1. Write a Prolog program to solve the 4-Queen problem, where no two queens should be in the same row, column, or diagonal.
2. Write a Prolog program to solve the Traveling Salesman Problem (TSP) by finding the shortest route that visits all cities and returns to the starting city.
3. Write a Prolog program to solve the water jug problem, where you have two jugs with capacities `X` and `Y`, and you want to measure a target amount of water.
4. Write a Prolog program that removes the Nth item from a list.
5. Write a Prolog program that inserts an item as the Nth item into a list.
6. Define a predicate `addLeaf(Tree, X, NewTree)` to insert `X` as a leaf node in a binary search tree (BST).
7. Define a predicate countLists(Alist, Ne, Nl) that counts the number of top-level items in a list and the number of empty lists.
8. Write a predicate `memCount(AList, BList, Count)` that counts the number of occurrences of `AList` in `BList`.
9. Write a Prolog program to find the greatest element in a list.
10. Write a Prolog program to check if an element is in a list.
11. Write a Prolog program to reverse a list.
12. Write a Prolog program to find the sum of the elements in a list.
13. Write a Prolog program to check if a list is sorted.
14. Write a Prolog program to find the minimum element in a list.
15. Write a Prolog program to count how many times an element appears in a list.
16. Write a Prolog program to check if a list is a palindrome.

Solution:

1.

```
% Solve the 4-Queens problem

safe_queen(_, [], _).

safe_queen(X, [Y|Ys], D) :-

   X \= Y,

   abs(X - Y) \= D,

   D1 is D + 1,

   safe_queen(X, Ys, D1).


queens(N, Solution) :-
```

```
    length(Solution, N),
    numlist(1, N, Numbers),
    safe_queen(Numbers, Solution, 1).

% Query
% ?- queens(4, Solution).
```

2.

```
% Define distances between cities
distance(a, b, 10).
distance(a, c, 15).
distance(b, c, 20).
distance(b, d, 25).
distance(c, d, 30).
distance(a, d, 35).

% Compute total distance of a tour
tour_distance([_], 0).
tour_distance([X, Y | Rest], D) :-
    distance(X, Y, D1),
    tour_distance([Y | Rest], D2),
    D is D1 + D2.

% Query for the shortest path
% ?- tour_distance([a, b, c, d, a], D).
```

3.

```
% Define a state as (Amount in Jug 1, Amount in Jug 2)

% Actions: fill jug, empty jug, pour from one jug to the other


% Base case: goal is reached

goal([Target, _], Target).

goal([_, Target], Target).


% Fill a jug

fill([_, Y], [X, Y]) :- X is 5, X =\= Y.

fill([X, _], [X, Y]) :- Y is 3, X =\= Y.


% Query

% ?- goal([0, 0], 4).
```

4.

```
remove_nth(1, [X|Xs], X, Xs).

remove_nth(N, [X|Xs], R, [X|Ys]) :-

   N > 1,

   N1 is N - 1,

   remove_nth(N1, Xs, R, Ys).


% Query

% ?- remove_nth(2, [a, b, c, d], R).
```

5.

```
insert_nth(1, X, Ys, [X|Ys]).

insert_nth(N, X, [Y|Ys], [Y|Zs]) :-
```

```
   N > 1,
   N1 is N - 1,
   insert_nth(N1, X, Ys, Zs).

% Query
% ?- insert_nth(2, x, [a, b, c], L).
```

6.

```
gt(X, Y) :- X > Y.
addLeaf(nil, X, t(X, nil, nil)).
addLeaf(t(Root, Left, Right), X, t(Root, Left, NewRight)) :-
   gt(X, Root),
   addLeaf(Right, X, NewRight).
addLeaf(t(Root, Left, Right), X, t(Root, NewLeft, Right)) :-
   \+ gt(X, Root),
   addLeaf(Left, X, NewLeft).

% Query
% ?- addLeaf(t(10, nil, nil), 5, NewTree).
```

7.

```
countLists([], 0, 0).
countLists([[]|Tail], Ne, Nl) :-
   countLists(Tail, Ne1, Nl1),
   Ne is Ne1 + 1,
   Nl is Nl1 + 1.
countLists([_|Tail], Ne, Nl) :-
```

```
    countLists(Tail, Ne, Nl).

% Query
% ?- countLists([[], [1, 2], [], [3]], Ne, Nl).
```

8.

```
memCount([], _, 0).
memCount([X|Xs], [X|Ys], Count) :-
    memCount(Xs, Ys, Count1),
    Count is Count1 + 1.
memCount([X|Xs], [Y|Ys], Count) :-
    X \= Y,
    memCount([X|Xs], Ys, Count).

% Query
% ?- memCount([a], [a, b, a, c, a], Count).
```

9.

```
greatest([X], X).
greatest([X|Xs], Max) :-
    greatest(Xs, Max1),
    Max is max(X, Max1).

% Query
% ?- greatest([1, 5, 2, 9, 3], Max).
```

10.

```
member(X, [X|_]).
```

```
member(X, [_|Ys]) :- member(X, Ys).


% Query

% ?- member(3, [1, 2, 3, 4]).
```

11.

```
reverse([], []).

reverse([X|Xs], Ys) :-

   reverse(Xs, Zs),

   append(Zs, [X], Ys).


% Query

% ?- reverse([1, 2, 3, 4], Reversed).
```

12.

```
sum([], 0).

sum([X|Xs], Sum) :-

   sum(Xs, Sum1),

   Sum is X + Sum1.


% Query

% ?- sum([1, 2, 3, 4], Sum).
```

13.

```
sorted([]).

sorted([_]).

sorted([X, Y | Ys]) :- X =< Y, sorted([Y | Ys]).
```

```
% Query
% ?- sorted([1, 2, 3, 4]).
```

14.

```
minimum([X], X).
minimum([X|Xs], Min) :-
   minimum(Xs, Min1),
   Min is min(X, Min1).


% Query
% ?- minimum([1, 5, 2, 9, 3], Min).
```

15.

```
count_occurrences([], _, 0).
count_occurrences([X|Xs], X, Count) :-
   count_occurrences(Xs, X, Count1),
   Count is Count1 + 1.
count_occurrences([Y|Xs], X, Count) :-
   X \= Y,
   count_occurrences(Xs, X, Count).


% Query
% ?- count_occurrences([a, b, a, c, a], a, Count).
```

16.

```
palindrome([]).
palindrome([_]).
palindrome([X|Xs]) :-
```

```
    append(Mid, [X], Xs),
    palindrome(Mid).

% Query
% ?- palindrome([a, b, a]).
```