, 11, 25]
 # Number of clusters
k = 2
 # Initial cluster centers (first two values)
c1 = data[0]
c2 = data[1]
 print("Initial Centers:", c1, c2)
 # Run clustering for few iterations
for step in range(5):
    cluster1 = []
    cluster2 = []
     # Assign points to nearest center
    for value in data:
        if abs(value - c1) < abs(value - c2):
            cluster1.append(value)
        else:
            cluster2.append(value)
```

```
    # Update centers
    if cluster1:
        c1 = sum(cluster1) / len(cluster1)
    if cluster2:
        c2 = sum(cluster2) / len(cluster2)
     print("\nStep", step + 1)
    print("Cluster 1:", cluster1)
    print("Cluster 2:", cluster2)
    print("New Centers:", round(c1, 2), round(c2, 2))
 print("\nFinal Clusters:")
print("Cluster 1:", cluster1)
print("Cluster 2:", cluster2)
```

Output:

```
Initial Centers: 2 4

Step 1
Cluster 1: [2]
Cluster 2: [4, 10, 12, 3, 20, 30, 11, 25]
New Centers: 2.0 14.38

Step 2
Cluster 1: [2, 4, 3]
Cluster 2: [10, 12, 20, 30, 11, 25]
New Centers: 3.0 18.0

Step 3
Cluster 1: [2, 4, 10, 3]
Cluster 2: [12, 20, 30, 11, 25]
New Centers: 4.75 19.6

Step 4
Cluster 1: [2, 4, 10, 12, 3, 11]
Cluster 2: [20, 30, 25]
New Centers: 7.0 25.0

Step 5
Cluster 1: [2, 4, 10, 12, 3, 11]
Cluster 2: [20, 30, 25]
New Centers: 7.0 25.0

Final Clusters:
Cluster 1: [2, 4, 10, 12, 3, 11]
Cluster 2: [20, 30, 25]
```

Result:

Thus, the above program is verified and executed successfully.

EX NO: 15 **NAME:**

DATE: **REG NO:**

Program 15: Reinforcement learning

Aim:

To implement a simple Reinforcement Learning program in Python that trains an agent to choose actions based on rewards and penalties to learn an optimal decision-making policy.

Code:

```
# Simple Reinforcement Learning Program
# Agent learns by trial and error
 # Possible actions
actions = ["Left", "Right"]
 # Rewards for actions
rewards = {
    "Left": -1,    # Penalty
    "Right": 1    # Reward
}
 # Initialize knowledge (Q-values)
Q = {
    "Left": 0,
    "Right": 0
}
 # Learning rate
alpha = 0.5
 # Training episodes
episodes = 10
 print("Training Agent...\n")
 # Learning process
```

```
for i in range(episodes):
    for action in actions:
        # Update Q-value
        Q[action] = Q[action] + alpha * (rewards[action] - Q[action])
    print("Episode", i + 1, "Q-values:", Q)
# Final decision
best_action = max(Q, key=Q.get)
print("\nBest Action Learned:", best_action)
```

Output:

```
Training Agent...

Episode 1 Q-values: {'Left': -0.5, 'Right': 0.5}
Episode 2 Q-values: {'Left': -0.75, 'Right': 0.75}
Episode 3 Q-values: {'Left': -0.875, 'Right': 0.875}
Episode 4 Q-values: {'Left': -0.9375, 'Right': 0.9375}
Episode 5 Q-values: {'Left': -0.96875, 'Right': 0.96875}
Episode 6 Q-values: {'Left': -0.984375, 'Right': 0.984375}
Episode 7 Q-values: {'Left': -0.9921875, 'Right': 0.9921875}
Episode 8 Q-values: {'Left': -0.99609375, 'Right': 0.99609375}
Episode 9 Q-values: {'Left': -0.998046875, 'Right': 0.998046875}
Episode 10 Q-values: {'Left': -0.9990234375, 'Right': 0.9990234375}

Best Action Learned: Right
```

Result:

Thus, the above program is verified and executed successfully.