1. Solve exercises 3.2.1 (i), (ii), (iii), and (iv).

i. A variant of Program 3.14 for sublist is defined by the following three rules:

```
subsequence([X|Xs],[X|Ys]) :- subsequence(Xs,Ys).
subsequence(Xs,[Y|Ys]) :- subsequence(Xs,Ys).
subsequence([],Ys).
```

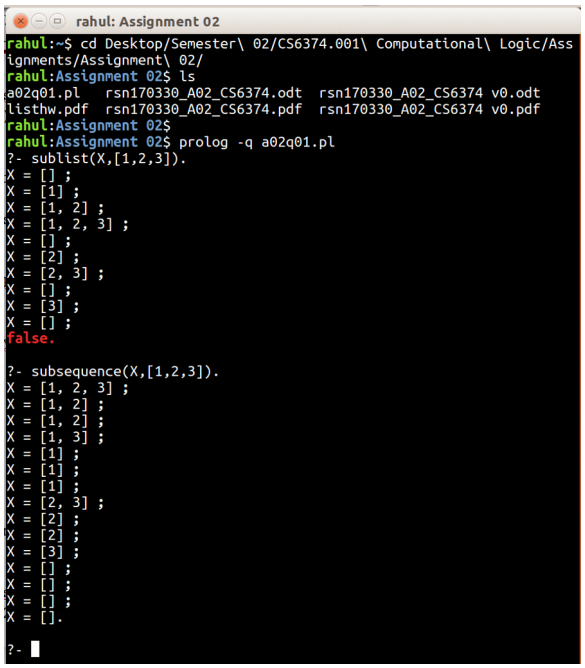
Explain why this program has different meaning from Program 3.14

```
% sublist(Sub,List): Sub is a sublist of List.

% a: suffix of a prefix
sublist(Xs,Ys) :- prefix(Ps,Ys), suffix(Xs,Ps).

% b: prefix of a suffix
sublist(Xs,Ys) :- prefix(Xs,Ss), suffix(Ss,Ys).

% c: recursive definition of a sublist
sublist(Xs,Ys) :- prefix(Xs,Ys).
sublist(Xs,[Y|Ys]) :- sublist(Xs,Ys).
```

Program 3.14 Determining sublist of lists

Ans.

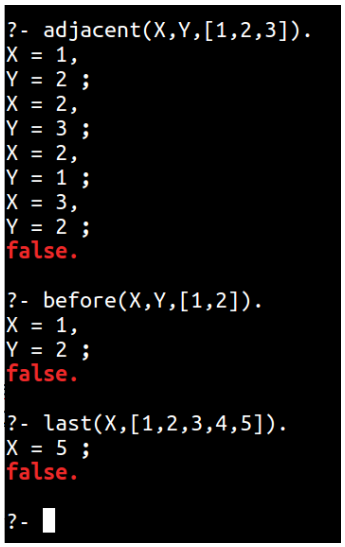
For sublist, the list is broken into all possible connected subsets.
Whereas for subsequence, it is broken into all possible disconnected/connected subsets.



ii. Write recursive programs for adjacent and last that have the same meaning as the predicates defined in the text in terms of append.

Ans.

```
% before(X,Y,List): X is before Y in List
before(X,Y,[X,Y|_]).
before(X,Y,[_|T]) :-
      before(X,Y,T).

% after(X,Y,List): X is after Y in List
after(X,Y,[Y,X|_]).
after(X,Y,[_|T]) :-
      after(X,Y,T).

% adjacent(X,Y,List): X is adjacent to Y in List
adjacent(X,Y,List) :-
      before(X,Y,List);
      after(X,Y,List).

% last(X,List): X is last element in List
last(X,[X]).
last(X,[_|T]) :-
      last(X,T).
```



iii. Write a program for double(List,ListList), where every element in the List appears twice in ListList, e.g., double([1,2,3],[1,1,2,2,3,3]) is true.

```
?- double([1,2,3],X).
X = [1, 1, 2, 2, 3, 3].

?- double(X,[1,1,23,23,45,45]).
X = [1, 23, 45].

?-
```

Ans.

```
% double(L,LL): every element of L occure twice in LL
double([],[]).
double([H|T1],[H,H|T2]) :-
      double(T1,T2).
```

iv. Compute the size of the proof tree as a function of the size of the input list for Programs 3.16a and 3.16b defining reverse.

```
% reverse(List,Tsil): Tsil is the result
of reversing the list List.

% a: Naive reverse

reverse([],[]).
reverse([X|Xs],Zs) :-
      reverse(Xs,Ys),
      append(Ys,[X],Zs).

% b: Reverse-accumulate
reverse(Xs,Ys) :-
      reverse(Xs,[],Ys).

reverse([X|Xs],Acc,Ys) :-
      reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

Program 3.16 Reversing a list

Ans.

Lets check with the trace command how many times we need to call (calls in the proof tree) for different values of list size n.

For naive reverse:

n	1	2	3	4	5
calls	6	12	20	30	42

Hence, calls(n) = $\mathbf{n^2 + 3n + 2}$

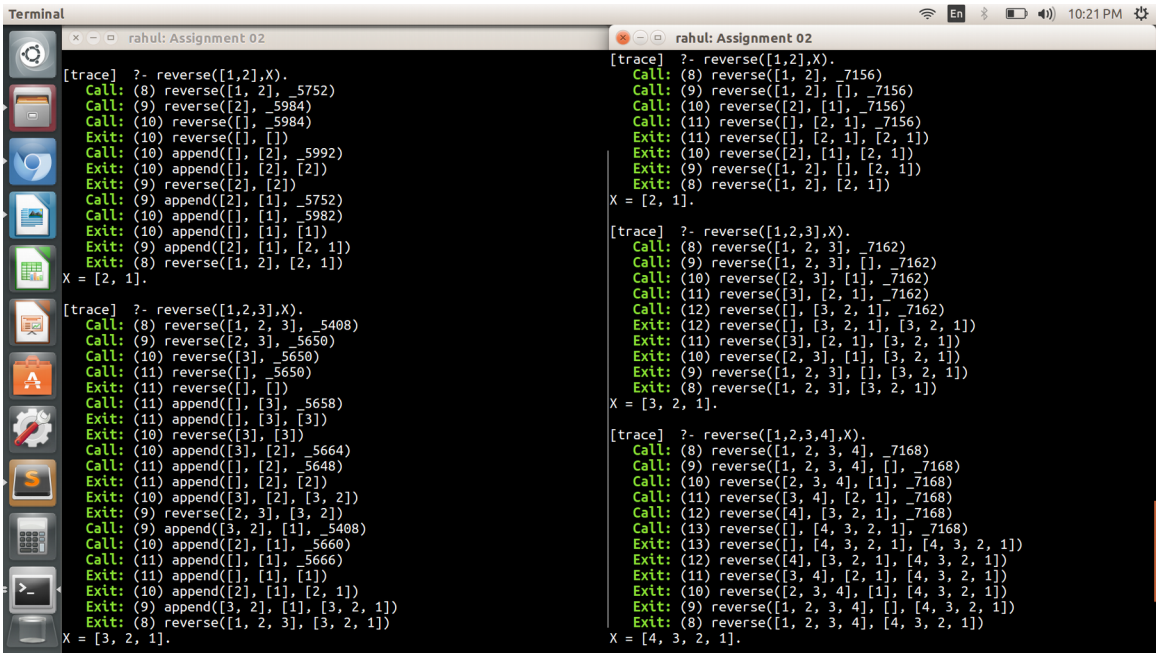
For reverse-accumulate

n	1	2	3	4	5
calls	6	8	10	12	14

Hence, calls(n) = **2n + 4**

So, naive reverse/3.16a is $O(n^2)$ whereas reverse-accumulate/3.16b is O(n), which is far better than $O(n^2)$.

*You can refer from this image, where left is for naive and right for rev-acc.



2. Solve exercises 3.3.1 (i), (ii), (iii), (v), (vi), (vii).

i. Write a program for substitute(X,Y,L1,L2), where L2 is the result of substituting Y for all occurrences of X in L1, e.g.,
substitute(a,x,[a,b,a,c],[x,b,x,c]) is true, whereas
substitute(a,x,[a,b,a,c],[a,b,x,c]) is false.

Ans.

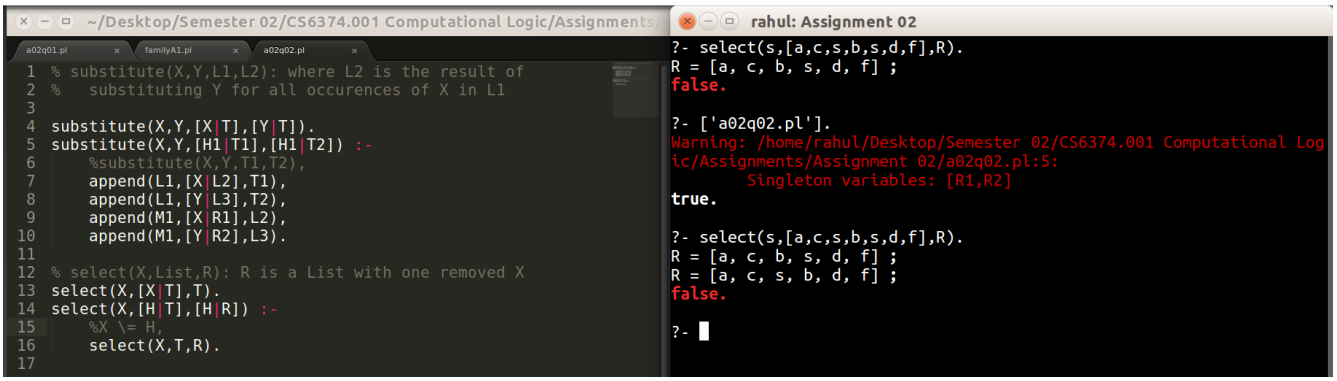
```
% substitute(X,Y,L1,L2): where L2 is the result of
%     substituting Y for all occurrences of X in L1
substitute(X,Y,[],[]).

substitute(X,Y,[X|T1],[Y|T2]) :-
      substitute(X,Y,T1,T2).

substitute(X,Y,[Z|T1],[Z|T2]) :-
      Z \= X,
      substitute(X,Y,T1,T2).
```

```
?- substitute(a,x,[a,b,a,c],X).
X = [x, b, x, c] .

?- substitute(a,x,[a,b,a,c],[x,b,x,c]).
true .

?- substitute(a,x,[a,b,a,c],[a,b,x,c]).
false.

?-
```

ii. What is the meaning of the variant of the select:
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :- X != Y, select(X,Ys,Zs).

Ans.

From fig below, first output of select() corresponds to above variant with X!=Y, whereas second select() output is without it (commented one).
Hence, above variant only outputs selecting first instance[**Deterministic**] whereas, original variant of select outputs all possible solutions, selecting every instances at a time[**Non-Deterministic**].



iii. Write a program for no_doubles(L1,L2), where L2 is the result of removing all duplicate elements from L1, e.g.,
no_doubles([a,b,c,b],[a,c,b]) is true. (Hint: use member.)

Ans.

```
% no_doubles(L1,L2): L2 is list of all distinct
%     values in L1, keeping last instance.
no_doubles([],[]).

no_doubles([H|T],R) :-
      member(H,T),
      no_doubles(T,R).

no_doubles([H|T],[H|R]) :-
      \+ member(H,T),
      no_doubles(T,R).
```

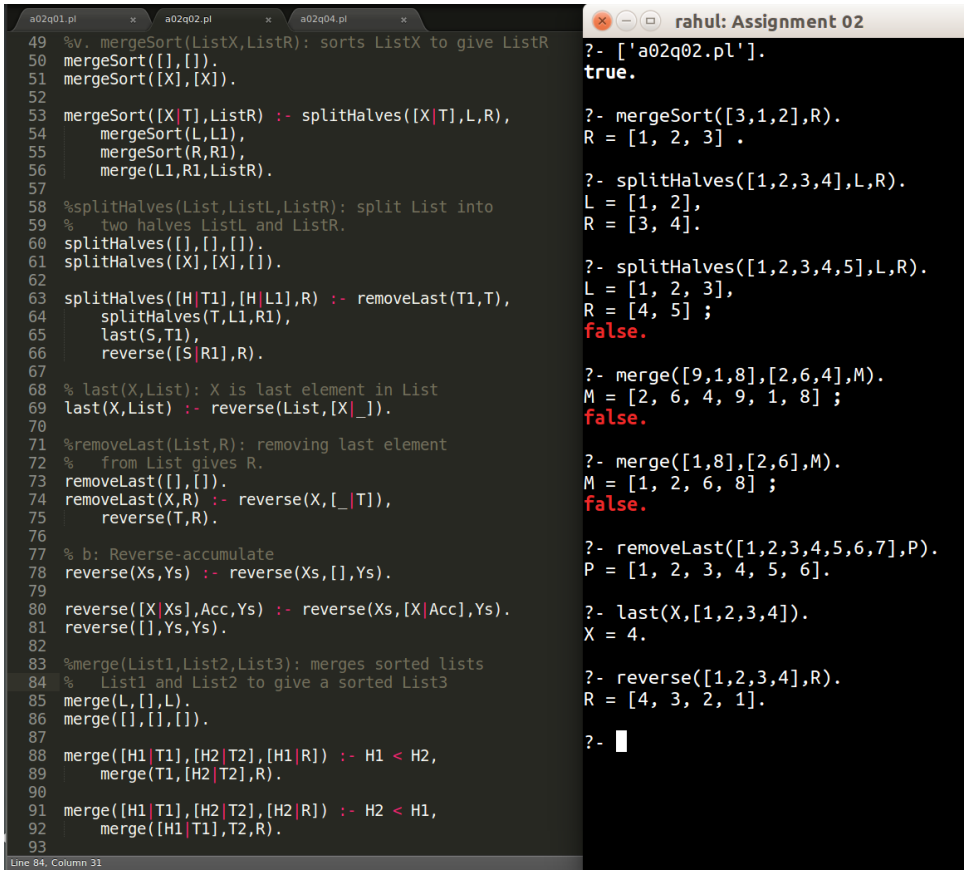
```
?- no_doubles([1,2,3,3,2,4,1],X).
X = [3, 2, 4, 1] ;
false.

?- no_doubles([a,b,c,b],[a,c,b]).
true .

?-
```

v. Write a program for merge sort.

Ans.

```
% mergeSort(ListX,ListR): sorts ListX to give ListR
mergeSort([],[]).
mergeSort([X],[X]).

mergeSort([X|T],ListR) :- splitHalves([X|T],L,R),
      mergeSort(L,L1),
      mergeSort(R,R1),
      merge(L1,R1,ListR).

%splitHalves(List,ListL,ListR): split List into
%     two halves ListL and ListR.
splitHalves([],[],[]).
splitHalves([X],[X],[]).

splitHalves([H|T1],[H|L1],R) :- removeLast(T1,T),
      splitHalves(T,L1,R1),
      last(S,T1),
      reverse([S|R1],R).

% last(X,List): X is last element in List
last(X,List) :- reverse(List,[X|_]).

%removeLast(List,R): removing last element
%     from List gives R.
removeLast([],[]).
removeLast(X,R) :- reverse(X,[_|T]),
      reverse(T,R).

% b: Reverse-accumulate
reverse(Xs,Ys) :- reverse(Xs,[],Ys).

reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).

%merge(List1,List2,List3): merges sorted lists
%     List1 and List2 to give a sorted List3
merge(L,[],L).
merge([],[],[]).

merge([H1|T1],[H2|T2],[H1|R]) :- H1 < H2,
      merge(T1,[H2|T2],R).

merge([H1|T1],[H2|T2],[H2|R]) :- H2 < H1,
      merge([H1|T1],T2,R).
```

Following is the screen-shot of the run.



```
a02q01.pl  a02q02.pl  a02q04.pl

%v. mergeSort(ListX,ListR): sorts ListX to give ListR
mergeSort([],[]).
mergeSort([X],[X]).

mergeSort([X|T],ListR) :- splitHalves([X|T],L,R),
    mergeSort(L,L1),
    mergeSort(R,R1),
    merge(L1,R1,ListR).

%splitHalves(List,ListL,ListR): split List into
%   two halves ListL and ListR.
splitHalves([],[],[]).
splitHalves([X],[X],[]).

splitHalves([H|T1],[H|L1],R) :- removeLast(T1,T),
    splitHalves(T,L1,R1),
    last(S,T1),
    reverse([S|R1],R).

% last(X,List): X is last element in List
last(X,List) :- reverse(List,[X|_]).

%removeLast(List,R): removing last element
%   from List gives R.
removeLast([],[]).
removeLast(X,R) :- reverse(X,[_|T]),
    reverse(T,R).

% b: Reverse-accumulate
reverse(Xs,Ys) :- reverse(Xs,[],Ys).

reverse([X|Xs],Acc,Ys) :- reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).

%merge(List1,List2,List3): merges sorted lists
%   List1 and List2 to give a sorted List3
merge(L,[],L).
merge([],[],[]).

merge([H1|T1],[H2|T2],[H1|R]) :- H1 < H2,
    merge(T1,[H2|T2],R).

merge([H1|T1],[H2|T2],[H2|R]) :- H2 < H1,
    merge([H1|T1],T2,R).

Line 84, Column 31
```

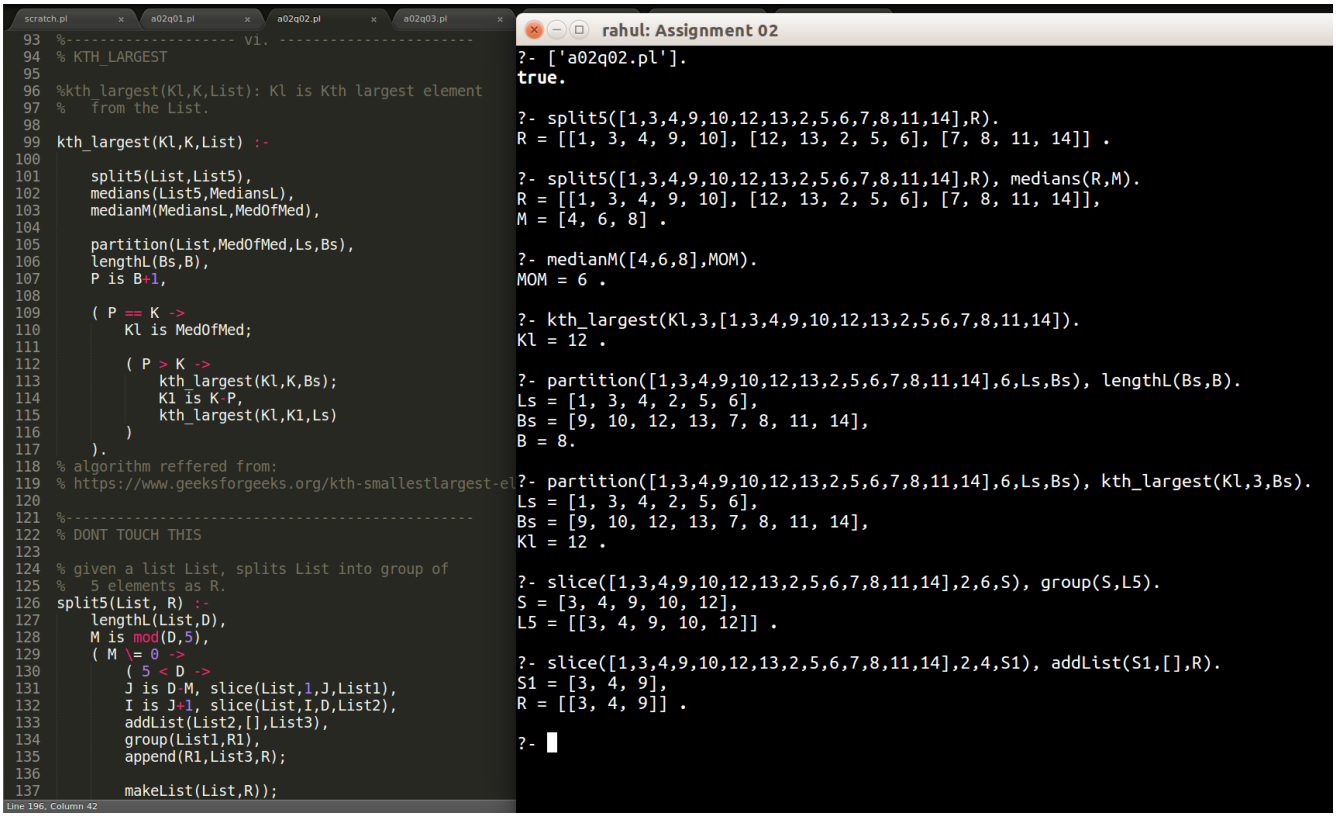
```
rahul: Assignment 02

?- ['a02q02.pl'].
true.

?- mergeSort([3,1,2],R).
R = [1, 2, 3] .

?- splitHalves([1,2,3,4],L,R).
L = [1, 2],
R = [3, 4].

?- splitHalves([1,2,3,4,5],L,R).
L = [1, 2, 3],
R = [4, 5] ;
false.

?- merge([9,1,8],[2,6,4],M).
M = [2, 6, 4, 9, 1, 8] ;
false.

?- merge([1,8],[2,6],M).
M = [1, 2, 6, 8] ;
false.

?- removeLast([1,2,3,4,5,6,7],P).
P = [1, 2, 3, 4, 5, 6].

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

?- reverse([1,2,3,4],R).
R = [4, 3, 2, 1].

?-
```

vi. Write a program for kth_largest(Xs,K) that implements the linear algorithm for finding the kth largest element K of a list Xs. The algorithm has following steps:

1. Break the list into the groups of five elements.
2. Efficiently find the median of each of the groups, which can be done with a fixed no of comparisons.
3. Recursively find the median of the medians.
4. Partition the original list with respect to the median of medians.
5. Recursively find the kth largest element in the appropriate smaller list.

Ans.

Please, refer file **a02q02.pl**. Here is the output for kth_largest.



vii. Write a program for the relation better_poker_hand(Hand1,Hand2,Hand) that succeeds when Hand is better poker hand between Hand1 and Hand2. For those unfamiliar with this card game, here are some rules of poker necessary for answering this exercise:

(a) The order of cards is 2,3,4,5,6,7,8,9,10,jack,queen,king,ace.

(b) Each hand consists of five cards.

(c) The rank of hands in ascending order is no pairs < one pair < two pairs < three of a kind < flush < straight < full house < four of a kind < straight flush.

(d) Where two cards have the same rank, the higher denomination wins, for example, a pair of kings beats a pair of 7's.

Hints:

1. Represent a poker hand by a list of terms of the form card(Suit,Value). For example a hand consisting of the 2 of clubs, the 5 of spades, the queen of hearts, the queen of diamonds, and the 7 of spades would be represented by the list [card(clubs,2), card(spades,5), card(hearts,queen), card(diamonds,queen), card(spades,7)].

2. It may be helpful to define relations such as
   has-flush(Hand),which is true if all the cards in Hand are of the same suit;
   has-full-house(Hand),which is true if Hand has three cards with the same value but in different suits, and the other two cards have the same different value; and
   has_straight(Hand),which is true if Hand has cards with consecutive values.

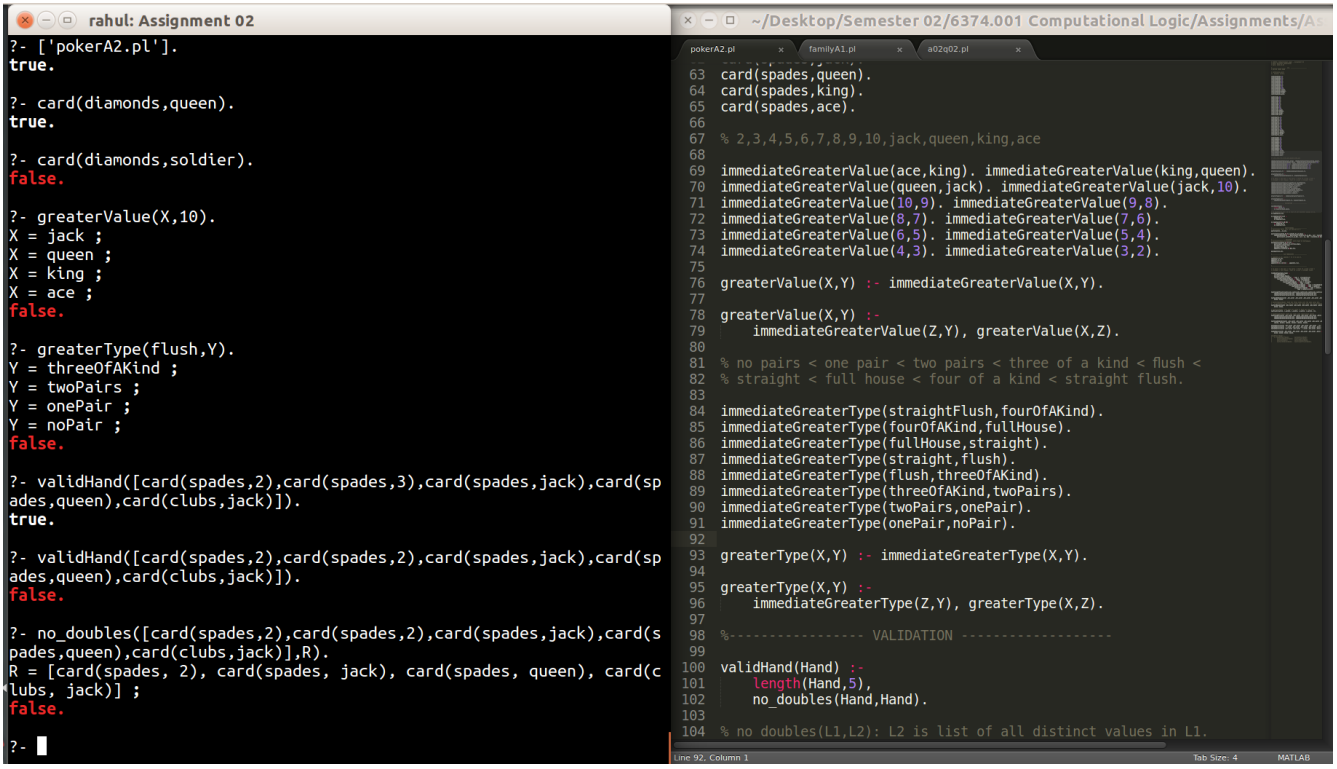
3. The number of cases to consider is reduced if the hand is first sorted.

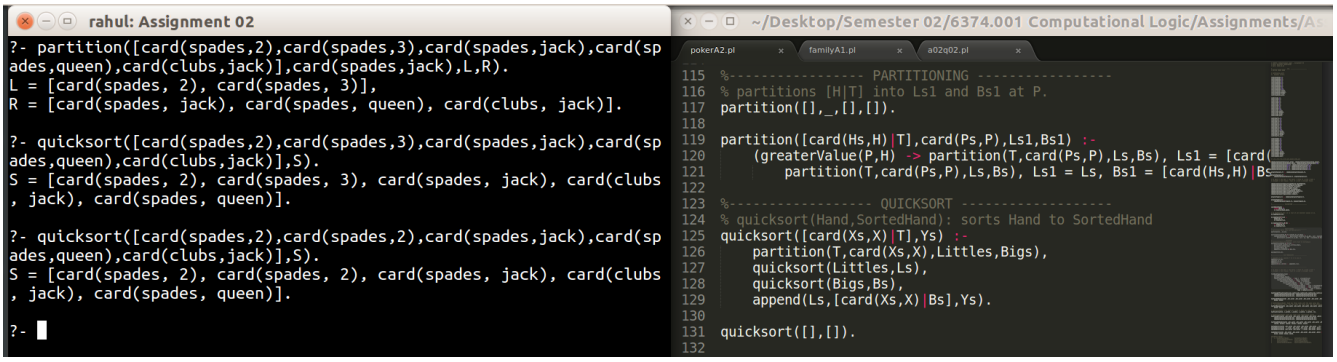
Ans.

?-['pokerA2.pl'].

Please, refer the file 'pokerA2.pl' attached herewith.

Following are the screen shots of the runs.





rahul: Assignment 02

```
?- findHandType([card(spades,7),card(spades,9),card(diamonds,10),card
(spades,2),card(spades,4)],Type).
Type = noPair.

?- findHandType([card(spades,7),card(spades,9),card(diamonds,10),card
(spades,10),card(spades,4)],Type).
Type = onePair.

?- findHandType([card(spades,7),card(clubes,10),card(diamonds,10),car
d(spades,10),card(spades,4)],Type).
Type = threeOfAKind.

?- findHandType([card(spades,7),card(clubs,7),card(diamonds,10),card(
spades,10),card(spades,4)],Type).
Type = twoPairs.

?- findHandType([card(spades,7),card(clubes,8),card(diamonds,9),card(
spades,10),card(spades,jack)],Type).
Type = straight.

?- findHandType([card(spades,7),card(spades,2),card(spades,9),card(sp
ades,10),card(spades,jack)],Type).
Type = flush.

?- findHandType([card(spades,2),card(clubs,2),card(hearts,jack),card(
spades,jack),card(clubs,jack)],Type).
Type = fullHouse.

?- findHandType([card(spades,2),card(diamonds,jack),card(spades,jack)
,card(hearts,jack),card(clubs,jack)],Type).
Type = fourOfAKind.

?- findHandType([card(spades,7),card(spades,8),card(spades,9),card(sp
ades,10),card(spades,jack)],Type).
Type = straightFlush.

?-
```

~/Desktop/Semester 02/6374.001 Computational Logic/Assignments/A

pokerA2.pl | familyA1.pl | a02q02.pl

```
findHandType(Hand,Type) :-
    validHand(Hand),
    quicksort(Hand,SHand),
    (hasStraightFlush(SHand) -> Type =  straightFlush;
        (hasFourOfAKind(SHand) -> Type = fourOfAKind;
            (hasFullHouse(SHand) -> Type = fullHouse;
                (hasStraight(SHand) -> Type = straight;
                    (hasFlush(SHand) -> Type = flush;
                        (hasThreeOfAKind(SHand) -> Type = threeOfAKin
                            (hasTwoPairs(SHand) -> Type = twoPairs;
                                (hasOnePair(SHand) -> Type = onePair;
                                    Type = noPair)))))))).

hasStraightFlush([card(S,X1),card(S,X2),card(S,X3),card(S,X4),card(S,
    immediateGreater(X5,X4), immediateGreater(X4,X3),
    immediateGreater(X3,X2), immediateGreater(X2,X1).

hasFourOfAKind([card(_,X1),card(_,X2),card(_,X2),card(_,X2),card(_,X3
    X2=X1; X2=X3.

% hasFullHouse(Hand): Hand has three cards with the same value but,
%   in different suits, and the other two cards have the same differe
hasFullHouse([card(_,X2),card(_,X2),card(_,X3),card(_,X4),card(_,X4)]
    X3=X2; X3=X4.

% hasFlush(Hand): true when all cards in Hand has same Suit.
hasFlush([card(S,_),card(S,_),card(S,_),card(S,_),card(S,_)]).

% hasStraight(Hand): Hand has cards with consecutive values
hasStraight([card(_,X1),card(_,X2),card(_,X3),card(_,X4),card(_,X5)])
    immediateGreater(X5,X4), immediateGreater(X4,X3),
    immediateGreater(X3,X2), immediateGreater(X2,X1).

hasThreeOfAKind([card(_,X1),card(_,X2),card(_,X3),card(_,X4),card(_,X
    (X1=X2, X2=X3); (X2=X3, X3=X4); (X3=X4, X4=X5).

hasTwoPairs([card(_,X1),card(_,X1),card(_,X3),card(_,X3),card(_,_)]).
hasTwoPairs([card(_,_),card(_,X1),card(_,X1),card(_,X3),card(_,X3)]).
hasTwoPairs([card(_,X1),card(_,X1),card(_,_),card(_,X3),card(_,X3)]).

hasOnePair([card(_,X1),card(_,X2),card(_,X3),card(_,X4),card(_,X5)])
    X1=X2; X2=X3; X3=X4; X4=X5.
```

Line 142, Column 1 | Tab Size: 4 | MATLAB

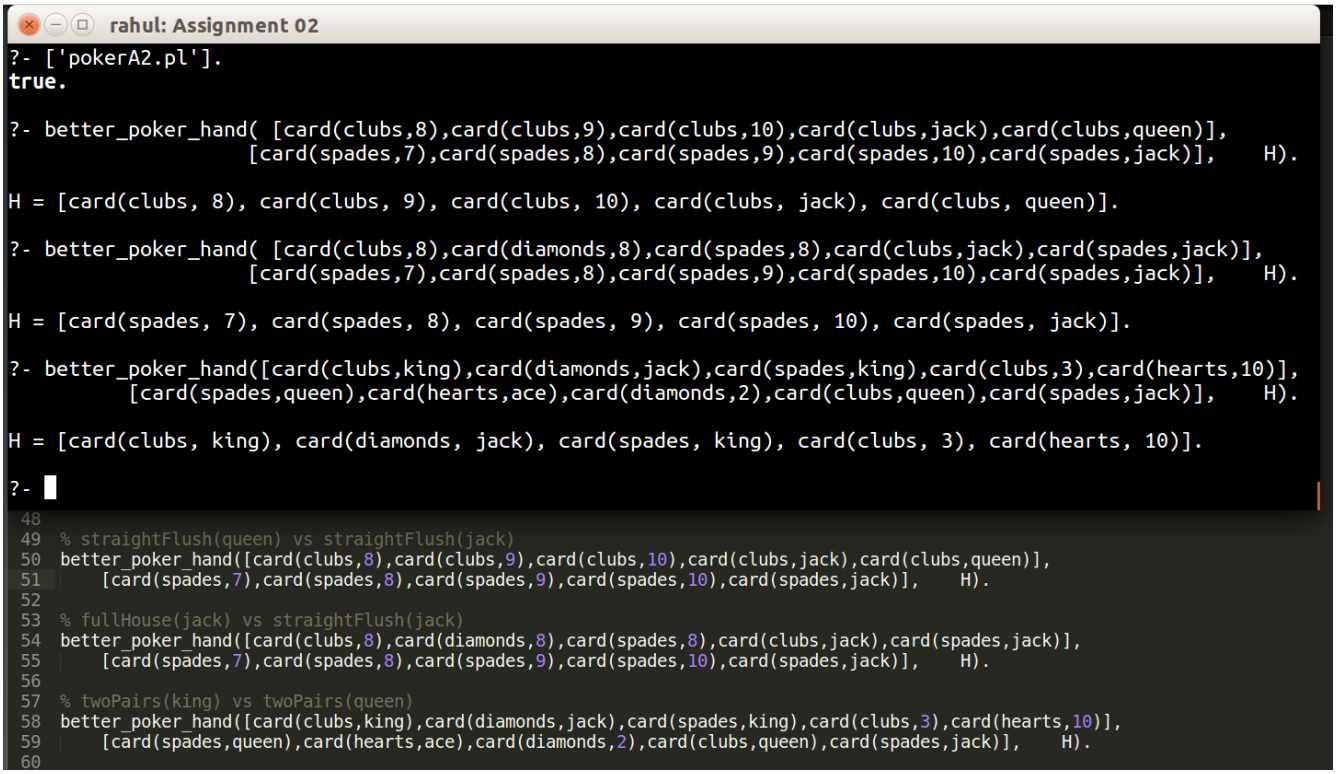
pokerA2.pl | scratch.pl | familyA1.pl | a02q02.pl

```
%---------------- TIE-BREAKS ------------------
betterDenomination(Hand1,Hand2,Type,R) :-
    ((Type == fullHouse; Type == fourOfAKind; Type == twoPairs;
    Type == onePair; Type == noPair) ->
        getDenomination5(Hand1,D1), getDenomination5(Hand2,D2),
        (winner(D1,D2,D1) -> R = Hand1;
            R = Hand2);
        getDenomination3(Hand1,E1), getDenomination3(Hand2,E2),
        (winner(E1,E2,E1) -> R = Hand1;
            R = Hand2)).

%-----------------------------------------------
% works for noPair, onePair, twoPairs, fourOfAKind, fullHouse
getDenomination5(Hand,D) :-
    quicksort(Hand,S),
    hasPair(S,D).

hasPair([card(_,X1),card(_,X2),card(_,X3),card(_,X4),card(_,X5)],Pair) :-
    ( X4=X5 -> Pair = X4;
        ( X4=X3 -> Pair = X3;
            ( X3=X2 -> Pair = X2;
                ( X2=X1 -> Pair = X1;
                    Pair = X5)))).
winner(D1,D2,D) :-
    (greaterValue(D1,D2) -> D = D1;
        D = D2).

%-----------------------------------------------
% works for straight, flush, straightFlush
getDenomination3(Hand,D) :-
    quicksort(Hand,S),
    last(card(_,D),S).

%------------------- GAME ----------------------

better_poker_hand(Hand1,Hand2,R) :-
    findHandType(Hand1,Type1),
    findHandType(Hand2,Type2),
    (Type1 == Type2 ->
        betterDenomination(Hand1,Hand2,Type1,R);
        (greaterType(Type1,Type2) -> R = Hand1;
            R = Hand2)).

%-----------------------------------------------
```

Line 226, Column 35

rahul: Assignment 02

```
?- better_poker_hand([card(clubs,king),card(diamonds,jack),card(spade
s,king),card(clubs,3),card(hearts,10)],
[card(spades,queen),card(hearts,ace),card(diamonds,2),card(clubs,quee
n),card(spades,jack)],    H).
H = [card(clubs, king), card(diamonds, jack), card(spades, king), car
d(clubs, 3), card(hearts, 10)].

?- findHandType([card(clubs,king),card(diamonds,jack),card(spades,kin
g),card(clubs,3),card(hearts,10)], Type1).
Type1 = onePair.

?- findHandType([card(spades,queen),card(hearts,ace),card(diamonds,2)
,card(clubs,queen),card(spades,jack)], Type2).
Type2 = onePair.

?- getDenomination5([card(clubs,king),card(diamonds,jack),card(spades
,king),card(clubs,3),card(hearts,10)], D1).
D1 = king.

?- getDenomination3([card(clubs,king),card(diamonds,jack),card(spades
,king),card(clubs,3),card(hearts,10)], D1).
D1 = king.

?- getDenomination5([card(spades,queen),card(hearts,ace),card(diamond
s,2),card(clubs,queen),card(spades,jack)], D2).
D2 = queen.

?- getDenomination3([card(spades,queen),card(hearts,ace),card(diamond
s,2),card(clubs,queen),card(spades,jack)], D2).
D2 = ace.

?- quicksort([card(spades,queen),card(hearts,ace),card(diamonds,2),ca
rd(clubs,queen),card(spades,jack)], S), hasPair(S,D).
S = [card(diamonds, 2), card(spades, jack), card(spades, queen), card
(clubs, queen), card(hearts, ace)],
D = queen.

?-
```

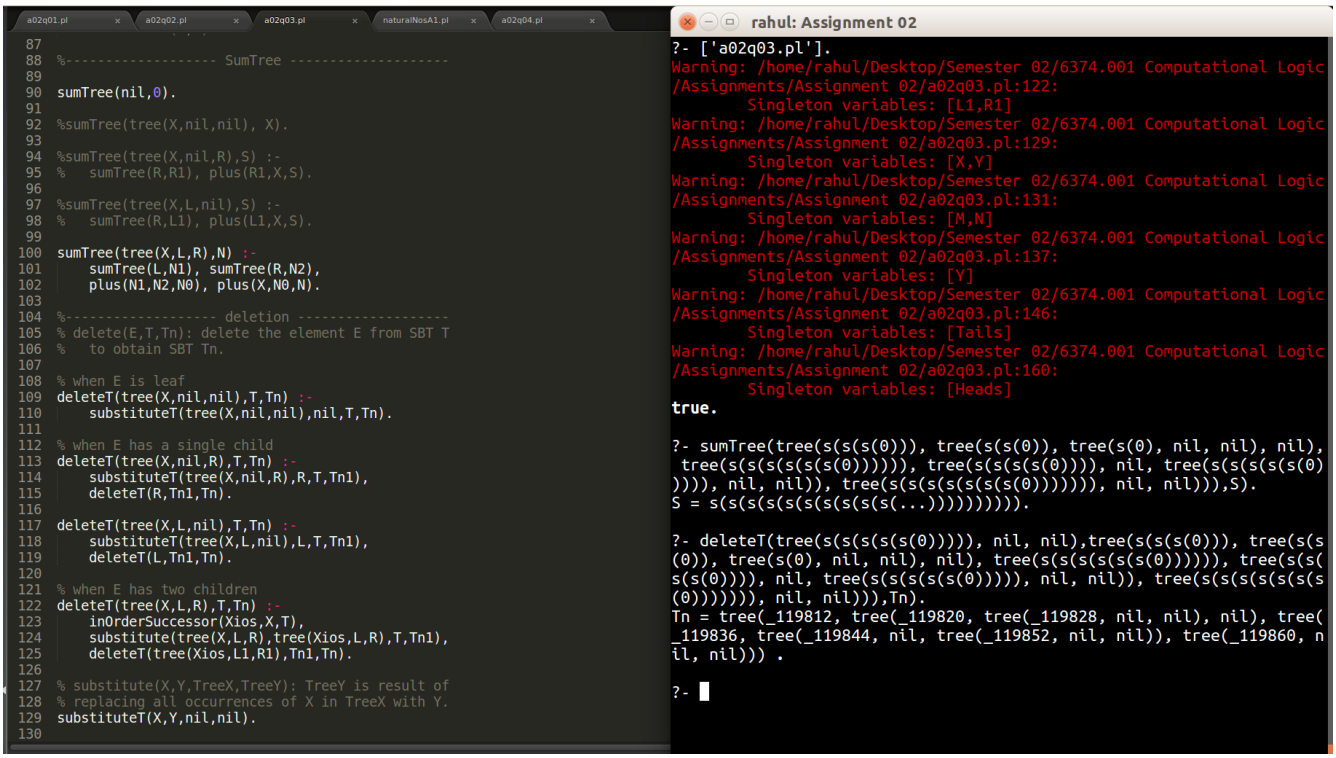


*typo - last run of better_poker_hand is of type **onePair(King) vs onePair(queen).**

3. Given the sorted binary tree (SBT) representation discussed in class, define the following functions
sumtree(T,N): N is the sum of elements in SBT T (use succ arithmetic).
delete(E,T,Tn): delete the element E from SBT T to obtain SBT Tn.
Ans.

```
%------------------- SumTree --------------------

sumTree(nil,0).

%sumTree(tree(X,nil,nil), X).

%sumTree(tree(X,nil,R),S) :-
%     sumTree(R,R1), plus(R1,X,S).

%sumTree(tree(X,L,nil),S) :-
%     sumTree(R,L1), plus(L1,X,S).

sumTree(tree(X,L,R),N) :-
      sumTree(L,N1), sumTree(R,N2),
      plus(N1,N2,N0), plus(X,N0,N).

%------------------- deletion -------------------
% delete(E,T,Tn): delete the element E from SBT T
%     to obtain SBT Tn.

% when E is leaf
deleteT(tree(X,nil,nil),T,Tn) :-
      substituteT(tree(X,nil,nil),nil,T,Tn).

% when E has a single child
deleteT(tree(X,nil,R),T,Tn) :-
      substituteT(tree(X,nil,R),R,T,Tn1),
      deleteT(R,Tn1,Tn).

deleteT(tree(X,L,nil),T,Tn) :-
      substituteT(tree(X,L,nil),L,T,Tn1),
      deleteT(L,Tn1,Tn).

% when E has two children
deleteT(tree(X,L,R),T,Tn) :-
      inOrderSuccessor(Xios,X,T),
      substitute(tree(X,L,R),tree(Xios,L,R),T,Tn1),
      deleteT(tree(Xios,L1,R1),Tn1,Tn).

% substitute(X,Y,TreeX,TreeY): TreeY is result of
% replacing all occurrences of X in TreeX with Y.
substituteT(X,Y,nil,nil).

substituteT(X,Y,tree(M,ML,MR),tree(N,NL,NR))    :-
      replace(X,Y,X,Y),
      substituteT(X,Y,ML,NL),
      substituteT(X,Y,MR,NR).

replace(X,Y,X,Y).
replace(X,Y,Z,Z) :- X \= Z.
```

```
% inorder(Tree,Iot): Iot is inorder traversal of Tree
inorder(tree(X,L,R), Xs) :-
      inorder(L,Ls), inorder(R,Rs), append(Ls,[X|Rs],Xs).
inorder(nil,[]).

% inOrderSuccessor(X,Y,T): X is inorder successor of Y
%     in tree T.
inOrderSuccessor(X,Y,Tree) :-
      inorder(Tree,T),
      sublist([Y, X|Tails],T),
      nat(X), nat(Y).
      %succ(X,Y,T).

% succ(X,Y,List): X is after Y in List
succ(X,Y,[Y,X|_]) :- nat(X).
succ(X,Y,[_|T]) :-
      succ(X,Y,T).

% sublist(Sub,List): Sub is a sublist of List.
% c: recursive definition of a sublist
sublist(Xs,Ys) :- prefix(Xs,Ys).
sublist(Xs,[Heads|Ys]) :- sublist(Xs,Ys).
```



4. Exercises 8.3.1 (i), (iii), (vi), (vii)

i. Write an iterative version for triangle(N,T), posed as Exercise 8.2(i).

Exercise 8.2
(i) The Nth triangular number is the sum of the numbers up to and including N. Write a program for the relation triangle (N ,T), where '1' is the Nth triangular number. (Hint: Adapt Program 8.2.)

```
%factorial(N,F): F is the integer N factorial.

factorial1(N,F) :-
      N > 0, N1 is N-1, factorial(N1,F1), F is N*F1.
factorial(0,1).
```

Program 8.2 Computing the factorial of a number

Ans.

```
%i. Nth_triangular(N,T): T = N*(N-1)/2

% triangle(N,T): T is sum of first N natural numbers.
triangle(N,T) :- triangle(0,N,0,T).

triangle(I,N,A,T) :-
      I < N, I1 is I+1,
      A1 is A+I1, triangle(I1,N,A1,T).

triangle(N,N,T,T).
```

```
?- ['a02q04.pl'].
true.

?- triangle(3,T).
T = 6 ;
false.

?- triangle(5,T).
T = 15 ;
false.

?-
```

iii. Rewrite Program 8.5 so that the successive integers are generated in descending order.

```
% between(I,J,K): K is an integer between the integers I and J inclusive.

between(I,J,I) :- I <= J.
between(I,J,K) :- I < J, I1 is I+1, between(I1,J,K).
```

Program 8.5 Generating a range of integers

Ans.

```
% between(I,J,K): K is an integer between the
%     integers I and J inclusive.

between(I,J,J) :- I =< J.
between(I,J,K) :- I < J, J1 is J-1, between(I,J1,K).
```

```
?- between(1,7,X).
X = 7 ;
X = 6 ;
X = 5 ;
X = 4 ;
X = 3 ;
X = 2 ;
X = 1 ;
false.

?-
```

vi. Write a program to find the minimum of a list of integers.

Ans.

```
% min(List,least): least integer from List
%min([M|_],M) :- min([M|_],M).
min([M],M).
min([H|T],H) :-
      min(T,M),
      H =< M.
min([H|T],M) :-
      min(T,M),
      M < H.
```

```
?- ['a02q04.pl'].
true.

?- min([1,2,3,0,12],M).
M = 0 ;
false.

?- min([21,42,-56,23],M).
M = -56 ;
false.

?-
```

vii. Rewrite Program 8.11 for finding the length of a list so that it is iterative. (Hint: Use a counter, as in Program 8.3.)

```
% length(Xs,N): N is the length of the list Xs.

length([X|Xe],N) :- length(Xs,N1), N is N1+1.
Length([],0).
```

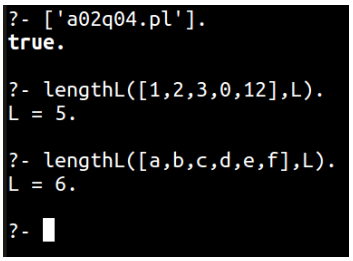
Program 8.11 Finding the length of a list

```
% factorial(N,F): F is the integer N factorial.

factorial(N,F) :- factorial(0,N,1,F).

factorial(I,N,T,F) :-
      I < N, I1 is I+1, T1 is T*I1, factorial(I1,N,T1,F).
factorial(N,N,F,F).
```

Program 8.3 An iterative factorial

Ans.

```
% length_itr(List,A,L): L is length of List,
%     A being accumulator

lengthL([],0).
lengthL([H|T],L) :- length_itr([H|T],0,L).
length_itr([H|T],A,L) :-
      A1 is A+1,
      length_itr(T,A1,L).
length_itr([],A,A).
```



NOTE:

1. All the programs are in files **a02q0X.pl**, where X is Question No s.t. X={1,2,3,4}.

2. Images are also attached as **asn2XY.png**, where X is Question No and Y is Sub-Question No. s.t. X={1,2,3,4} and Y={0,1,2,3,4,...}.