1. `member/2`: Verifică dacă un element aparține unei liste.
```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).
```

2. `append/3`: Concatenează două liste.
```
append([], List, List).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

3. `length/2`: Calculează lungimea unei liste.
```
length([], 0).
length([_|T], N) :- length(T, N1), N is N1 + 1.
```

4. `reverse/2`: Inversează ordinea elementelor unei liste.
```
reverse(List, Reversed) :- reverseAcc(List, [], Reversed).
reverseAcc([], Acc, Acc).
reverseAcc([H|T], Acc, Reversed) :- reverseAcc(T, [H|Acc], Reversed).
```

5. `sum_list/2`: Calculează suma elementelor unei liste.
```
sum_list([], 0).
sum_list([X|T], Sum) :- sum_list(T, Rest), Sum is X + Rest.
```

6. `max_list/2`: Găsește valoarea maximă dintr-o listă.
```
max_list([X], X).
max_list([H|T], Max) :- max_list(T, RestMax), Max is max(H, RestMax).
```

7. `min_list/2`: Găsește valoarea minimă dintr-o listă.
```
min_list([X], X).
min_list([H|T], Min) :- min_list(T, RestMin), Min is min(H, RestMin).
```

8. `sort/2`: Sortează o listă în ordine crescătoare.
```
sort(List, Sorted) :- predsort(@=<, List, Sorted).
```

9. `delete/3`: Elimină prima apariție a unui element dintr-o listă.
```
delete(X, [X|T], T).
delete(X, [H|T], [H|Result]) :- delete(X, T, Result).
```

10. `select/3`: Elimină prima apariție a unui element dintr-o listă și întoarce restul listei.
```
select(X, [X|T], T).
select(X, [H|T], [H|Result]) :- select(X, T, Result).
```

11. `nth0/3`: Accesează al N-lea element dintr-o listă (indexare de la 0).
```
nth0(0, [X|_], X).
nth0(N, [_|T], X) :- N > 0, N1 is N - 1, nth0(N1, T, X).
```

12. `nth1/3`: Accesează al N-lea element dintr-o listă (indexare de la 1).
```
nth1(1, [X|_], X).
nth1(N, [_|T], X) :- N > 1, N1 is N - 1, nth1(N1, T, X).
```

13. `sublist/2`: Verific
ă dacă o listă este sublista unei alte liste.
```
sublist(Sublist, List) :- append(_, Rest, List), append(Sublist, _, Rest).
```

14. `subset/2`: Verifică dacă o listă este un submulțime a unei alte liste.
```
subset([], _).
subset([X|T], Set) :- member(X, Set), subset(T, Set).
```

15. `flatten/2`: Transformă o listă multidimensională într-o listă unidimensională.
```
flatten(List, Flattened) :- flattenAcc(List, [], Flattened).
flattenAcc([], Acc, Acc).
flattenAcc([H|T], Acc, Flattened) :- flattenAcc(T, Acc1, Flattened1), append(H, Flattened1, Acc1).
```

16. `zip/3`: Creează o listă de perechi folosind elemente din două liste.
```
zip([], [], []).
zip([X|T1], [Y|T2], [(X,Y)|Zipped]) :- zip(T1, T2, Zipped).
```

17. `maplist/2`: Aplică un predicat asupra fiecărui element al unei liste.
```
maplist(_, []).
maplist(Pred, [X|Xs]) :- call(Pred, X), maplist(Pred, Xs).
```

18. `filter/3`: Filtrează elementele unei liste pe baza unui predicat.
```
filter(_, [], []).
filter(Pred, [X|Xs], Filtered) :- (call(Pred, X) -> Filtered = [X|Rest] ; Filtered = Rest), filter(Pred, Xs, Rest).
```

19. `partition/4`: Separă elementele unei liste în două liste, pe baza unui predicat.
```
partition(_, [], [], []).
partition(Pred, [X|Xs], [X|True], False) :- call(Pred, X), partition(Pred, Xs, True, False).
partition(Pred, [X|Xs], True, [X|False]) :- \+ call(Pred, X), partition(Pred, Xs, True, False).
```

20. `findall/3`: Colectează toate soluțiile unui predicat într-o listă.
```
findall(X, Goal, List) :- bagof(X, Goal, List), !.
findall(_, _, []).
```

21. `forall/2`: Verifică dacă un predicat este adevărat pentru toate soluțiile date de un alt predicat.
```
forall(Condition, Action) :- \+ (Condition, \+ Action).
```

22. `once/1`: Caută o singură soluție a unui predicat, ignorând alte posibile soluții.
```
once(Goal) :- Goal, !.
```

23. `between/3`: Generează numere întregi într-un interval specificat.
```
between(Min, Max, N) :- Min =< Max, (Min = N ; (Min1 is Min+1, between(Min1, Max, N))).
```

24. `atom/1`: Verifică dacă un termen este un atom.
```
atom(X)
 :- atomic(X).
```

25. `number/1`: Verifică dacă un termen este un număr.
```
number(X) :- atomic(X), atom_number(X, _).
```

26. `atomic_list_concat/2`: Concatenează o listă de atomi într-un atom.
```
atomic_list_concat(List, Atom) :- atomic_list_concat(List, '', Atom).
```

27. `atomic_concat/3`: Concatenează două atomi într-un alt atom.
```
atomic_concat(Atom1, Atom2, Concat) :- atom_concat(Atom1, Atom2, Concat).
```

28. `atomic_length/2`: Calculează lungimea unui atom.
```
atomic_length(Atom, Length) :- atom_length(Atom, Length).
```

29. `call/1`: Execută un termen ca un apel la un predicat.
```
call(Term) :- call(Term).
```

30. `ground/1`: Verifică dacă un termen este instanțiat în întregime.
```
ground(X) :- \+ var(X), X \= [], X \= [_|_], X \= (_,_).
```

1. Predicatul `even_list(+List)` care verifică dacă toate elementele unei liste sunt numere pare:
```
even_list([]).
even_list([H|T]) :- 0 is H mod 2, even_list(T).
```

2. Predicatul `merge_lists(+List1, +List2, -Merged)` care combină două liste într-una singură, prin intercalarea elementelor:
```
merge_lists([], List, List).
merge_lists(List, [], List).
merge_lists([H1|T1], [H2|T2], [H1,H2|Merged]) :- merge_lists(T1, T2, Merged).
```

3. Predicatul `palindrome(+List)` care verifică dacă o listă este palindrom (se citește la fel în ambele sensuri):
```
palindrome(List) :- reverse(List, List).
```

4. Predicatul `remove_duplicates(+List, -WithoutDuplicates)` care elimină duplicații dintr-o listă:
```
remove_duplicates([], []).
remove_duplicates([H|T], WithoutDuplicates) :- member(H, T), !, remove_duplicates(T, WithoutDuplicates).
remove_duplicates([H|T], [H|WithoutDuplicates]) :- remove_duplicates(T, WithoutDuplicates).
```

5. Predicatul `sublist(+Sublist, +List)` care verifică dacă `Sublist` este o sublistă a lui `List`:
```
sublist(Sublist, List) :- append(_, Sublist, Suffix), append(Suffix, _, List).
```

6. Predicatul `sum_list(+List, -Sum)` care calculează suma elementelor unei liste:
```
sum_list([], 0).
sum_list([H|T], Sum) :- sum_list(T, Rest), Sum is H + Rest.
```

7. Predicatul `last_element(+List, -Last)` care găsește ultimul element al unei liste:
```
last_element([X], X).
last_element([_|T], Last) :- last_element(T, Last).
```

8. Predicatul `even_odd_list(+List, -Even, -Odd)` care separă elementele pare și impare dintr-o listă în două liste separate:
```
even_odd_list([], [], []).
even_odd_list([H|T], [H|Even], Odd) :- 0 is H mod 2, even_odd_list(T, Even, Odd).
even_odd_list([H|T], Even, [H|Odd]) :- 1 is H mod 2, even_odd_list(T, Even, Odd).
```

9. Predicatul `flatten_list(+List, -Flattened)` care aplanează o listă multidimensională într-o listă unidimensională:
```
flatten_list([], []).
flatten_list([H|T], Flattened) :- is_list(H), flatten_list(H, FlatH), flatten_list(T, FlatT), append(FlatH, FlatT, Flattened).
flatten_list([H|T], [H|Flattened]) :- \+ is_list(H), flatten_list(T, Flattened).
```

10. Predicatul `rotate_list(+List, +N, -Rotated)` care rotește o listă la dreapta de N ori:
```
rotate_list(List, 0, List).
rotate_list([H|T], N, Rotated) :- N > 0, append(T, [H], NewList), N1 is N - 1, rotate_list(NewList, N1, Rotated).
```

11. Predicatul `max_list(+List, -Max)` care găsește valoarea maximă dintr-o listă:
```
max_list([X], X).
max_list([H|T], Max) :- max_list(T, RestMax), Max is max(H, RestMax).
```

12. Predicatul `delete_element(+Element, +List, -Result)` care șterge toate aparițiile unui element dintr-o listă:
```
delete_element(_, [], []).
delete_element(X, [X|T], Result) :- delete_element(X, T, Result).
delete_element(X, [H|T], [H|Result]) :- X \= H, delete_element(X, T, Result).
```

13. Predicatul `split_list(+List, +N, -Prefix, -Suffix)` care împarte o listă în două părți: prefixul de lungime N și sufixul:
```
split_list(List, N, Prefix, Suffix) :- length(Prefix, N), append(Prefix, Suffix, List).
```

14. Predicatul `is_sorted(+List)` care verifică dacă o listă este sortată în ordine crescătoare:
```
is_sorted([]).
is_sorted([_]).
is_sorted([X
,Y|T]) :- X =< Y, is_sorted([Y|T]).
```

15. Predicatul `intersection_list(+List1, +List2, -Intersection)` care găsește intersecția a două liste (elementele comune):
```
intersection_list([], _, []).
intersection_list([H|T], List2, [H|Intersection]) :- member(H, List2), intersection_list(T, List2, Intersection).
intersection_list([_|T], List2, Intersection) :- intersection_list(T, List2, Intersection).
```

16. Predicatul `alternating_list(+List, -Alternating)` care construiește o listă formată din elementele alternative dintr-o listă:
```
alternating_list([], []).
alternating_list([X], [X]).
alternating_list([X,_|T], [X|Alternating]) :- alternating_list(T, Alternating).
```

17. Predicatul `duplicate_list(+List, -Duplicated)` care dublează fiecare element dintr-o listă:
```
duplicate_list([], []).
duplicate_list([X|T], [X,X|Duplicated]) :- duplicate_list(T, Duplicated).
```

18. Predicatul `merge_sort(+List, -Sorted)` care sortează o listă folosind algoritmul Merge Sort:
```
merge_sort([], []).
merge_sort([X], [X]).
merge_sort(List, Sorted) :- divide_list(List, Left, Right), merge_sort(Left, SortedLeft), merge_sort(Right, SortedRight), merge_lists(SortedLeft, SortedRight, Sorted).
divide_list(List, Left, Right) :- length(List, Len), HalfLen is Len // 2, length(Left, HalfLen), append(Left, Right, List).
merge_lists([], List, List).
merge_lists(List, [], List).
merge_lists([X|T1], [Y|T2], [X|Sorted]) :- X =< Y, merge_lists(T1, [Y|T2], Sorted).
merge_lists([X|T1], [Y|T2], [Y|Sorted]) :- X > Y, merge_lists([X|T1], T2, Sorted).
```

19. Predicatul `insert_sorted_list(+Element, +List, -Sorted)` care inserează un element într-o listă sortată, menținând ordinea:
```
insert_sorted_list(X, [], [X]).
insert_sorted_list(X, [H|T], [X,H|T]) :- X =< H.
insert_sorted_list(X, [H|T], [H|Sorted]) :- X > H, insert_sorted_list(X, T, Sorted).
```

20. Predicatul `remove_nth(+N, +List, -Result)` care elimină al N-lea element dintr-o listă:

```
remove_nth(_, [], []).
remove_nth(1, [_|T], T).
remove_nth(N, [H|T], [H|Result]) :- N > 1, N1 is N - 1,
remove_nth(N1, T, Result).
```