>>>>>>>>>>>>>>>>>> Inputs for prolog programmes..................

17....   sum_n(5, Sum).

18..... find_dob('Alice Smith', DOB).

19.... find_teacher_and_subject('Alice', Teacher, Subject, Subject_Code).

20...... find_planet('Mars', Type, Distance, Diameter).

21....   hanoi(3, 'A', 'C', 'B').

22....   can_fly(sparrow). or   cannot_fly(ostrich).

23....   find_father(mary, Father).

24....   suggest_diet(diabetes).

25....   solve.

26....   find_fruit_by_color(yellow, Fruit).

27...   best_first_search(a, j, Path).

28...   diagnose(Disease).   and give yes

(((('''' if this programme is not running directly in the software go to command prompt and type(cd C:\GNU-Prolog\bin C:\GNU-Prolog\bin>gprolog)

29.... forward_chaining(hot).

30...... backward_chaining(flood).

31.... https://kotiking123.wordpress.com/2025/02/25/koti-group-of-companies/   CREATING WEBBLOG...

32.... match_pattern([a, b, c], [a, b, c]).

33.... count_vowels_in_list([h, e, l, l, o], Count).

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>prolog codes..................

17. sum

```
% Base case: The sum of numbers from 1 to 0 is 0.
sum_n(0, 0).

% Recursive case: Sum of numbers from 1 to N.
sum_n(N, Sum) :-
   N > 0,
   N1 is N - 1,
   sum_n(N1, Sum1),
   Sum is Sum1 + N.
```

18. DOB

```
% Facts: Name and Date of Birth (DOB)
person('John Doe', '1995-06-15').
person('Alice Smith', '1998-11-23').

person('Bob Johnson', '2000-03-10').


% Rule to find the DOB of a given person

find_dob(Name, DOB) :- person(Name, DOB).
```

19. student teacher

```
% Facts: student(Name, Teacher, Subject, Subject_Code)

student('Alice', 'Mr. Sharma', 'Mathematics', 'M101').

student('Bob', 'Mrs. Iyer', 'Physics', 'P202').

student('Charlie', 'Dr. Rao', 'Computer Science', 'CS303').

student('David', 'Mr. Sharma', 'Mathematics', 'M101').

student('Eve', 'Mrs. Iyer', 'Physics', 'P202').


% Rule to find the teacher and subject based on student's name

find_teacher_and_subject(Student, Teacher, Subject, Subject_Code) :-

    student(Student, Teacher, Subject, Subject_Code).
```

20. planet

```
% Facts: planet(Name, Type, Distance_from_Sun_Million_KM, Diameter_KM).
planet('Mercury', 'Terrestrial', 57.9, 4879).
planet('Venus', 'Terrestrial', 108.2, 12104).
planet('Earth', 'Terrestrial', 149.6, 12742).
planet('Mars', 'Terrestrial', 227.9, 6779).
planet('Jupiter', 'Gas Giant', 778.3, 139820).
planet('Saturn', 'Gas Giant', 1427, 116460).
planet('Uranus', 'Ice Giant', 2871, 50724).
planet('Neptune', 'Ice Giant', 4495, 49244).

% Rule to find details of a planet based on its name.
find_planet(Name, Type, Distance, Diameter) :-
    planet(Name, Type, Distance, Diameter).
```

21.Towers of Hanoi

```
% Base Case: Move 1 disk directly from Source to Destination
hanoi(1, Source, Destination, _) :-
    write('Move disk from '), write(Source), write(' to '), write(Destination), nl.
```

```
% Recursive Case: Move N disks using Auxiliary as intermediate

hanoi(N, Source, Destination, Auxiliary) :-

    N > 1,

    M is N - 1,

    hanoi(M, Source, Auxiliary, Destination),  % Move N-1 disks to Auxiliary

    hanoi(1, Source, Destination, _),         % Move the largest disk to Destination

    hanoi(M, Auxiliary, Destination, Source). % Move N-1 disks from Auxiliary to Destination
```

22. birds can fly or not

```
% Facts: bird(Name, CanFly)

bird(sparrow, yes).

bird(eagle, yes).

bird(pigeon, yes).

bird(ostrich, no).

bird(penguin, no).

bird(parrot, yes).

bird(kiwi, no).


% Rule to check if a bird can fly

can_fly(Bird) :-

    bird(Bird, yes),

    write(Bird), write(' can fly.'), nl.
```

```
cannot_fly(Bird) :-
    bird(Bird, no),
    write(Bird), write(' cannot fly.'), nl.
```

23.Family tree

```
% Facts: Relationships in the family
parent(john, mary).
parent(john, david).
parent(susan, mary).
parent(susan, david).
parent(mary, alice).
parent(mary, bob).
parent(david, charlie).
parent(david, eve).

% Gender Information
male(john).
male(david).
male(bob).
male(charlie).

female(susan).
female(mary).
female(alice).
female(eve).

% Rules: Define relationships
father(F, C) :- parent(F, C), male(F).
mother(M, C) :- parent(M, C), female(M).

sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.
brother(B, S) :- sibling(B, S), male(B).
sister(S, B) :- sibling(S, B), female(S).

grandparent(GP, C) :- parent(GP, P), parent(P, C).
grandfather(GF, C) :- grandparent(GF, C), male(GF).
grandmother(GM, C) :- grandparent(GM, C), female(GM).

% Query Rules
find_father(Child, Father) :- father(Father, Child).
find_mother(Child, Mother) :- mother(Mother, Child).
find_siblings(Person, Sibling) :- sibling(Person, Sibling).
find_grandparent(Child, Grandparent) :- grandparent(Grandparent, Child).
```

24. suggest diet

```
% Facts: Diet recommendations for specific diseases
```

```
diet(diabetes, 'Eat fiber-rich foods, whole grains, lean proteins, and avoid sugar & processed food.').
diet(hypertension, 'Consume low-sodium foods, leafy greens, bananas, and avoid salty & fried items.').
diet(obesity, 'Eat high-fiber foods, fruits, vegetables, and avoid junk food & sugary drinks.').
diet(anemia, 'Increase iron intake with leafy greens, red meat, and avoid caffeine while eating iron-rich foods.').
diet(heart_disease, 'Eat omega-3 rich foods, whole grains, and avoid trans fats & processed meat.').
diet(kidney_disease, 'Limit sodium, potassium, and eat high-quality protein sources.').

% Rule to suggest a diet based on disease
suggest_diet(Disease) :-
   diet(Disease, Diet),
   write('Recommended diet for '), write(Disease), write(': '), nl,
   write(Diet), nl.
```

25.Monkey BAnana

```
% Initial state: Monkey is on the ground and banana is on a high platform
state(at_ground, not_holding_banana).

% Actions the monkey can perform
```

```
action(walk) :- write('Monkey walks towards the banana.\n').
action(climb) :- write('Monkey climbs the platform.\n').
action(grab) :- write('Monkey grabs the banana!\n').

% Rule to solve the problem
solve :-
    action(walk),
    action(climb),
    action(grab),
    write('Monkey successfully gets the banana!').
```

26.Fruit colouring

```
% Facts: Fruit and their colors
fruit_color(apple, red).
fruit_color(banana, yellow).
fruit_color(grape, purple).
fruit_color(orange, orange).
fruit_color(guava, green).
fruit_color(mango, yellow).
fruit_color(strawberry, red).

% Rule to find the color of a fruit
find_color(Fruit, Color) :-
    fruit_color(Fruit, Color).

% Rule to find fruits of a specific color
find_fruit_by_color(Color, Fruit) :-
    fruit_color(Fruit, Color).
```

27...BFS

```
% Facts: Define edges with their corresponding cost
edge(a, b, 4).
edge(a, c, 3).
edge(b, d, 5).
edge(b, e, 12).
edge(c, f, 8).
edge(d, g, 7).
edge(e, h, 2).
edge(f, i, 6).
edge(g, j, 10).

% Heuristic values (Estimated cost to goal)
heuristic(a, 10).
heuristic(b, 6).
heuristic(c, 8).
heuristic(d, 5).
heuristic(e, 2).
heuristic(f, 7).
heuristic(g, 3).
heuristic(h, 1).
heuristic(i, 4).
heuristic(j, 0).  % Goal node

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

% If goal is reached, return the path
best_first([[Goal | Rest] | _], Goal, [Goal | Rest]).
```

```
% Expand the best path and continue searching
best_first([[Node | Path] | OtherPaths], Goal, FinalPath) :-
    findall([Next, Node | Path],
            (edge(Node, Next, _), \+ member(Next, [Node | Path])),
            NewPaths),
    append(OtherPaths, NewPaths, UpdatedPaths),
    sort_paths_by_heuristic(UpdatedPaths, SortedPaths),
    best_first(SortedPaths, Goal, FinalPath).

% Sort paths based on the heuristic value of the first node in each path
sort_paths_by_heuristic(Paths, SortedPaths) :-
    get_heuristic_list(Paths, HeuristicPairs),
    sort_heuristic_list(HeuristicPairs, SortedPairs),
    extract_paths(SortedPairs, SortedPaths).

% Get heuristic values for each path
get_heuristic_list([], []).
get_heuristic_list([[Node | Rest] | Paths], [(H, [Node | Rest]) | HeuristicPairs]) :-
    heuristic(Node, H),
    get_heuristic_list(Paths, HeuristicPairs).

% Simple sorting based on heuristic values (Selection Sort)
sort_heuristic_list(List, Sorted) :- sort_heuristic(List, [], Sorted).

sort_heuristic([], Acc, Acc).
sort_heuristic(List, Acc, Sorted) :-
```

```
    select_min(List, Min, Rest),
    sort_heuristic(Rest, [Min | Acc], Sorted).

% Select the path with the minimum heuristic value
select_min([X], X, []).
select_min([(H1, P1), (H2, P2) | Rest], Min, [(H2, P2) | NewRest]) :-
    H1 =< H2, !, select_min([(H1, P1) | Rest], Min, NewRest).
select_min([(H1, P1), (H2, P2) | Rest], Min, [(H1, P1) | NewRest]) :-
    select_min([(H2, P2) | Rest], Min, NewRest).

% Extract only paths from sorted (H, Path) pairs
extract_paths([], []).
extract_paths([(_, Path) | Rest], [Path | Paths]) :-
    extract_paths(Rest, Paths).
```

28...Medical Diagnosis..

```
% Medical Diagnosis System
symptom(flu, fever).
symptom(flu, runny_nose).
symptom(covid19, fever).
symptom(covid19, dry_cough).
symptom(malaria, fever).
symptom(malaria, chills).

% Diagnosis Rule
diagnose(Disease) :-
    symptom(Disease, Symptom),
    write('Do you have '), write(Symptom), write('? (yes/no): '),
    read(yes),
    write('You might have '), write(Disease), nl, !.

diagnose(_) :-
    write('No matching disease found. Stay healthy!'), nl.
```

29. Forward CHAINING

```
% Facts
fact(sunny).
fact(warm).
fact(summer).
```

```
% Rules
infer(hot) :- fact(sunny), fact(warm).
infer(go_to_beach) :- infer(hot), fact(summer).

% Forward Chaining Execution
forward_chaining(Goal) :- infer(Goal), !.
forward_chaining(Goal) :- fact(Goal), !.
forward_chaining(_) :- write('Goal cannot be inferred'), fail.
```

30.Bckward chain

```
% Facts
fact(rains).
fact(wet_ground).

% Rules
rule(wet_ground) :- fact(rains).
rule(flood) :- fact(wet_ground), fact(heavy_rain).

% Backward Chaining Execution
backward_chaining(Goal) :- fact(Goal), !.
backward_chaining(Goal) :- rule(Goal), !.
backward_chaining(_) :- write('Goal cannot be proven'), fail.
```

31.https://kotiking123.wordpress.com/2025/02/25/koti-group-of-companies/ CREATING WEBBLOG...

32..pattern matching

```
% Pattern matches an empty list (Base Case)

pattern_match([], []).


% If the heads match, recursively check the tail

pattern_match([H|T], [H|T2]) :-

    pattern_match(T, T2).


% If there is a wildcard `_`, skip one element and continue checking

pattern_match([_|T], [_|T2]) :-

    pattern_match(T, T2).


% Predicate to check if a pattern matches a list

match_pattern(Pattern, List) :-

    pattern_match(Pattern, List),

    write('Pattern matches!'), nl.
```

```
match_pattern(_, _) :-
    write('Pattern does not match!'), nl.
```

33..Voweels

```
% Define vowels
is_vowel(a).
is_vowel(e).
is_vowel(i).
is_vowel(o).
is_vowel(u).
is_vowel(A).
is_vowel(E).
is_vowel(I).
is_vowel(O).
is_vowel(U).

% Base case: Empty list, count is 0
count_vowels([], 0).

% If the first character is a vowel, increase the count
count_vowels([H|T], Count) :-
    is_vowel(H),
    count_vowels(T, RestCount),
    Count is RestCount + 1.
```

```
% If the first character is not a vowel, continue checking
count_vowels([_|T], Count) :-
    count_vowels(T, Count).

% Predicate to count vowels in a character list
count_vowels_in_list(CharList, Count) :-
    count_vowels(CharList, Count).
```