```
% primitives used

member( X, [X | Tail] ).
member( X, [Head | Tail] ) :-
  member( X, Tail).

conc( [], L, L).
conc( [X | L1], L2, [X | L3]) :-
  conc( L1, L2, L3).

sublist( S, L) :-
  conc( L1, L2, L),
  conc( S, L3, L2).

del( X, [X | Tail], Tail).
del( X, [Y | Tail], [Y | Tail1]) :-
  del( X, Tail, Tail1).

%
% Exercise 1
%

range(N,N,[N]):-!.
range(N,N,[M|Tail]):-
        M < N,
        M1 is M+1,
        range(M1,N,Tail).

%test: range(1,10,L);

%
% Exercise 2
%

split([], [], []).
split([X | Tail], [X | Pos], Neg) :-
        X >= 0, !,
        split(Tail, Pos, Neg).
split([X | Tail], Pos, [X | Neg]) :-
        split(Tail, Pos, Neg).

%test: split([3,-6,5,0,-2], Pos, Neg).

%
% Exercise 3
%
```

```
disjoint(L1,L2) :-
        not((member(X,L1), member(X,L2))).

%test: disjoint([1,3,5,7],[2,4,6,8]).
%test: disjoint([1,3,5,7],[2,4,5,8]).

%
% Exercise 4
%

ackermann(0, N, Res) :-
        !, Res is N + 1.
achermann(M, 0, Res) :-
        !, N1 is M - 1.
ackermann(M,N,Res) :-
        N1 is N-1,
        ackermann(M,N1,Tmp),
        M1 is N-1,
        ackermann(M1,Tmp,Res).

%test: ackermann(0,3,Res).
%test: ackermann(3,0,Res).
%test: ackermann(3,3,Res).

%
% Exercise 5
%

flatten([],[]).
flatten([H|T],[H|FT]) :-
        atomic(H), !,
        flatten(T,FT).
flatten([H|T], Flat) :-
        flatten(H,FH),
        flatten(T,FT),
        conc(FH,FT,Flat).

%test: flatten([[a],[b],[c,a],[[b]],Flat).

%
% Exercise 6
%

subsum(L,Sum,L1) :-
        subset(L1,L),
      ordered(L1),
        sum(L1,Sum).
```

```
subset([], L).
subset([X|T],L) :-
        member(X,L),
        del(X,L,L2),
        subset(T,L2).

sum([X],X).
sum([X1,X2|Tail],S) :-
        sum([X2|Tail], S2),
        S is S2 + X1.

ordered([X]).
ordered([First,Second|Tail]) :-
  First =< Second,
  ordered([Second|Tail]).

%test: subsum([1,2,10,5,7,3], 10, Sub).

%
% Exercise 7
%


strangeSort([],[]) :-!.
strangeSort(L1,L2) :-
        minList(L1,L2).

max([Max],Max).
max([X,Y|Rest], Max) :-
        X > Y, !,
        max([X|Rest], Max).
max([X,Y|Rest], Max) :-
        max([Y|Rest], Max).

min([Min],Min).
min([X,Y|Rest], Min) :-
        X < Y, !,
        min([X|Rest], Min).
min([X,Y|Rest], Min) :-
        min([Y|Rest], Min).

maxList([],[]) :- !.
maxList(L1,[Max|MinList]) :-
        max(L1,Max),
        del(Max,L1,L2),
        minList(L2,MinList).
```

```
minList([],[]) :- !.
minList(L1,[Min|MaxList]) :-
        min(L1,Min),
        del(Min,L1,L2),
        maxList(L2,MaxList).

%test: strangeSort([3,6,1,2,4,7,5], L).

%
% Exercise 8
%

uniqueMerge(L,[],L).
uniqueMerge([],L,L).

uniqueMerge([MH|T1], L2, [MH|MT]) :-
        uniqueMerge(T1, L2, [MH|MT]), !.

uniqueMerge(L1, [MH|T2], [MH|MT]) :-
        uniqueMerge(L1, T2, [MH|MT]), !.

uniqueMerge([H1|T1], [H2|T2], [H1|Rest]) :-
        H1 < H2, !,
        uniqueMerge(T1, [H2|T2], Rest).

uniqueMerge([H1|T1], [H2|T2], [H2|Rest]) :- !,
        uniqueMerge([H1|T1], T2, Rest).

%test: uniqueMerge([2,3,5,7,8], [1,2,5,6,8], L).

%
% Exercise 9
%

binary_tree(void).
binary_tree(tree(X,L,R)) :-
        binary_tree(L),
        binary_tree(R).

tree_member(X,tree(X,_,_)).
tree_member(X,tree(_,L,R) :-
        tree_member(X,L);
        tree_member(X,R).

isotree(void,void).
isotree(tree(X,L1,R1), tree(X,L2,R2)) :-
```

```
        (
          isotree(L1,L2),
          isotree(R1,R2)
        );
        (
          isotree(L1,R2),
          isotree(R1,L2)
        ).

%test: binary_tree(tree(a, tree(b, void, void), tree(c, tree(d, void, void), tree(e, void, void)))).
%test: tree_member(X, tree(a, tree(b, void, void), tree(c, tree(d, void, void), tree(e, void,
void)))).
%test: isotree(tree(a, tree(b, void, void), tree(c, tree(d, void, void), tree(e, void, void))),
tree(a, tree(c, tree(d, void, void), tree(e, void, void)), tree(b, void, void))).

%
% Exercise 10
%

path(A, Z, Graph, Path) :-
        path1(A, [Z], Graph, Path).
path1(A, [A | Path1], _, [A | Path1]).
path1(A, [Y | Path1], Graph, Path) :-
        adjacent(X, Y, Graph),
        not member(X, Path1),
        path1(A, [X, Y | Path1], Graph, Path).

adjacent(Node1, Node2, Graph) :-
        member(Node1-Node2, Graph)
        ;
        member(Node2-Node1, Graph).

node(Node, Graph) :-
        adjacent(Node, _, Graph).

cycle(Cycle, Graph) :-
        path(X,Y,Graph,[E1,E2|Rest]),
        conc(_,[Y],Rest),
        adjacent(X,Y,Graph),
        conc([E1,E2|Rest],[X],Cycle).

cycles_of(Graph, Cycles) :-
        setof(Cycle, cycle(Cycle,Graph), Cycles).

%test: cycles_of([a-b, b-c, c-a, b-d, d-c], Cycles).
```