Lists in Prolog (1) - P1

> 1A - Write a predicate to determine the lowest common multiple of a list formed from integer numbers.

Matematical Model **`gratest_common_divizor(N1, N2)`** =

> {N1, if N2 = 0

> { `gratest_common_divizor(N2, N1 mod N2)`

Matematical Model **`lowest_common_multiple(N1, N2)`** = N1 * N2 / `gratest_common_divizor(N1, N2)`

```
% gratest_common_divizor(N1 - first number, N2 - second number, R - result)
% flow model (i, i, o), (i, i, i)
gratest_common_divizor(N1, 0, N1). % the case when N2 is 1
gratest_common_divizor(N1, N2, R):-
    Reminder is N1 mod N2,
    gratest_common_divizor(N2, Reminder, R), !.

% lowest_common_multiple(N1 - first number, N2 - second number, R - result)
% flow model (i, i, o), (i, i, i)
lowest_common_multiple(_, 0, 0).
lowest_common_multiple(0, _, 0).
lowest_common_multiple(N1, N2, R):-
    gratest_common_divizor(N1, N2, GCD),
    R is N1 * N2 / GCD. % "is" should only be used when evaluating arithmetic operations on the
```

```
≡ ?- lowest_common_multiple(9, 7, R).
```

Matematical Model **lowest_common_multiple_all(l1..ln, R)** =

> { `lowest_common_multiple(l1, R)` , n = 1

> {lowest_common_multiple_all(l2..ln, `lowest_common_multiple(l1, R)`), otherwise

```
lowest_common_multiple_all([SingleNum], SingleNum).
lowest_common_multiple_all([FirstNum, SecondNum|Tail], LCM):-
    lowest_common_multiple(FirstNum, SecondNum, TempLCM),
    lowest_common_multiple_all([TempLCM|Tail], LCM), !.
```

```
≡ ?- lowest_common_multiple_all([2, 4, 6, 8], R).
```

7B => Write a predicate to create a list (m, ..., n) of all integer numbers from the interval [m, n].

```
% sublist(L - list, I - current index, S - start of sublist, E - end of sublist, R - resulted l
% flow model (i, i, i, i, o) (i, i, i, i, i)
sublist([], _, _, _, []).
sublist([H|T], I, S, E, R):-
    I >= S,
    I =< E,
    NewI is I + 1,
    sublist(T, NewI, S, E, NewR),
    R = [H|NewR], !.
sublist([_|T], I, S, E, R):-
    NewI is I + 1,
    sublist(T, NewI, S, E, R).
sublis(L, S, E, R):-
    sublist(L, 1, S, E, R).
```

```
?- sublist([1, 2, 3, 4, 5, 6, 7], 3, 6, R).
```

R = [3, 4, 5, 6]

5B. Write a predicate to determine the set of all the pairs of elements in a list. Eg.: L = [a b c d] => [[a b] [a c] [a d] [b c] [b d] [c d]].

```
% myMember(E - elemnt, L - list)
% flow model (i, i)
myMember(E, [E|_]).
myMember(E, [_|T]):- % if they are not equal keep searching
    myMember(E, T).
```

```
?- myMember(2, [2, 3, 4, 1]).
```

true 1

```
% concatenate(L1 - list 1, L2 -  list 2, R - resulted list)
concatenate([], L2, L2).
concatenate([H|T], L2, R):-
    concatenate(T, L2, NewR),
    R = [H|NewR], !.
```

```
?- concatenate([1, 2, 3], [4, 5, 6], R).
```

R = [1, 2, 3, 4, 5, 6]

```
% Base case: An empty list has no pairs.
pairs([], []).

% Recursive case: For a list with head H and tail T, a pair is [H, X] for each element X in T,
% combined with all pairs in T.
pairs([H|T], Pairs) :-
    findall([H, X], myMember(X, T), CurrentPairs), % places in H_Pairs all pairs having as elem
    %write(H_Pairs), nl,
    pairs(T, NewPairs), % recursive call returning the pairs untill then
    concatenate(CurrentPairs, NewPairs, Pairs). %
```

```
?- pairs([1, 2, 3, 4, 5], R).
```

```
R = [[1, 2], [1, 3], [1, 4], [1, 5], [2, 3], [2, 4], [2, 5], [3, 4], [3, 5], [4, 5]]
```

6b. Write a predicate to remove the first three occurrences of an element in a list. If the element occurs less than three times, all occurrences will be removed.

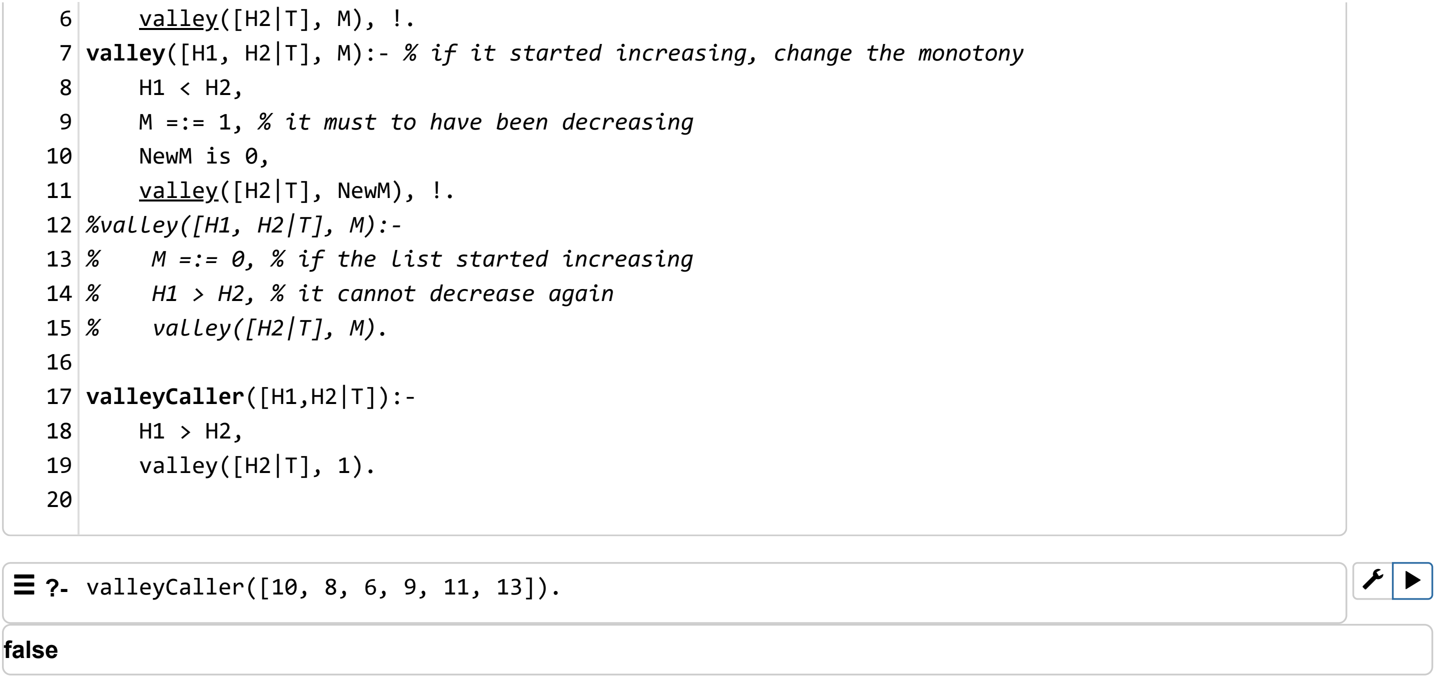
```
% removeMax3(L - list, E - element to be removed, C - how many removed, R - resulted list)
% flow model (i, i, i, o)
removeMax3([], _, _, []).
removeMax3([E|T], E, 2, T). % if it was already removed 2 times and the head = element, resulte
removeMax3([E|T], E, C, R):-
    NewC is C + 1,
    removeMax3(T, E, NewC, R), !.
removeMax3([H|T], E, C, [H|R]):- % if the head and the element are not equal add H to the resul
    removeMax3(T, E, C, R), !.
    %R = [H|NewR].
removeMax3Caller(L, E, R):-
    removeMax3(L, E, 0, R).
```

```
?- removeMax3Caller([1, 10, 2, 10, 3, 10, 4, 10, 5, 10], 10, R).
```

```
R = [1, 2, 3, 4, 10, 5, 10]
```

10.a. Define a predicate to test if a list of an integer elements has a "valley" aspect (a set has a "valley" aspect if elements decreases up to a certain point, and then increases. eg: 10 8 6 9 11 13 – has a “valley” aspect

```
% valley(L - list, M (0 - increasing, 1 - decreasing))
valley([], M):-
    M =:= 0.
valley([H1, H2|T], M):- % if it continues to decrese, keep going
    H1 > H2,
```



```
    valley([H2|T], M), !.
valley([H1, H2|T], M):- % if it started increasing, change the monotony
    H1 < H2,
    M =:= 1, % it must to have been decreasing
    NewM is 0,
    valley([H2|T], NewM), !.
%valley([H1, H2|T], M):-
%    M =:= 0, % if the list started increasing
%    H1 > H2, % it cannot decrease again
%    valley([H2|T], M).

valleyCaller([H1,H2|T]):-
    H1 > H2,
    valley([H2|T], 1).

```

```
?- valleyCaller([10, 8, 6, 9, 11, 13]).
```

```
false
```