Q1.

```
7 ?- myappend([a,b], [c], X).
X = [a, b, c].

8 ?- myreverse([1,2,3], X).
X = [3, 2, 1].

9 ?- myflatten([[1],[2,3],[4]],X).
X = [1, 2, 3, 4] .

10 ?- mymember(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 .

11 ?- myremove(X,[2,3],3).
X = [2] .
```

```
myappend([], Y, Y).
myappend([H|X], Y, [H|Z]) :- myappend(X, Y, Z).

myreverse([], []).
myreverse([H|T], Z) :- myreverse(T, X), myappend(X, [H], Z).

myflatten([], []).
myflatten([H|T], Y) :- myflatten(H, HF), myflatten(T, TF), myappend(HF, TF, Y).
myflatten(X, [X]) :- not(is_list(X)).

is_list([]).
is_list([_|_]).

mymember(X, [X|_]).
mymember(X, [_|T]) :- mymember(X, T).

myremove(X, [X|T], T).
myremove([H|Z], [H|T], X) :- myremove(X, T, Z).
```

a
myappend(X, Y, Z), it takes three arguments and X is the first list and Y is the second list, Z is the appended list.

myappend([], Y, Y).
This is the base case, X is empty and Y list is same as Z list. This works since empty list appends Y will equal to Y.

myappend([H|X], Y, [H|Z]) :- myappend(X, Y ,Z).
We recursively get the append list, for X, H represents head element and X represent the rest of list. For Z, we also have head element H and rest of element Z.

([a, b], [c], Z)

We have [a, b] as the X and [c] as Y, which the head element H for X is a, b is X. We do the recursion with this value ([b], [c], Z1)
([b], [c], Z1) becomes ([], [c], Z2) which we did the recursion until base case is met.
Now Z2 has value c, we had [H|Z], for Z2, the H value will be b, and Z2 has value [c] so Z1 now has [b,c]. Now H has a and Z1 has [b,c] so it eventually becomes [a,b,c] for Z.

b
myreverse([], []). It takes two argument and has base case of empty list.
myreverse([H|T], Z) :- myreverse(T, X), myappend(X, [H], Z).

myreverse([1,2,3], X)
We will be doing recursion, it becomes ([2,3], X1) -> ([3], X2) -> ([], X3) -> []
And then we append the values, myappend(X3,[3],X2) -> myappend(X2,[2],X1) -> myappend(X1,[1],X).
X2 = [3], X1 = [3,2], X = [3,2,1].

c
myflatten([], []). This is the base case, and then we do recursion until we reach base case.
myflatten([H|T], Y) :- myflatten(H, HF), myflatten(T, TF), myappend(HF, TF, Y).
myflatten([[1],[2,3],[4]], X)
We have H = [1] and T = [[2,3],[4]]. [1] is in list so we do recursion so that H = 1 and T = [] where 1 is not list anymore so we append the 1 to HF. Next, myflatten([[2,3],[4]], TF), now H = [2,3] and T = [4] we do the same process as 1. It eventually becomes 2 and 3 in HF. Due to myflatten([4], TF). [4] will now be in TF, we append HF with TF so it becomes [2,3,4]. Append again with HF with TF so [1,2,3,4].

d.
mymember(X, [X|_]).
This is the base case. If X is equal to head of element, it's member.
mymember(X, [_|T]) :- mymember(X, T).
We do recursion until it matches, for example, ([2,[1,2,3]) since 2 isn't equal to head since it has head value 1. We then check ([2,[2,3]) which the head value equals to each other so it returns true.

e.
myremove(X, [X|T], T).
This is the base case. If the head of list is equal to X, then T becomes the result.
myremove([H|Z], [H|T], X) :- myremove(X, T, Z).
If head of list isn't equal to X then save the value in H and we do the recursion.
myremove(X, [2,3], 3), since head of list isn't equal to 3 so we save the value 2|Z in X. Now do recursion (X, [3], 3) so head of list equal to 3. Which tail of the list becomes the result that is 2.

Q2.

```
mymember2(X, Y) :-
    mymember(X, Y),
    myremove(X, Y, Z),
    mymember(X, Z),
    myremove(X, Z, XS),
    not(mymember(X, XS)).
```

```
2 ?- mymember2(1,[1,1,2]).
true .

3 ?- mymember2(2,[1,1,2]).
false.

4 ?- mymember2(1,[1,1,1]).
false.
```

mymember2(1, [1,1,2]). We first check if there is 1 in the list with mymember, then we remove the 1 from the list so list becomes [1,2] then now checks whether it still has 1 in the list. Again remove the 1 from the list and list now has [2], then we check whether 1 is in the list and since we don't have it we should return false but by using not(mymember) we return true. This only returns true when there is 2 occurrences of X in the list Y.

Q3

```
8 ?- mysubstring([3,4,5],[1,2,3,4,5,6]).
true

9 ?- mysubstring([1,3,5],[1,2,3,4,5,6]).
false.
```

```
mysubstring(X,Y) :- myappend(_, T, Y), myappend(X, _, T).
```

mysubstring(X,Y) :- myappend(_, T, Y), myappend(X, _, T).

Unknown list appending with T equals to Y which means that after removing the unknown list, T equal to the tails of the list Y. X appending with unknown list equals to T, which means that list X is equal to prefix of list T. If the list X is prefix of T and T is suffix of Y then then X is the substring of T.

Q4

```
5 ?- findall(L, mysublists([a, b, c], L), X).
X = [[a, b, c], [a, b], [a, c], [a], [b, c], [b], [c], []].

6 ?- findall(L, mysublistsR([a, b, c], L), X).
X = [[], [c], [b], [b, c], [a], [a, c], [a, b], [a|...]].
```

```
mysublists([], []).
mysublists([X|T], [X|TS]) :- mysublists(T, TS).
mysublists([_|T], TS) :- mysublists(T, TS).

mysublistsR([_|T], TS) :- mysublistsR(T, TS).
mysublistsR([X|T], [X|TS]) :- mysublistsR(T, TS).
mysublistsR([], []).
```

mysublist([a,b,c], X)
We have list [a,b,c]. mysublists([X|T], [X|TS]) :- mysublists(T, TS). which X = [a] and T = [b,c] so store in X = [a|TS] and we then recursively do mysublist([b,c],TS). X = b|TS -> X = c|TS, Due to mysublist([b,c],TS) -> mysublist([c],TS) -> mysublist([],TS)
We now reached base case where it's empty list.
For mysublists([_|T], TS) :- mysublists(T, TS). we skip the first element in the list. It creates mysublists([b,c],TS) -> mysublist([c],TS). Where TS = [c]. Then we get the TS = [a|TS] and TS= [c] which it becomes X = [a,c].

Q5.

```
7 ?- mypermutation([1,2,3], X).
X = [1, 2, 3] ;
X = [2, 1, 3] ;
X = [2, 3, 1] ;
X = [1, 3, 2] ;
X = [3, 1, 2] ;
X = [3, 2, 1] ;
false.
```

```
mypermutation([], []).
mypermutation([H|T], Y) :-
  mypermutation(T, Z),
  myappend(X, W, Z),
  myappend(X, [H|W], Y).
```

mypermutation([1,2,3], X).
We have list [1,2,3], then we do the recursion with tails, which it would be [2,3]. mypermutation([1,2,3], X) -> mypermutation([2,3], X). -> mypermutation([3], X) - > mypermutation([], X). We reached empty list so now we append 3 to the Z. We then also append 2 to before 3 and after 3 so it would be [2,3] and [3,2]. We also append 1 before and after to those two list. Then we have all possible list now.