Prolog

1. Write prolog program to perform below operations.

- Logical AND, OR, NOT, NOR, NAND.
- Fibonacci Series.
- If a number is member of a list or not.
- Find the minimum and maximum of a list.
- GCD of a number.
- Concatenation of two list.
- Reverse of a list.
- Union, intersection of two list.

1. Logical AND, OR, NOT, NOR, NAND.

Program:

```
and(0, _, 0).
and(_, 0, 0).
and(1, 1, 1).

or(X, 0, X).
or(0, X, X).
or(1, X, 1).

not(X, R):- R is 1-X.

nor(A, B, R):- or(A, B, Res), not(Res, R).

nand(A, B, R):- and(A, B, Res), not(Res, R).
```

Output:

```
11 ?- [prolog].
Warning: c:/users/arnab/onedrive/desktop/prolog.pl:7:
Warning:    Singleton variables: [X]
true.

12 ?- and(1, R, 1).
R = 1.

13 ?- and(0, R, X).
X = 0 ;
R = X, X = 0.

14 ?- or(X, 1, 1).
X = 0 ;
X = 1.

15 ?- not(1, X).
X = 0.

16 ?- nor(0, 0, X).
X = 1 ;
X = 1.

17 ?- nor(1, 0, X).
X = 0 .

18 ?- nand(1, 0, X).
X = 1 .

19 ?- nand(0, 0, X).
X = 1 .

20 ?- nand(1, 1, X).
X = 0.
```

2. Fibonacci Series.

```
fib(N, F):- N > 2, N1 is N - 1, N2 is N - 2, fib(N1, F1), fib(N2, F2), F is F1
+ F2.
fib(N, 1):- N =< 2, N > 0.
```

Output:

```
21 ?- [prolog].
true.

22 ?- fib(5, A).
A = 5 .

23 ?- fib(7, A).
A = 13 .

24 ?- fib(10, A).
A = 55 .
```

3. If a number is a member of a list or not.

Program:

```
is_member(X, [X|_]).
is_member(X, [_|Tail]) :- is_member(X, Tail).
```

Output:

```
25 ?- [prolog].
true.

26 ?- is_member(4, [1, 2, 3, 4, 5, 6]).
true .

27 ?- is_member(10, [1, 2, 3, 4, 5, 6]).
false.
```

4. Find the minimum and maximum of a list.

Program:

```
minm(X, Y, X):- Y > X.
minm(X, Y, Y):- X >= Y.

minl([X, Y|T], M):- minm(X, Y, Z), minl([Z|T], M).
minl([X], X).

maxm(X, Y, X):- X >= Y.
maxm(X, Y, Y):- Y > X.

maxl([X, Y|T], M):- maxm(X, Y, Z), maxl([Z|T], M).
maxl([X], X).
```

Output:

```
38 ?- [prolog].
true.

39 ?- minl([1, 2, 3, 4, 5, 6], M).
M = 1 .

40 ?- maxl([1, 2, 3, 4, 5, 6], M).
M = 6 .
```

5. GCD of two numbers.

Program:

```
gcd(X, Y, Z):- X > Y, X1 is X - Y, gcd(X1, Y, Z).
gcd(X, Y, Z):- Y > X, Y1 is Y - X, gcd(X, Y1, Z).
gcd(X, X, X).
```

Output:

```
42 ?- [prolog].
true.

42 ?- gcd(12, 48, X).
X = 12 .

43 ?- gcd(12, 49, X).
X = 1 .

44 ?- gcd(12, 4, X).
X = 4 .
```

6. Concatenation of two lists.

Program:

```
concat([H|T], L, [H|C]):- concat(T, L, C).
concat([], L, L).
```

Output:

```
45 ?- [prolog].
true.

46 ?- concat([1, 2, 3], [a, b, c], L).
L = [1, 2, 3, a, b, c].

47 ?- concat([1, 2, 3, EFG], [a, b, c], L).
L = [1, 2, 3, EFG, a, b, c].
```

7. Reverse of a list.

Program:

```
rev([H|T], L):- rev(T, A), concat(A, [H], L).
rev([], []).
```

Output:

```
50 ?- [prolog].
true.

51 ?- rev([1, 2, 3, 4, 5], A).
A = [5, 4, 3, 2, 1].

52 ?- rev([1, 2, fsh, v, 3, 4, 5], A).
A = [5, 4, 3, v, fsh, 2, 1].
```

8. Union & Intersection of two list.

Program:

```
union([H|T], Y, Z):- is_member(H, Y), union(T, Y, Z).
union([H|T], Y, [H|Z]):- \+is_member(H, Y), union(T, Y, Z).
union([], L, L).

intersection([H|T], Y, [H|Z]):- is_member(H, Y), intersection(T, Y, Z).
intersection([H|T], Y, Z):- not(is_member(H, Y)), intersection(T, Y, Z).
intersection([], _, []).
```

2. Write a PROLOG program to calculate the sum of two numbers.

```
% Rule to calculate the sum of two numbers
sum(X, Y, Result) :-
   Result is X + Y.
```

---

3. Write a PROLOG program to find the maximum of two numbers.

```
% Rule to find the maximum of two numbers
max(X, Y, Max) :-
   (X >= Y, Max = X) ; (Y > X, Max = Y).
```

---

4. Write a PROLOG program to find the area and perimeter of a square with the side length
being provided by the user.

```
% Rule to calculate the area of a square
square_area(Side, Area) :-
    Area is Side * Side.

% Rule to calculate the perimeter of a square
square_perimeter(Side, Perimeter) :-
    Perimeter is 4 * Side.

% Rule to display the area and perimeter of a square
display_square_info(Side) :-
    square_area(Side, Area),
    square_perimeter(Side, Perimeter),
    format('Square with side length ~w has area ~w and perimeter ~w.~n', [Side, Area, Perimeter]).
```

---

5. Write a PROLOG program to find the area and perimeter of a circle with the radius
being provided by the user.

```
% Rule to calculate the area of a circle
circle_area(Radius, Area) :-
    Area is pi * Radius * Radius.

% Rule to calculate the perimeter (circumference) of a circle
circle_perimeter(Radius, Perimeter) :-
    Perimeter is 2 * pi * Radius.

% Rule to display the area and perimeter of a circle
display_circle_info(Radius) :-
    circle_area(Radius, Area),
    circle_perimeter(Radius, Perimeter),
    format('Circle with radius ~w has area ~w and perimeter ~w.~n', [Radius, Area, Perimeter]).
```

---
6. Write a PROLOG program to print a Fibonacci series with stating numbers are provided
by user.

```
% Rule to print Fibonacci series without using lists
print_fibonacci(First, Second, N) :-
    N > 0,
    write(First), nl,
    Next is First + Second,
    print_fibonacci(Second, Next, N - 1).

% Example usage
% print_fibonacci(FirstNumber, SecondNumber, NumberOfTerms).
```

---
7. Write a PROLOG program to calculate the sum of natural numbers up to a limit (using
recursion).

```
% Rule to calculate the sum of natural numbers up to a limit
sum_of_naturals(0, 0).  % Base case: sum of 0 natural numbers is 0
sum_of_naturals(N, Sum) :-
    N > 0,
    N1 is N - 1,
    sum_of_naturals(N1, SubSum),
    Sum is N + SubSum.

% Example usage
% To calculate the sum of natural numbers up to the limit of 5:
% ?- sum_of_naturals(5, Result).
% Result will be unified with 15.
```

---
8. Write a PROLOG program to calculate the sum of a range.

```
% Rule to calculate the sum of a range of numbers
sum_of_range(Start, End, Sum) :-
    sum_of_range_helper(Start, End, 0, Sum).

% Base case: Sum of the range [N, N] is N
sum_of_range_helper(N, N, Acc, Sum) :-
    Sum is Acc + N.

% Recursive case: Sum of the range [Start, End] is Start + Sum of [Start+1,
End]
sum_of_range_helper(Start, End, Acc, Sum) :-
    Start < End,
    NextStart is Start + 1,
    NewAcc is Acc + Start,
    sum_of_range_helper(NextStart, End, NewAcc, Sum).

% Example usage
% To calculate the sum of the range from 1 to 5:
% ?- sum_of_range(1, 5, Result).
% Result will be unified with 15.
```

---

9. Write a PROLOG program to calculate the factorial of a given number.

```
% Rule to calculate the factorial of a number
factorial(0, 1).  % Base case: factorial of 0 is 1
factorial(N, Result) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, SubResult),
    Result is N * SubResult.

% Example usage
% To calculate the factorial of 5:
% ?- factorial(5, Result).
% Result will be unified with 120.
```

---

10. Write a PROLOG program to find the last element of a list(list must have at least 4

items).

```
% Rule to find the last element of a list
last_element([X], X).  % Base case: last element of a list with one item is the item itself
last_element([_|Tail], Last) :-
    last_element(Tail, Last).

% Example usage
% To find the last element of a list [1, 2, 3, 4]:
% ?- last_element([1, 2, 3, 4], Result).
% Result will be unified with 4.
```

---

11. Write a PROLOG program to find the length of a list (list must have at least 4 elements).

```
% Rule to find the length of a list
list_length([], 0).  % Base case: length of an empty list is 0
list_length([_|Tail], Length) :-
    list_length(Tail, TailLength),
    Length is TailLength + 1.

% Example usage
% To find the length of a list [1, 2, 3, 4]:
% ?- list_length([1, 2, 3, 4], Result).
% Result will be unified with 4.
```

---

12. Write a PROLOG program to find the average of a list of numbers

```
% Rule to calculate the sum of a list of numbers
sum_list([], 0).  % Base case: sum of an empty list is 0
sum_list([X|Xs], Sum) :-
    sum_list(Xs, TailSum),
    Sum is X + TailSum.

% Rule to calculate the length of a list
list_length([], 0).  % Base case: length of an empty list is 0
list_length([_|Tail], Length) :-
```

```
    list_length(Tail, TailLength),
    Length is TailLength + 1.

% Rule to calculate the average of a list of numbers
average_list([], 0).  % Base case: average of an empty list is 0
average_list(List, Average) :-
    sum_list(List, Sum),
    list_length(List, Length),
    Length > 0,  % To avoid division by zero
    Average is Sum / Length.

% Example usage
% To find the average of a list [1, 2, 3, 4]:
% ?- average_list([1, 2, 3, 4], Result).
% Result will be unified with 2.5.
```

---

13. Write a PROLOG program to implement maxlist(List,Max) so that Max is the greatest
among all the elements present in the list (list must have at least 4 items).

```
% Rule to find the maximum element in a list
maxlist([X], X).  % Base case: maximum element in a list with one item is the item itself
maxlist([H|T], Max) :-
    maxlist(T, TailMax),
    (H > TailMax, Max = H; H =< TailMax, Max = TailMax).

% Example usage
% To find the maximum element in a list [3, 1, 5, 2]:
% ?- maxlist([3, 1, 5, 2], Result).
% Result will be unified with 5.
```

---

14. Write a PROLOG program to calculate the sum of numbers present in the List. (At first
create a list of numbers. The length of the list must not be less than 4.)

```
% Rule to calculate the sum of numbers in a list
sum_list([], 0).  % Base case: sum of an empty list is 0
```

```
sum_list([X|Xs], Sum) :-
    sum_list(Xs, TailSum),
    Sum is X + TailSum.

% Example usage
% To calculate the sum of numbers in a list [1, 2, 3, 4]:
% ?- sum_list([1, 2, 3, 4], Result).
% Result will be unified with 10.
```

---

15. Write a PROLOG program to find out the GCD and LCM of two numbers. Use these
predicates to find out the GCD and LCM of a list of numbers.

```
% Rule to find the GCD of two numbers
gcd(X, 0, X) :- X > 0.
gcd(X, Y, GCD) :-
    Y > 0,
    Z is X mod Y,
    gcd(Y, Z, GCD).

% Rule to find the LCM of two numbers
lcm(X, Y, LCM) :-
    gcd(X, Y, GCD),
    LCM is abs(X * Y) // GCD.

% Rule to find the GCD of a list of numbers
gcd_list([], 0).
gcd_list([X], X).
gcd_list([X, Y | Rest], GCD) :-
    gcd(X, Y, TempGCD),
    gcd_list([TempGCD | Rest], GCD).

% Rule to find the LCM of a list of numbers
lcm_list([], 1).
lcm_list([X], X).
lcm_list([X, Y | Rest], LCM) :-
    lcm(X, Y, TempLCM),
    lcm_list([TempLCM | Rest], LCM).

% Example usage
```

```
% To find the GCD and LCM of a list of numbers [12, 18, 24]:
% ?- gcd_list([12, 18, 24], GCDResult).
% GCDResult will be unified with 6.
%
% ?- lcm_list([12, 18, 24], LCMResult).
% LCMResult will be unified with 72.
```

---

16. Write a PROLOG program to insert an element at the kth position of a list.

```
% Rule to insert an element at the kth position of a list
insert_at(Element, 1, List, [Element|List]).
insert_at(Element, K, [Head|Tail], [Head|Result]) :-
    K > 1,
    K1 is K - 1,
    insert_at(Element, K1, Tail, Result).

% Example usage
% To insert the element 42 at the 3rd position of the list [1, 2, 3, 4]:
% ?- insert_at(42, 3, [1, 2, 3, 4], Result).
% Result will be unified with [1, 2, 42, 3, 4].
```

---

17. Write a PROLOG program to compute the sum of the digits of an integer.

```
% Rule to compute the sum of the digits of an integer
sum_of_digits(0, 0).  % Base case: sum of digits of 0 is 0
sum_of_digits(N, Sum) :-
    N > 0,
    Digit is N mod 10,
    N1 is N // 10,
    sum_of_digits(N1, SubSum),
    Sum is Digit + SubSum.

% Example usage
% To compute the sum of digits of the integer 12345:
% ?- sum_of_digits(12345, Result).
% Result will be unified with 15.
```

--

19. Write a PROLOG program to print the reverse of a given list.

```
% Rule to reverse a list
reverse_list([], []).
reverse_list([Head|Tail], Reversed) :-
   reverse_list(Tail, TailReversed),
   append(TailReversed, [Head], Reversed).

% Example usage
% To reverse the list [1, 2, 3, 4]:
% ?- reverse_list([1, 2, 3, 4], Reversed).
% Reversed will be unified with [4, 3, 2, 1].
```

---

20. Write a PROLOG program to concatenate two given lists.

```
% Rule to concatenate two lists
concatenate_lists([], L, L).
concatenate_lists([Head|Tail1], L2, [Head|Result]) :-
   concatenate_lists(Tail1, L2, Result).

% Example usage
% To concatenate the lists [1, 2, 3] and [4, 5, 6]:
% ?- concatenate_lists([1, 2, 3], [4, 5, 6], Result).
% Result will be unified with [1, 2, 3, 4, 5, 6].
```

---

21. Write a PROLOG program to find the union, difference and intersection of two given
lists.

```
% Rule to find the union of two lists
union_lists([], L, L).
union_lists([Head|Tail], L2, Union) :-
   member(Head, L2),  % Check if Head is in L2
   !,
   union_lists(Tail, L2, Union).
union_lists([Head|Tail], L2, [Head|Union]) :-
   union_lists(Tail, L2, Union).
```

```
% Rule to find the difference of two lists
difference_lists([], _, []).
difference_lists([Head|Tail], L2, Difference) :-
   member(Head, L2),  % Check if Head is in L2
   !,
   difference_lists(Tail, L2, Difference).
difference_lists([Head|Tail], L2, [Head|Difference]) :-
   difference_lists(Tail, L2, Difference).

% Rule to find the intersection of two lists
intersection_lists([], _, []).
intersection_lists([Head|Tail], L2, Intersection) :-
   member(Head, L2),  % Check if Head is in L2
   !,
   Intersection = [Head|Rest],
   intersection_lists(Tail, L2, Rest).
intersection_lists([_|Tail], L2, Intersection) :-
   intersection_lists(Tail, L2, Intersection).

% Example usage
% To find the union, difference, and intersection of the lists [1, 2, 3, 4] and
[3, 4, 5, 6]:
% ?- union_lists([1, 2, 3, 4], [3, 4, 5, 6], UnionResult).
% UnionResult will be unified with [1, 2, 3, 4, 5, 6].
%
% ?- difference_lists([1, 2, 3, 4], [3, 4, 5, 6], DifferenceResult).
% DifferenceResult will be unified with [1, 2].
%
% ?- intersection_lists([1, 2, 3, 4], [3, 4, 5, 6], IntersectionResult).
% IntersectionResult will be unified with [3, 4].
```

-

22. Write a PROLOG program to check whether a given list is palindrome or not.

```
% Rule to check if a list is a palindrome
is_palindrome(List) :-
   reverse_list(List, List).

% Rule to reverse a list
reverse_list([], []).
```

```
reverse_list([Head|Tail], Reversed) :-
    reverse_list(Tail, TailReversed),
    append(TailReversed, [Head], Reversed).

% Example usage
% To check if the list [1, 2, 3, 2, 1] is a palindrome:
% ?- is_palindrome([1, 2, 3, 2, 1]).
% This will succeed, indicating that the list is a palindrome.

% To check if the list [1, 2, 3, 4] is a palindrome:
% ?- is_palindrome([1, 2, 3, 4]).
% This will fail, indicating that the list is not a palindrome.
```

---

23. Write a PROLOG program to check whether a given number is prime or not. Use this
predicate to print all the prime numbers upto a given number.

```
% Rule to check if a number is prime
is_prime(2).
is_prime(3).
is_prime(N) :-
    N > 3,
    N mod 2 =\= 0,
    \+ has_factor(N, 3).

% Rule to check if a number has a factor
has_factor(N, Factor) :-
    Factor * Factor =< N,
    N mod Factor =:= 0.
has_factor(N, Factor) :-
    NextFactor is Factor + 2,
    has_factor(N, NextFactor).

% Rule to print prime numbers up to a given number
print_primes_up_to(N) :-
    between(2, N, X),
    is_prime(X),
    write(X), write(' '),
    fail. % Backtrack to find more primes
print_primes_up_to(_).
```

```
% Example usage
% To check if 17 is a prime number:
% ?- is_prime(17).
% This will succeed, indicating that 17 is a prime number.

% To print all prime numbers up to 30:
% ?- print_primes_up_to(30).
% This will print: 2 3 5 7 11 13 17 19 23 29
```

---

24. Write a PROLOG program to print the reverse of a given list.

```
% Rule to print the reverse of a list
print_reverse([]).
print_reverse([Head|Tail]) :-
    print_reverse(Tail),
    write(Head), write(' ').

% Example usage
% To print the reverse of the list [1, 2, 3, 4]:
% ?- print_reverse([1, 2, 3, 4]).
% This will print: 4 3 2 1
```

---

25. Write a PROLOG program to delete the kth element of a list.

```
% Rule to delete the kth element of a list
delete_kth(_, K, _, _) :-
    K < 1,  % K must be a positive integer
    write('Invalid value for K. K should be a positive integer.').
delete_kth(Element, 1, [Element|Tail], Tail).
delete_kth(Element, K, [Head|Tail], [Head|Result]) :-
    K > 1,
    K1 is K - 1,
    delete_kth(Element, K1, Tail, Result).

% Example usage
% To delete the 3rd element of the list [1, 2, 3, 4, 5]:
% ?- delete_kth(_, 3, [1, 2, 3, 4, 5], Result).
% Result will be unified with [1, 2, 4, 5].
```

---

26. Write a PROLOG program to check whether a path exists between two nodes of a graph.
You can implement any arbitrary graph. Print the series of the nodes in the path(s) if
some path exists.

```
% Facts defining the directed edges in the graph
edge(a, b).
edge(b, c).
edge(c, d).
edge(d, e).
edge(b, e).
edge(e, f).

% Rule to check if there is a path between two nodes
path(X, Y) :-
   path_helper(X, Y, [X]).

% Base case: There is a path between X and Y if there is a direct edge from X to Y
path_helper(X, Y, _) :- edge(X, Y).

% Recursive case: There is a path between X and Y if there is a direct edge from X to Z,
% and there is a path from Z to Y
path_helper(X, Y, Visited) :-
   edge(X, Z),
   \+ member(Z, Visited),  % Ensure we don't revisit nodes
   path_helper(Z, Y, [Z | Visited]).

% Rule to print the nodes in the path between X and Y
print_path(X, Y) :-
   path(X, Y),
   write('Path between '), write(X), write(' and '), write(Y), write(': '),
   print_path_helper(X, Y, [X]),
   nl.

print_path_helper(X, X, _).
print_path_helper(X, Y, [Next | Rest]) :-
   write(Next), write(' '),
```

```
    print_path_helper(Next, Y, Rest).

% Example usage:
% To check if there is a path between 'a' and 'f' and print the nodes in the path:
% ?- print_path(a, f).
% This will print: "Path between a and f: a b e f"
```