**Question 1.**

A.

(i) ?- member(0, [1,2]).  
False as 0 is not a member of the list [1,2]

(ii) ?- member(X, X).

X = [X|\_...]

Succeeds in any case where X is a list which contains itself, but this is never possible and the X’s will never unify, prolog searches forever i think

(iii) ?- setof(X, member(X, X), L).

Setof returns the sorted list L (no duplicates) of the possibilities where X satisfies the predicate member(X, X). Infinite loop as prolog keeps searching when there are no possibilities where the predicate succeeds.

(iv) ?- findall(X, \+member(X, X), L).

Findall gets all possibilities where X satisfies the predicate \+member(X, X) returns result in list L

L = [] ??? -- I think this means that no X satisfies the predicate bc of the infinite loop therefore L is empty.

(v) ?- number(X)

Checks if X is a number, but X is a variable which is not instantiated, response is false

(vi) [a|[b,c]] = .(a,.(b,X)).

Error as cannot assign value to a non variable

B.

An anonymous variable is a variable which is independent, its value is different every time, which means the value is basically ignored. A singleton variable is a variable which is only referred to once. Therefore an anonymous variable is also a singleton variable as it is only referred to once, as it is always independent.

C.

lessSome(List1, List2) :-

allNumeral(List1),

allNumeral(List2),

min(List1, Min),

max(List2, Max),

Min < Max.

allNumeral([H]) :-

number(H), !.

allNumeral([H|T]) :-

number(H),

allNumeral(T).

maxAcc([], A, A).

maxAcc([H|T], Acc, Max) :-

H > Acc,

maxAcc(T, H, Max).

maxAcc([H|T], Acc, Max) :-

Acc >= H,

maxAcc(T, Acc, Max).

max([H|T], Max) :-

maxAcc(T, H, Max), !.

minAcc([], A, A).

minAcc([H|T], Acc, Min) :-

Acc < H,

minAcc(T, Acc, Min).

minAcc([H|T], Acc, Min) :-

Acc >= H,

minAcc(T, H, Min).

min([H|T], Min) :-

minAcc(T, H, Min), !.

D.

Using max/maxAcc, allNumeral from above

lessAll(List1, List2) :-

allNumeral(List1),

allNumeral(List2),

max(List1, List1Max),

lessThanAll(List2, List1Max).

lessThanAll([H], List1Max) :-

H > List1Max, !.

lessThanAll([H|T], List1Max) :-

H > List1Max,

lessThanAll(T, List1Max).

E.

(i)

append([], L, L).

append([H|L1], L2, [H|L3]) :-

append(L1, L2, L3).

(ii) union1 gets the non repeating possibilities X where X is a member of L1 or X is a member of L2. List is returned in L3

Union2 does the same but iterates over returned list U and returns in L3

??

Union1 -> L3 is a sorted union of L1 & L2 e.g union1([1,2], [1,3,4], [1,3,2,4]). Will fail

Union2 -> L3 is an unsorted union of L1 & L2 e.g union2([1,2], [0, 5], [1,2,0,5]) will be true

But because union2 runs set of with member(X,L3), aswell as L1 & L2 all 3 lists need to be given else it runs out of stack

(iii) ??? maybe p(List):- var(List). %if L3 is a variable use union1 and never union2 (see cut)

**Question 2.**

A.

sumAcc([], A, A).

sumAcc([H|T], Acc, X) :-

NewAcc is Acc + H,

sumAcc(T, NewAcc, X).

sum([], 0).

sum([H|T], X) :-

sumAcc([H|T], 0, X).

B.

A tail recursive predicate is a recursive predicate that has no goals left on the stack once it has gotten to the base case, result has been fully calculated when the base case is met

lengthAcc([], A, A).

lengthAcc([\_|T], Acc, N) :-

NewAcc is Acc + 1,

lengthAcc(T, NewAcc, N).

length([], 0).

length(List, N) :-

lengthAcc(List, 0, N).

C.

split(Number, List, Small, Big) :-

member(Number, List),

allNumeral(List),

addToList(Number, List, [], Small, [], Big), !.

addToList(\_, [], Small, Small, Big, Big).

addToList(Number, [H|T], SmallAcc, Small, BigAcc, Big) :-

H > Number,

append([H], BigAcc, NewBigAcc),

addToList(Number, T, SmallAcc, Small, NewBigAcc, Big), !.

addToList(Number, [H|T], SmallAcc, Small, BigAcc, Big) :-

H < Number,

append([H], SmallAcc, NewSmallAcc),

addToList(Number, T, NewSmallAcc, Small, BigAcc, Big), !.

addToList(Number, [H|T], SmallAcc, Small, BigAcc, Big) :-

H =:= Number,

addToList(Number, T, SmallAcc, Small, BigAcc, Big), !.

allNumeral([H]) :-

number(H), !.

allNumeral([H|T]) :-

number(H),

allNumeral(T).

D.

median(List, Median) :-

length(List, Len),

mod(Len, 2) =:= 1, % checking length of list is odd

setof(X, member(X, List), NoDups), % setof removes any duplicates

length(NoDups, Len), % checking set with no duplicates is same length as list

countSmaller(Median, List, 0, Count),

countLarger(Median, List, 0, Count).

countSmaller(\_, [], A, A):- !.

countSmaller(Number, [H|T], Acc, Count) :-

H < Number,

NewAcc is Acc + 1,

countSmaller(Number, T, NewAcc, Count), !.

countSmaller(Number, [\_|T], Acc, Count) :-

countSmaller(Number, T, Acc, Count), !.

countLarger(\_, [], A, A):- !.

countLarger(Number, [H|T], Acc, Count) :-

H > Number,

NewAcc is Acc + 1,

countLarger(Number, T, NewAcc, Count), !.

countLarger(Number, [\_|T], Acc, Count) :-

countLarger(Number, T, Acc, Count), !.

E.

remove(X, List, Rest) :-

member(X, List),

removeAcc(X, List, Rest), !.

removeAcc(\_, [], []).

removeAcc(X, [H|T], Rest) :-

X == H,

removeAcc(X, T, Rest).

removeAcc(X, [H|T], [H|Rest]) :-

removeAcc(X, T, Rest).

F.

permute([], []).

permute([H|T], List2) :-

remove(H, List2, NewList2),

length(T, Len),

length(NewList2, Len),

permute(T, NewList2).