9. Construiți în Prolog un predicat `up(+L, -LUp)` care produce în `LUp` o listă cu următoarele proprietăți:
   - primul element din `LUp` este același cu primul element din lista `L`
   - următorul element din `LUp` este următorul element din `L`, mai mare decât primul.
   - următorul element din `LUp` este următorul element din `L`, mai mare decât anteriorul din `LUp`, etc.

   Exemplu: `up([5, 6, 3, 4, 8, 5, 9, 2], LUp)` leagă `LUp` la `[5, 6, 8, 9]`.
   Soluție:

```
% up(+L, -LUp)
up([H|T], [H|TUp]) :- upAux(H, T, TUp).
upAux(_, [], []).
% H face parte din secvența crescătoare:
upAux(CurrentMax, [H|T], [H|TUp]) :- H > CurrentMax, upAux(H, T, TUp).
% H nu face parte din secvența crescătoare:
upAux(CurrentMax, [H|T], TUp) :- H =< CurrentMax, upAux(CurrentMax, T, TUp).
```

   Alternativ, pot evita comparația din ultima regulă, dacă folosesc `!` după comparația din penultima regulă.

9. Construiți în Prolog un predicat `dn(+L, -LDown)` care produce în `LDown` o listă cu următoarele proprietăți:
   - primul element din `LDown` este același cu primul element din lista `L`
   - următorul element din `LDown` este următorul element din `L`, mai mic decât primul.
   - următorul element din `LDown` este următorul element din `L`, mai mic decât anteriorul din `LDown`, etc.

   Exemplu: `dn([5, 6, 3, 4, 8, 5, 9, 2], LDown)` leagă `LDown` la `[5, 3, 2]`.
   Soluție:

```
% dn(+L, -LDown)
dn([H|T], [H|TDown]) :- dnAux(H, T, TDown).
dnAux(_, [], []).
% H face parte din secvența crescătoare:
dnAux(CurrentMin, [H|T], [H|TDown]) :- H < CurrentMin, dnAux(H, T, TDown).
% H nu face parte din secvența crescătoare:
dnAux(CurrentMin, [H|T], TDown) :- H >= CurrentMin, dnAux(CurrentMin, T, TDown).
```

   Alternativ, pot evita comparația din ultima regulă, dacă folosesc `!` după comparația din penultima regulă.

8. Implementați predicatul `filtersorted(+LLIn, -LLOut)`, care primește în `LLIn` o listă de liste de numere și pune în `LLOut` doar acele liste de numere care sunt sortate crescător. De exemplu, `filtersorted([[2, 1], [1, 3], [1, 2, 3, 4], [1, 2, 4, 2]], X)` leagă `X` la `[[1, 3], [1, 2, 3, 4]]`. Explicați cum funcționează implementarea.

   Soluție:

```
filtersorted([], []).
filtersorted([E|LLIn], [E|LLOut]) :- sort(E, E), !, filtersorted(LLIn, LLOut).
filtersorted([_|LLIn], LLOut) :- filtersorted(LLIn, LLOut).
```

9. Folosiți unul sau mai multe dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `med(+L, -Med)`, care găsește elementul median al listei, cu proprietatea că numărul de elemente mai mari decât `Med` este diferit cu cel mult 1 de numărul de elemente mai mici decât `Med`). Nu utilizați recursivitate explicită. Explicați cum funcționează soluția.

   Soluție:

```
med(L, Med) :- member(Med, L), findall(X, (member(X, L), X < Med), L1),
               findall(X, (member(X, L), X > Med), L2), length(L1, LL1),
               length(L2, LL2), abs(LL1-LL2) =< 1.
```

8. Implementați predicatul `gen(+Sample, +Length, -Output)`, care produce în `Output` o listă de lungime `Length` care constă din repetări ale listei `Sample`.

   Exemplu: `g([1, 2, 3], 10, X)` leagă `X` la `[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]`.

   Explicați cum funcționează implementarea.

   Soluție:

```
g(_, 0, []).
g([E|T], Len, [E|R]) :- Len > 0, Len1 is Len - 1, append(T, [E], T1), g(T1, Len1, R).
```

9. Folosiți o singură dată unul dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `minmax(+L, -Min, -Max)` care leagă `Min`, respectiv `Max`, la elementul minim, respectiv maxim, al listei. Nu utilizați recursivitate explicită. Explicați cum funcționează soluția.

   Soluție:

```
minmax(L, Min, Max) :- member(Min, L), member(Max, L),
                       forall(member(X, L), (X >= Min, X =< Max)).
```

8. Implementați predicatul `a(+L1, +L2, +L3, -L)` care primește 3 liste și produce în `L` o listă de tripluri cu elementele ce corespund din cele 3 liste. Lista `L` are lungimea celei mai scurte liste dintre `L1`, `L2` și `L3`. Exemplu: `a([1, 2], [a, b, c], [x, y, z, t], X)` leagă `X` la `[(1, a, x), (2, b, y)]`.

   Explicați cum funcționează implementarea.

   Soluție:

```
a([], _, _, []).
a(_, [], _, []).
a(_, _, [], []).
a([H1|L1], [H2|L2], [H3|L3], [(H1, H2, H3) | L123]) :- a(L1, L2, L3, L123).
```

9. Folosiți unul sau mai multe dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `secondMin(+L, -X)` care găsește al doilea cel mai mic element din lista `L` (care conține cel puțin 2 elemente și nu conține duplicate). Nu utilizați recursivitate explicită. Explicați cum funcționează soluția.

   Soluție:

```
secondMin(L, X) :- member(X, L), findall(Y, (member(Y, L), Y < X), [_]).
```

8. Implementați predicatul `mul(+Lin, -Lout)` care primește o listă de numere ca prim argument și leagă al doilea argument la lista acelor numere din prima listă care sunt divizibile cu toate celelalte numere care le urmează.

   Exemplu: `mul([24, 4, 12, 6, 3, 2], X)` leagă `X` la `[24, 12, 6, 2]`.

   Explicați cum funcționează implementarea.

Soluție:

```
check(_, []).
check(X, [H|T]) :- mod(X, H) =:= 0, check(X, T).
mul([], []).
mul([H|LIn], [H|LOut]) :- check(H, LIn), !, mul(LIn, LOut).
mul([_|LIn], LOut) :- mul(LIn, LOut).
```

9. Folosiți unul sau mai multe dintre predicatele `findall`, `forall`, `bagof`, `setof` pentru a implementa predicatul `secondMax(+L, -X)` care găsește al doilea cel mai mare element din lista `L` (care conține cel puțin 2 elemente și nu conține duplicate). Nu utilizați recursivitate explicită. Explicați cum funcționează soluția.

Soluție:

```
secondMin(L, X) :- member(X, L), findall(Y, (member(Y, L), Y < X), [_]).
```

8. Se dă programul Prolog:

```
p(R, S) :- member(X, R),
        findall(Y, (member(Y, R), Y \= X), T), !, q(X, T, S).
q(X, A, [X|A]). q(X, [A|B], [A|C]) :- q(X, B, C).
```

Dacă predicatul `p` primește în primul argument o listă, la ce valori leagă al doilea argument? Câte soluții are interogarea `p([1, 2, 3, 4], S)` ?

Soluție:

Ia primul element (și elimină duplicatele lui) și îl pune pe diverse poziții ale listei, inclusiv pe prima. Patru soluții: `[1, 2, 3, 4]`, `[2, 1, 3, 4]`, `[2, 3, 1, 4]`, `[2, 3, 4, 1]`

8. Se dă programul Prolog:

```
p(_, [], []).
p(A, [A|B], B) :- !.
p(A, [B|C], [B|D]) :- p(A, C, D).
```

Ce relație există între cele 3 valori `X`, `Y`, `Z`, dacă `p(X, Y, Z)` este adevărat?

Soluție:

Este predicatul `select`, iar dacă primul argument este nelegat face `select` la primul element. Predicatul `select(X, Y, Z)` este adevărat dacă `X` este un element din lista `Y`, iar `Z` este exact lista `Y`, în afară de elementul `X`.

8. Scrieți un predicat Prolog `diff(A, B, R)` care leagă R la diferența mulțimilor (reprezentate ca liste) A și B.

Soluție:

```
intersect(A, B, R) :- findall(X, (member(X, A), member(X, B)), R).
```

8. Scrieți un predicat Prolog `intersect(A, B, R)` care leagă R la intersecția mulțimilor (reprezentate ca liste) A și B.

Soluție:

```
diff(A, B, R) :- findall(X, (member(X, A), \+ member(X, B)), R).
```

7. Implementați în Prolog predicatul `x(L, A, B, N)` care determină, pentru o listă `L`, numărul `N` de elemente care sunt mai mari decât `A` și mai mici decât `B`. Nu folosiți recursivitate explicită.

Soluție:

```
x(L, A, B, N) :- findall(X, (member(X, L), X > A, X < B), S), length(S, N).
```

7. Implementați în Prolog predicatul `x(L, A, B, N)` care determină, pentru o listă `L`, numărul `N` de elemente care nu sunt mai mari decât `A` și mai mici decât `B`. Nu folosiți recursivitate explicită.

Soluție:

```
x(L, A, B, N) :- findall(X, (member(X, L), (X < A; X > B)), S), length(S, N).
```

7. Implementați în Prolog predicatul `x(L, M)` care determină, pentru o listă `L`, `M`, maximul listei. Nu folosiți recursivitate explicită.

Soluție:

```
x(L, M) :- member(M, L), forall(member(E, L), X > E).
```

7. Implementați în Prolog predicatul `x(L, M)` care determină, pentru o listă `L`, `M`, minimul listei. Nu folosiți recursivitate explicită.

Soluție:

```
x(L, M) :- member(M, L), forall(member(E, L), X < E).
```

10. Care este efectul aplicării predicatului `p` asupra listelor `L1` și `L2` (la ce este legat argumentul `R` în apelul `p(L1, L2, R)` ?):
```
p(A, [], A). p(A, [E|T], [E|R]) :- p(A, T, R).
```
Soluție:
```
R = L2 ++ L1
```

10. Care este efectul aplicării predicatului `p` asupra listelor `L1` și `L2` (la ce este legat argumentul `R` în apelul `p(L1, L2, R)` ?):
```
p([], A, A). p([E|T], A, [E|R]) :- p(T, A, R).
```
Soluție:
```
R = L1 ++ L2
```

10. Implementați în Prolog un predicat având semnătura `mapF(+L, -LO)`, care să fie echivalent cu expresia Haskell `(map f)`, știind că există deja definit un predicat `f(+X, -XO)`.
Soluție:
mapF(L, LO):- findall(XO, (member(X, L), f(X, XO)), LO).

11. Câte soluții are interogarea `p(L, [1, 2, 3])` în condițiile în care avem definiția de mai jos? Ce formă au aceste soluții?
```
p(D, [A, B, C]) :- member(A, D), member(B, D), member(C, D).
```
Soluție:
O infinitate. Soluțiile sunt liste care conțin 1, 2 și 3 (în ordine) și un număr ≥ 0 de elemente neinstanțiate.

10. Implementați în Prolog un predicat având semnătura `filterF(+L, -LO)`, care să fie echivalent cu expresia Haskell `(filter f)`, știind că există deja definit un predicat `f(+X)`.
Soluție:
filterF(L, LO):- findall(X, (member(X, L), f(X)), LO).

11. Câte soluții are interogarea `p([1, 2, 3], L)` în condițiile în care avem definiția de mai jos? Ce formă au aceste soluții?
```
p(D, [A, B, C]) :- member(A, D), member(B, D), member(C, D).
```
Soluție:
27. Sunt toate combinațiile de 1,2,3, inclusiv în care un element apare de mai multe ori.

10. Ce rezultat are în Prolog evaluarea lui `p(L, X)`, cu `L` o listă și `X` nelegat:
```
r([], _).
r([H|T], X) :- member(H, X), r(T, X).
p(L, X) :- length(L, N), length(X, N), r(L, X).
```

Soluție:
Lista X are aceeași lungime ca și L și aceeași membri, în orice ordine (diversele soluții pentru X sunt permutările listei L).

11. Scrieți un predicat Prolog `up` (și eventual predicate ajutătoare) care identifică secvențele (cel puțin două elemente) strict crescătoare dintr-o listă. Exemplu:
`up([5, 1, 2, 3, 2, 3, 1, 1, 0, 9, 10], LS)` → `LS = [1, 2, 3, 2, 3, 0, 9, 10]`

Soluție:
```
up([], []).
up([H, H1 | T], [H, H1 | LS]) :- H1 > H, !, up(T, H1, LS).
up([_ | T], LS) :- up(T, LS).
up([], _, []) :- !.
up([H | T], E, [H | LS]) :- H > E, !, up(T, H, LS).
up(L, _, LS) :- up(L, LS).
```

10. Scrieți un predicat Prolog `down` (și eventual predicate ajutătoare) care identifică secvențele (cel puțin două elemente) strict descrescătoare dintr-o listă. Exemplu:
`down([5, 1, 2, 3, 2, 1, 1, 1, 10, 9, 10], LS)` → `LS = [5, 1, 3, 2, 1, 10, 9]`

Soluție:
```
down([], []).
down([H, H1 | T], [H, H1 | LS]) :- H1 < H, !, down(T, H1, LS).
down([_ | T], LS) :- down(T, LS).
down([], _, []) :- !.
down([H | T], E, [H | LS]) :- H < E, !, down(T, H, LS).
down(L, _, LS) :- down(L, LS).
```

11. Ce rezultat are în Prolog evaluarea lui `transformer(L, X)`, cu `L` o listă și `X` nelegat:
```
processor([], _).
processor([H|T], X) :- member(H, X), processor(T, X).
transformer(L, X) :- length(L, N), length(X, N), processor(L, X).
```

Soluție:
Lista X are aceeași lungime ca și L și aceeași membri, în orice ordine (diversele soluții pentru X sunt permutările listei L).

9. Scrieți predicatul `filter(+L, +T, -LF)` în Prolog care ia o listă și o filtrează, întorcând în `LF` doar acele elemente mai mari (strict) decât pragul `T`, fără a utiliza recursivitate explicită. Exemplu: `filter([1,5,9,3,2,6,3,8], 5, LF)` leagă `LF` la `[9, 6, 8]`.

Soluție:
```
filter(L, T, LF) :- findall(X, (member(X, L), X > T), LF).
```

10. Realizați un predicat Prolog care elimină duplicatele dintr-o listă.

Soluție:
```
 nodups([], []).
nodups([H|T], [H|LO]) :- findall(X, (member(X, T), X \= H), T1),
nodups(T1, LO).
```
sau
```
rem(_, [], []).
rem(X, [X|L], LO) :- rem(X, L, LO), !.
rem(X, [H|L], [H|LO]) :- rem(X, L, LO).
nodups([], []).
nodups([H|T], [H|LO]) :- rem(H, T, T1), nodups(T1, LO).
```