INSA
INSTITUT NATIONAL
DES SCIENCES
APPLIQUÉES
RENNES

Computer Science
Department

Ch 3 - Lists

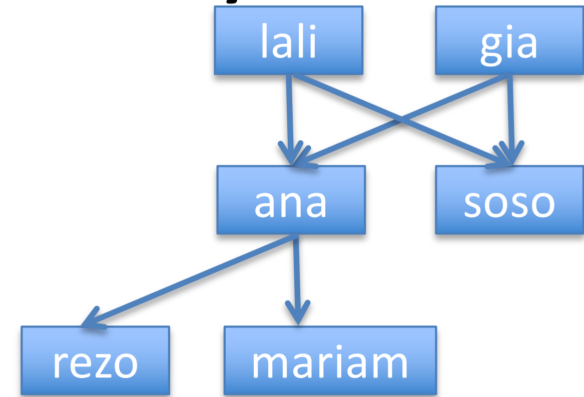
Mireille Ducassé

Last revision April 2024

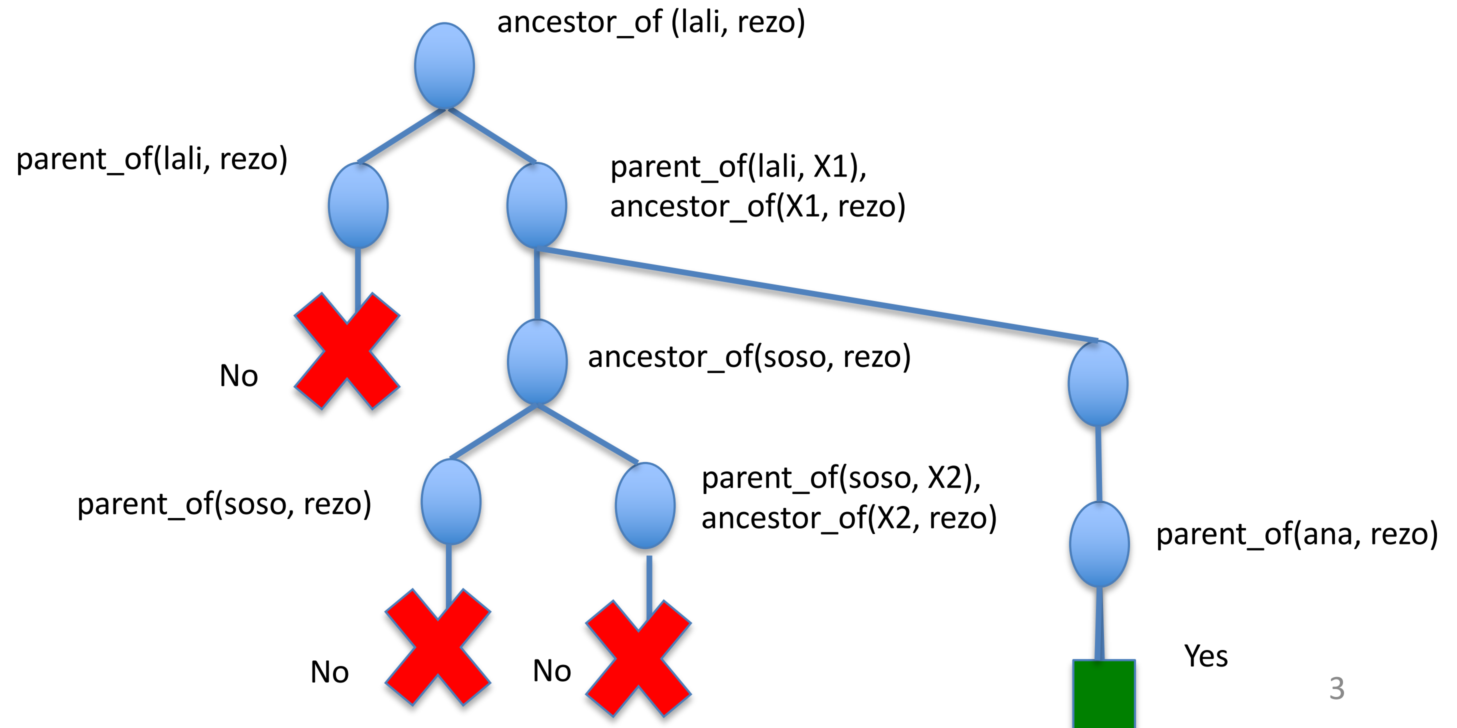
Remember: recusion is key!

lali parent_of ana.
lali parent_of soso.
gia parent_of ana.
gia parent_of soso.
ana parent_of mariam.
ana parent_of rezo.

Where can you see recursion here ?




```
ancestor_of(A, C) :-
      parent_of(A, C).
ancestor_of(A, C) :-
      parent_of(A, X),
      ancestor_of(X, C).
```



Lists

Lists are the most important **structured terms** of Prolog

- A list is a **finite sequence** of elements in between brackets
  - the order is important, as opposed to sets
- The length is flexible
  - as opposed to functor that have a fixed arity
- The elements are not necessarily of the same type
  - as opposed to functional programming
- The empty list ([]) is an important special list

Examples

- [lali, soso, ana, gia, mariam]
- [lali, father_of(soso), ana, X, 2, maia]
- []
- [lali, [soso, [ana], gia], mother_of(mariam), [2, [b,c]], [ ], Z, [2, [b,c]]]
- [a, b, c | LL]

Head and Tail

- A **non-empty** list can be decomposed in two parts
  - **Head**
    - first item of the list
  - **Tail**
    - remaining of the list
    - **always a list**

  Example: [lali, soso, ana, gia, mariam]

  Head: lali

  Tail: [soso, ana, gia, mariam]

- **Built-in operator '|'**

?- [lali, soso, ana, gia, mariam] = [Head | Tail].

Head = lali

Tail = [soso, ana, gia, mariam]

Exercise 3.1

1. ?-[father_of(soso), ana, X, 2, maia] = [Head | Tail].

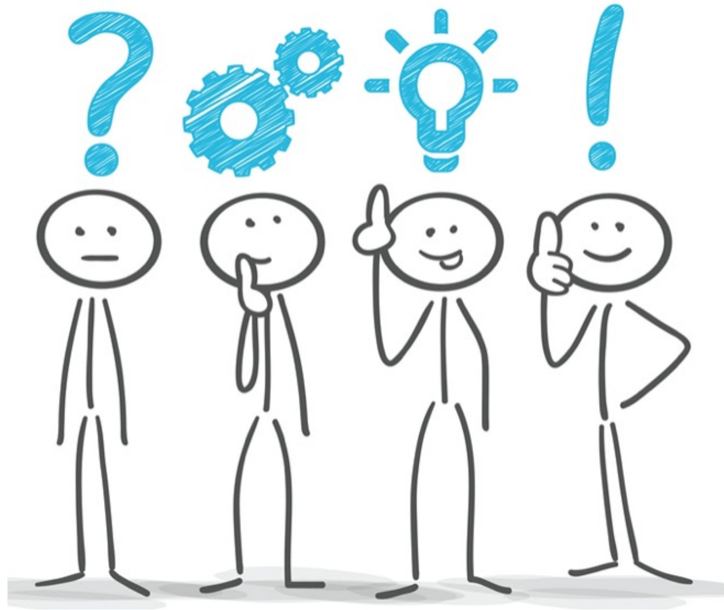
2. ?-[] = [Head | Tail].

3. ?-[[[ana], gia], mother_of(mariam), [2, [b,c]], [ ], Z, [2, [b,c]]]
   = [Foo | Bar].

4. ?-[[ ], ana, X, 2, maia] = [H | T].

5. ?-[a, b, c | LL] = [X, Y | Tail].

6. ?-[mother_of(mariam)] = [Head | Tail].



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.1 (bis)

```
?-[father_of(soso), ana, X, 2, maia] = [Head | Tail].
Head = father_of(soso)
Tail = [ana, X, 2, maia]

?-[] = [Head | Tail].
No

?-[[[ana], gia], mother_of(mariam), [2, [b,c]], [ ], Z, [2, [b,c]]] = [Foo | Bar].
Foo = [[ana], gia]]
Bar = [mother_of(mariam), [2, [b,c]], [ ], Z, [2, [b,c]]]

?-[[ ], ana, X, 2, maia] = [H | T].
H = [ ]
T = [ana, X, 2, maia]

?-[a, b, c | LL] = [X, Y | Tail].
X = a
Y = b
Tail = [c | LL])

?-[mother_of(mariam)] = [Head | Tail].
Head = mother_of(mariam)
Tail =[]
```

Hindsight

- What can you say about the name of the variables ?

- **Note item 5: [a, b, c | LL] = [X, Y | Tail].**

Remarks: **empty list** []

- is a special list without any internal structure
- has neither a head nor a tail
- plays an important role in recursive predicates for list processing in Prolog

Anonymous variable

Suppose we are interested in the second and fourth element of a list

```
?- [X1, X2, X3, X4 | Tail] = [mia, vincent, marsellus, jody, yolanda].
X1 = mia
X2 = vincent
X3 = marsellus
X4 = jody
Tail = [yolanda]
yes
```

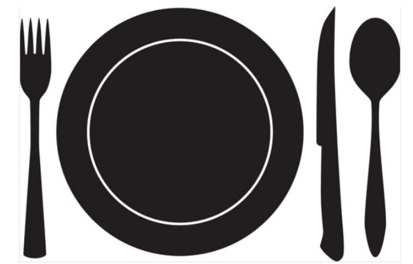
A simpler way to obtain only the information we want:

```
?- [ _, X2,  _, X4 | _ ] = [mia, vincent, marsellus, jody, yolanda].
X2 = vincent
X4 = jody
yes
```

The underscore is the **anonymous variable**

- Used when you need to use a variable, but you are not interested in what Prolog instantiates it to
- Each occurrence of the anonymous variable is **independent**,
  - i.e. can be bound to something different

Back to French menu: Update 3



- Predicates with same name and different arities are a strong **source of bugs**
  - in particular when the program is updated to add new functionalities.
- Update your program so that the french_menu predicate has **arity 1** for all possible structures and that the users only give the order of the courses
  - *Change as little as possible*

Start from the last solution given in previous chapter: with rules to define main_course/1 and with predicate dessert_or_cheese/1

Valid menus (Test cases -> Yes)

?- french_menu([salad, trout_with_rice]).

?- french_menu([trout_with_rice, roquefort]).

?- french_menu([salad, trout_with_rice, roquefort]).

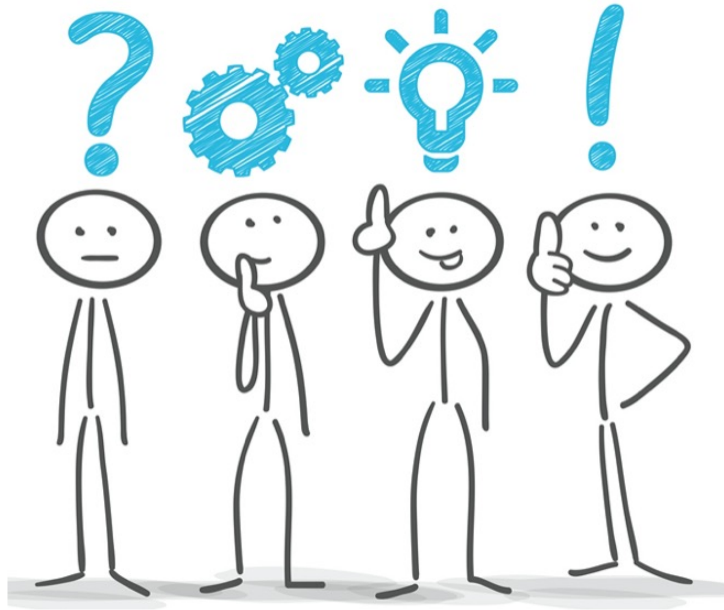
?- french_menu([salad, trout_with_rice, roquefort, cake]).

Invalid menus (Test cases -> No)

?- french_menu([trout_with_rice, salad]).

?- french_menu([salad, trout_with_rice, cake, cake]).

?- french_menu([salad, trout_with_rice, roquefort, cake, coffee]).



Take your time to search, code and test your own program

Then take your time to understand the following solution

French menu: Update 3 (bis)

Facts

```
appetizer(salad).
appetizer(poached_egg).
appetizer(artichoke).

meat_course(steak_with_vegetables).
meat_course(chicken_with_fries).

fish_course(trout_with_rice).
fish_course(salmon_with_eggplant).

veggy_course(falafel_with_rice).
veggy_course(vegetable_lasagna).

dessert(fruit_salad).
dessert(fresh_fruit).
dessert(cake).

cheese(roquefort).
cheese(camembert).
```

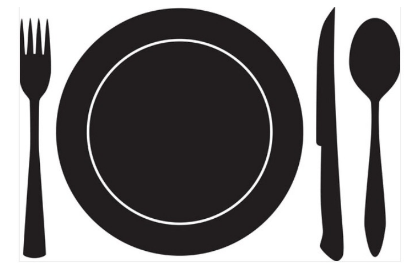
Rules

```
french_menu([A, M]) :-
        appetizer(A),
        main_course(M).
french_menu([M, D]) :-
        main_course(M),
        dessert_or_cheese(D).
french_menu([A, M, D]) :-
        appetizer(A),
        main_course(M),
        dessert_or_cheese(D).
french_menu([A, M, C, D]) :-
        appetizer(A),
        main_course(M),
        cheese(C),
        dessert(D).

main_course(M) :-
        meat_course(M).
main_course(M) :-
        fish_course(M).
main_course(M) :-
        veggy_course(M).

dessert _or_cheese(D) :-
        cheese(D).
dessert _or_cheese(D) :-
        dessert(D).
```

One predicate only for french_menu/1

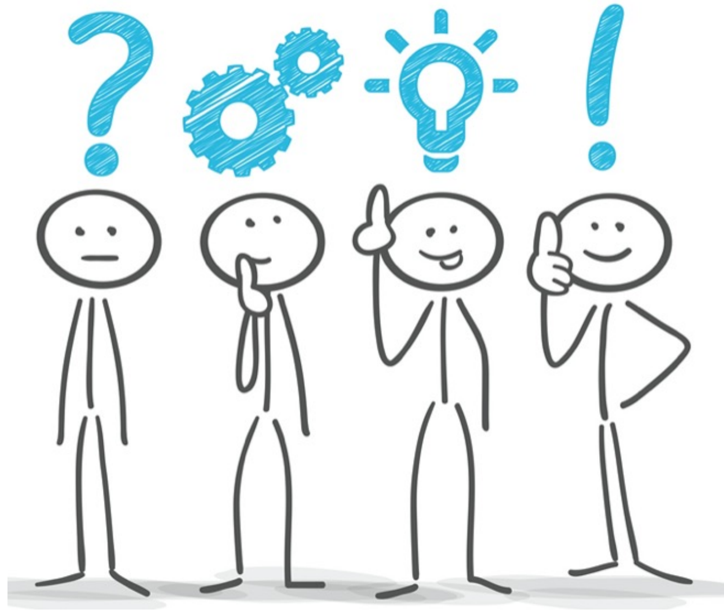
You have to change the way you call french_menu !

French menu: Update 4



- In the previous version there is a lot of code duplication, also a strong **source of bugs**
- Update your program for a strictly equivalent version with less duplication
  - Define and use predicates
    - **menu_main/1** that checks a list starting with a main_course
    - **menu_dessert/1** that checks a list starting with a dessert or cheese

Same test cases as before.



Take your time to search, code and test your own program

Then take your time to understand the following solution

French menu: Update 4 (bis)

```
% Same facts as for the previous version


main_course(M) :-
        meat_course(M).
main_course(M) :-
        fish_course(M).
main_course(M) :-
        veggy_course(M).

dessert _or_cheese(D) :-
        cheese(D).
dessert _or_cheese(D) :-
        dessert(D).
```

```
%french_menu/1
french_menu([A | Tail]) :-
        appetizer(A ),
        menu_main(Tail).
french_menu(M) :-
        menu_main(M).


% french_menu_main/1
menu_main([M | Tail])
        main_course(M),
        menu_dessert(Tail).


% menu_dessert/1  (and/or cheese)
menu_dessert([]).
menu_dessert([A]) :-
        dessert_or_cheese(A).
menu_dessert([C, D]) :-
        cheese(C),
        dessert(D).
```

Remark

- Most of the predicates that we will define in this chapter already exist in the Eclipse Prolog library of predefined predicates
  - http://eclipseclp.org/doc/bips/index.html
- When defining your own version, in order to be able to test it, actually prefix the name of the predicates with ‘my_’
  - eg. **my_member**
  - otherwise compilation error message : “trying to redefine predicate…”

Predicate **member/2**

when given a term X and a list L, tells whether or not X belongs to L

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

Base clause

Recursive clause

It is true that an element X is a member of a list L
if X is the first element of L
or if X is a member of the tail of L.

```
?- member(trudy, [yolanda,trudy,vincent]).
yes
?- member(zed,[yolanda,trudy,vincent]).
no
?- member(X, [yolanda,trudy,vincent]).
X = yolanda;
X = trudy;
X = vincent;
no
```

Exercise 3.2

Write a version of member/2 with anonymous variables when relevant

Exercise 3.2 (bis)

Write a version of member/2 with anonymous variables when relevant

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

Equivalent to

```
member(X, L) :-
    L = [X | _].
member(X, L):-
    L = [_ | T],
    member(X, T).
```

It is true that an element X is a member of a list L
if X is the first element of L
or
if X is a member of the tail of L.

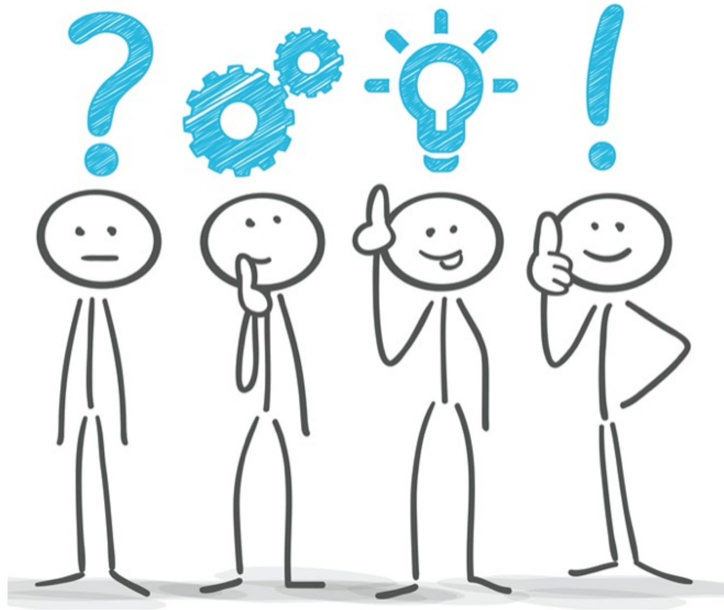
Exercise 3.3: a2b

Write the Prolog predicate a2b/2 that takes two lists as arguments

and succeeds

- if the first argument is a list of a's, and
- if the second argument is a list of b's of **exactly the same length**

```
?- a2b([a,a,a,a],[b,b,b,b]).
yes
?- a2b([a,a,a,a],[b,b,b]).
no
?- a2b([a,c,a,a],[b,b,b,t]).
no
?- a2b([a,a,a,a,a], X).
X = [b,b,b,b,b]
yes
?- a2b(X,[b,b,b,b,b,b,b]).
X = [a,a,a,a,a,a,a]
yes
?- a2b([], []).
No
```



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.3 : a2b (bis)

Write the Prolog predicate a2b/2 that takes two lists as arguments and succeeds

- if the first argument is a list of a's, and
- the second argument is a list of b's of exactly the same length

```
?- a2b([a,a,a,a],[b,b,b,b]).
yes
?- a2b([a,a,a,a],[b,b,b]).
no
?- a2b([a,c,a,a],[b,b,b,t]).
no
?- a2b([a,a,a,a,a], X).
X = [b,b,b,b,b]
yes
?- a2b(X,[b,b,b,b,b,b,b]).
X = [a,a,a,a,a,a,a]
yes
```

```
a2b([a], [b]).
a2b([a | L1],[b | L2]):-
    a2b(L1, L2).
```

Trace 1

```
a2b([a], [b]).
a2b([a | L1],[b | L2]):-
    a2b(L1, L2).
```

```
?- a2b([a, a, a, a], [b, b, b, b]).
  (1) 1 CALL  a2b([a, a, a, a], [b, b, b, b])
  (1) 1 NEXT  a2b([a, a, a, a], [b, b, b, b])
  (2) 2 CALL  a2b([a, a, a], [b, b, b])
  (2) 2 NEXT  a2b([a, a, a], [b, b, b])
  (3) 3 CALL  a2b([a, a], [b, b])
  (3) 3 NEXT  a2b([a, a], [b, b])
  (4) 4 CALL  a2b([a], [b])
  (4) 4 *EXIT  a2b([a], [b])
  (3) 3 *EXIT  a2b([a, a], [b, b])
  (2) 2 *EXIT  a2b([a, a, a], [b, b, b])
  (1) 1 *EXIT  a2b([a, a, a, a], [b, b, b, b])
```

Trace 2

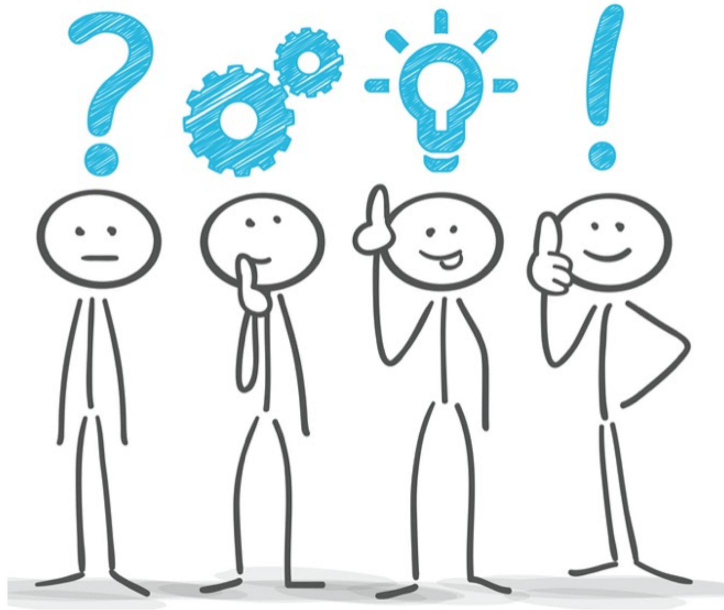
```
a2b([a], [b]).
a2b([a | L1],[b | L2]):-
    a2b(L1, L2).
```

```
?- a2b([a, a, a, a], B).
  (1) 1 CALL  a2b([a, a, a, a], B)
  (1) 1 NEXT  a2b([a, a, a, a], B)
  (2) 2 CALL  a2b([a, a, a], _283)
  (2) 2 NEXT  a2b([a, a, a], _283)
  (3) 3 CALL  a2b([a, a], _370)
  (3) 3 NEXT  a2b([a, a], _370)
  (4) 4 CALL  a2b([a], _457)
  (4) 4 *EXIT  a2b([a], [b])
  (3) 3 *EXIT  a2b([a, a], [b, b])
  (2) 2 *EXIT  a2b([a, a, a], [b, b, b])
  (1) 1 *EXIT  a2b([a, a, a, a], [b, b, b, b])

B = [b, b, b, b]
```

Exercise 3.4: double/2

Write a program for double(List1, List2) where every element of the first list appears twice in a row in the second list

```
?- double([1,2], [1,1, 2,2]).
yes

?- double([a, b, c], [a, a, b, b, c, c]).
yes

?- double([], []).
No
```



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.4: double/2 (bis)

Write a program for double(List1, List2) where every element of the first list appears twice in a row in the second list

```
?- double([1,2], [1,1, 2,2]).
yes

?- double([a, b, c], [a, a, b, b, c, c]).
yes

?- double([], []).
No
```

```
double([X], [X,X]).
double([X | T1], [X,X | T2]) :-
    double(T1, T2).
```

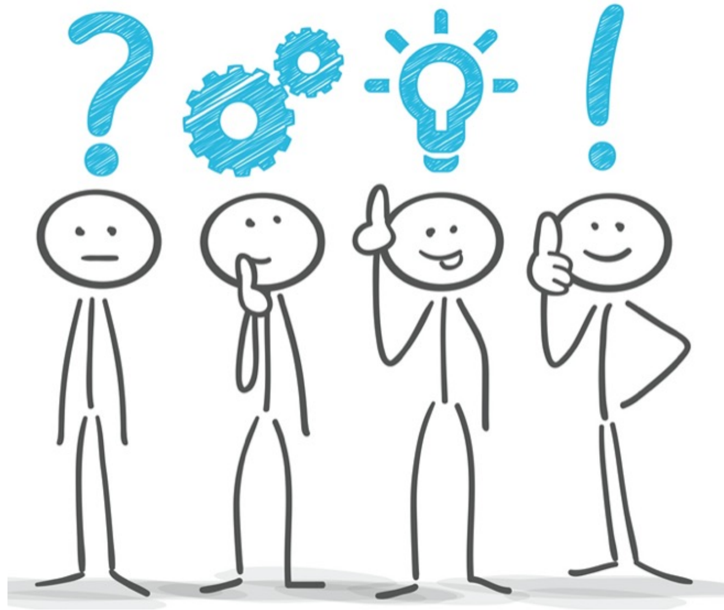
Exercise 3.5: deleteXs/3

Write a program for **deleteXs(X, List1, List2)** where List2 is List1 with all the Xs deleted.

?- deleteXs(3, [1, 2, 3, 4, 3, 5], [1, 2, 4, 5]).
yes

?- deleteXs(3, [1, 2, 3, 4, 3, 5], L).
L = [1, 2, 4, 5]

What happens if you try the following ? Why ?
?- deleteXs(3, L, [1, 2, 4, 5]).



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.5: deleteXs/3 (bis)

Write a program for **deleteXs(X, List1, List2)** where List2 is List1 with all the Xs deleted.

?- deleteXs(3, [1,2,3,4,3,5], [1,2,4,5]).

yes

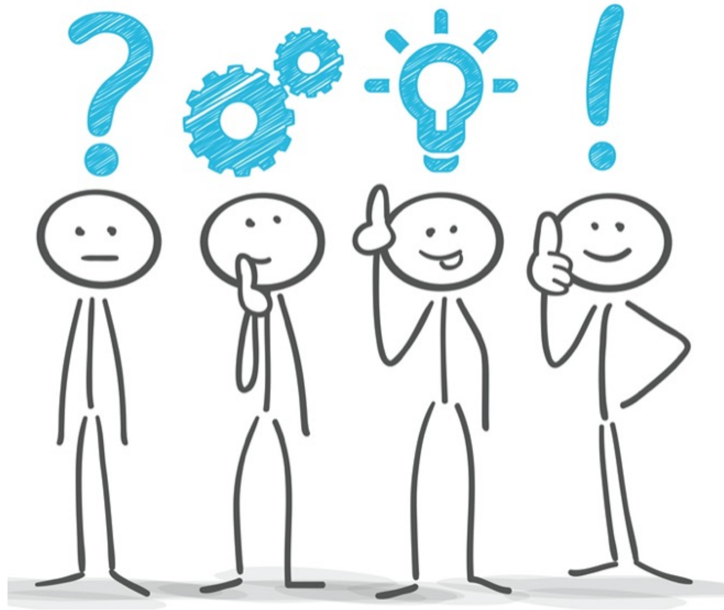
Try all modes in queries.

```
deleteXs(_X, [], []).
deleteXs(X, [X | L1], L2) :-
    deleteXs(X, L1, L2).
deleteXs(X, [Z | L1], [Z | L2]) :-
    X \= Z,
    deleteXs(X, L1, L2).
```

Exercise 3.6: substitute/4

Write a Prolog program for substitute(X, Y, L1, L2) where L2 is the result of substituting Y to all occurrences of X in L1

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

?- substitute(a, x, [a,b,a,c], [x,b,a,c]).
no

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

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



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.6: substitute/4 (bis)

Write a Prolog program for substitute(X, Y, L1, L2) where L2 is the result of substituting Y to all occurrences of X in L1

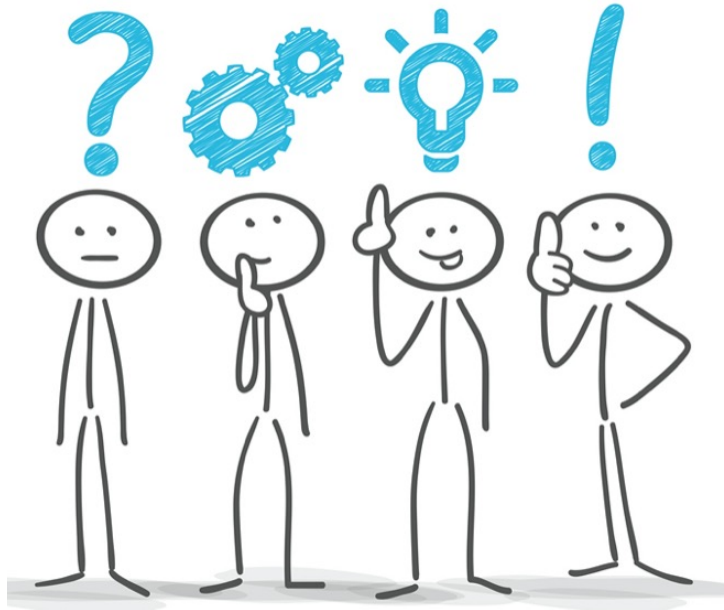
```
?- substitute(a, x, [a,b,a,c], [x,b,x,c]).
yes
?- substitute(a, x, [a,b,a,c], [x,b,a,c]).
no
?- substitute(a, x, [a,b,a,c], L).
L = [x, b, x, c]
?- substitute(a, x, L, [x,b,x,c]).
L = [a, b, a, c]
```

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

Exercise 3.7: append/3

- Write a Prolog predicate that concatenates lists: append(L1, L2, L3) is true if list L3 is the result of concatenating the lists L1 and L2 together
- Test cases

```
?- append([a, b, c, d],[3, 4, 5], [a, b, c, d, 3, 4, 5]).
yes
?- append([a, b, c, d], [3, 4, 5], [a, b, c, 3, 4, 5]).
no
?- append(X, Y,  [a, b, c, d]).                    %splitting up lists !
X=[ ]
X=[a]
X=[a,b]
X=[a,b,c]
X=[a,b,c,d]
no
```



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.7: append/3 (bis)

- Write a Prolog predicate that concatenates lists: append(L1, L2, L3) is true if list L3 is the result of concatenating the lists L1 and L2 together
- Test cases

```
?- append([a, b, c, d], [3, 4, 5], [a, b, c, d, 3, 4, 5]).
yes
?- append([a, b, c, d], [3, 4, 5], [a, b, c, 3, 4, 5]).
no
?- append(X, Y,  [a, b, c, d]).                    %splitting up lists !
X=[ ]
X=[a]
X=[a,b]
X=[a,b,c]
X=[a,b,c,d] no
```

- Code

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

Exercise 3.8: prefix/2

Using append/3, write a Prolog predicate that computes a prefix of a list: **prefix/2.** A list P is a prefix of some list L when there is some list such that L is the result of concatenating P with that list.

- Test case

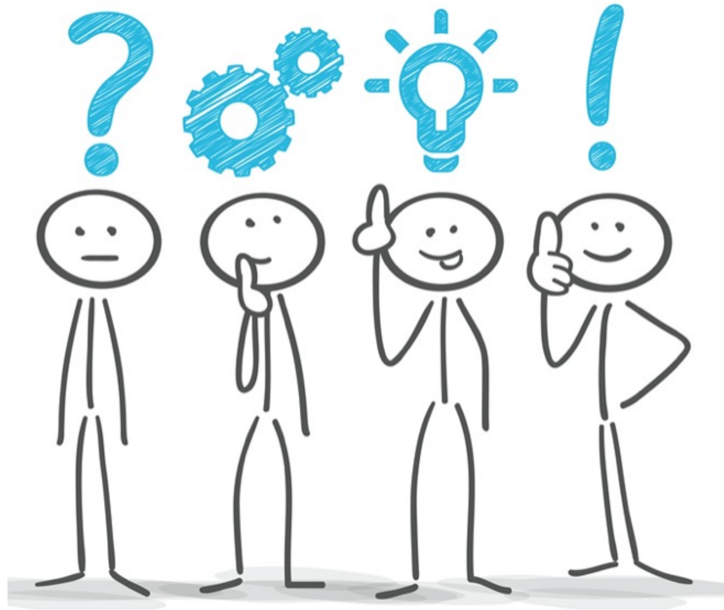
?- prefix(X, [a, b, c, d]).

X=[ ];
X=[a];
X=[a,b];

X=[a,b,c];

X=[a,b,c,d];

no



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.8: prefix/2 (bis)

Using append/3, write a Prolog predicate that computes a prefix of a list: **prefix/2.** A list P is a prefix of some list L when there is some list such that L is the result of concatenating P with that list.

- Test case

```
?- prefix(X, [a, b, c,  d]).
X=[ ];
X=[a];
X=[a, b];
X=[a, b, c];
X=[a, b, c, d];
no
```

- Code

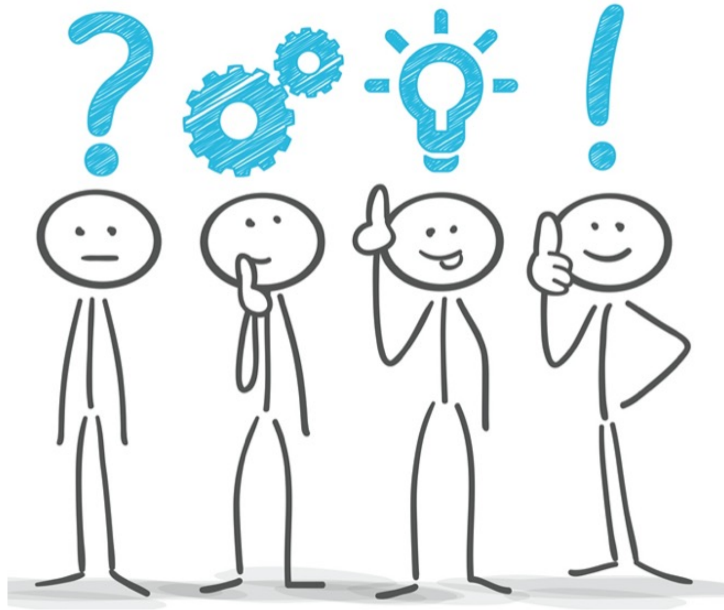
```
prefix(P, L):-
    append(P, _, L).
```

Exercise 3.9: suffix/2

Same exercise for suffix/2

- Test case

```
?- suffix(X, [a, b, c, d]).
X=[a, b, c, d];
X=[b, c, d];
X=[c, d];
X=[d];
X=[];
no
```



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.9: suffix/2 (bis)

Same exercise for suffix/2

- Test case

```
?- suffix(X, [a, b, c, d]).
X=[a, b, c, d];
X=[b, c, d];
X=[c, d];
X=[d];
X=[];
no
```

- Code

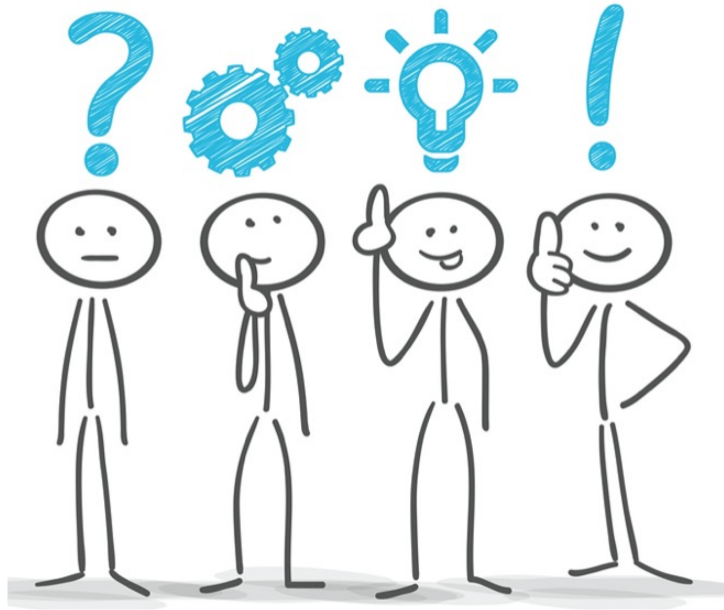
```
suffix(S, L):-
    append(_, S, L).
```

Exercise 3.10: sublist/2

Write a predicate that finds sub-lists of lists

The sub-lists of a list L are simply the prefixes of suffixes of L

- Specify test cases



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.10: sublist/2 (bis)

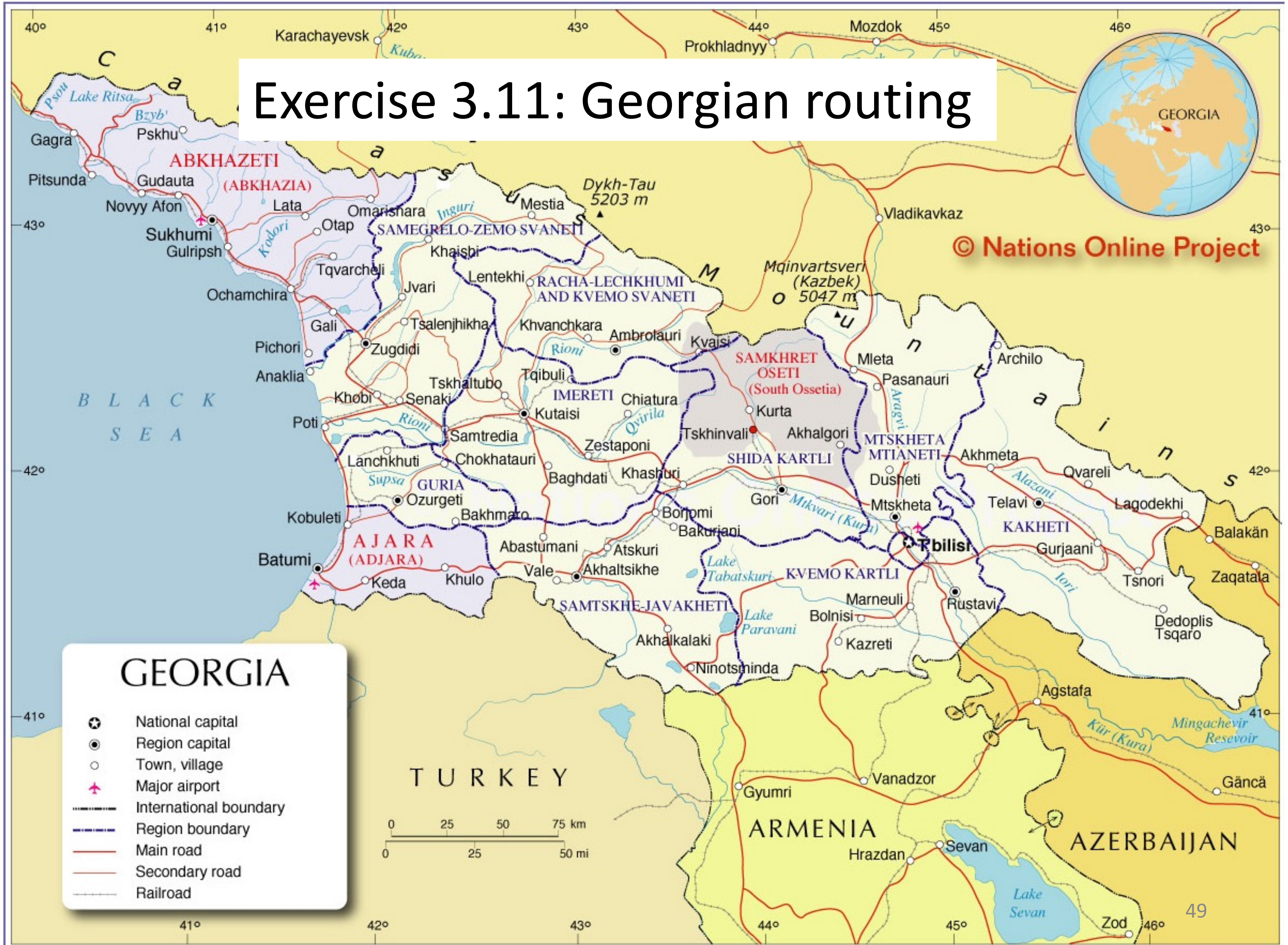
Write a predicate that finds sub-lists of lists

The sub-lists of a list L are **prefixes of suffixes** of L

- Specify test cases
- Code

```
sublist(Sub, List):-
    suffix(Suffix, List),
    prefix(Sub, Suffix).
```

Exercise 3.11: Georgian routing

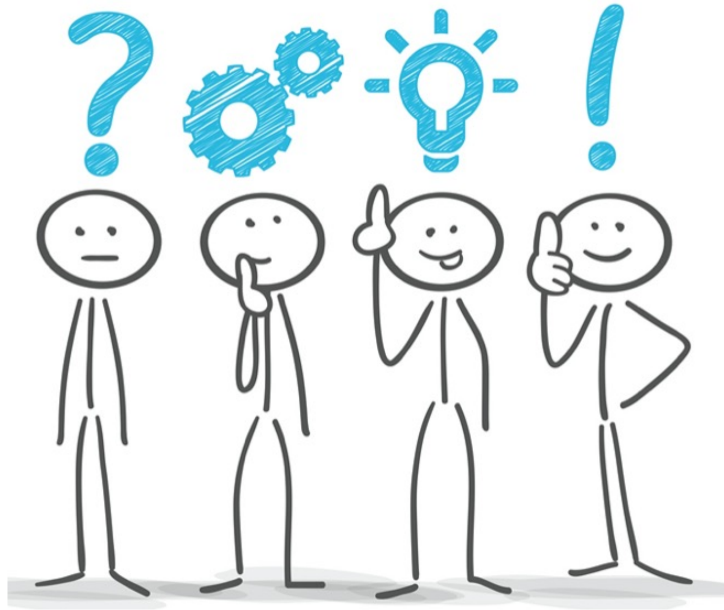


Exercise 3.11: Georgian routing

```
road(tbilisi, rustavi).
road(tbilisi, mtskheta).
road(tbilisi, gurjaani).
road(tbilisi, akhmeta).
road(mtskheta, gori).
road(gurjaani, telavi).
road(akhmeta, telavi).
road(gori, khashuri).
```

- Write predicate route/2 to be able to discover, from where to where we can travel by road in the direction given by those facts (Tbilisi centered).
  - Where can we drive from tbilisi ?
  - From where can we go to telavi ?

Note that a version that takes into account that roads go in both directions will be addressed later



Take your time to search, code and test your own program

Then take your time to understand the following solution

Exercise 3.11: Georgian routing (bis)

```
road(tbilisi, rustavi).
road(tbilisi, mtskheta).
road(tbilisi, gurjaani).
road(tbilisi, akhmeta).
road(mtskheta, gori).
road(gurjaani, telavi).
road(akhmeta, telavi).
road(gori, khashuri).
```

```
route(X, Y) :-
    road(X, Y).
route(X, Y) :-
    road(X, Z),
    route(Z, Y).
```

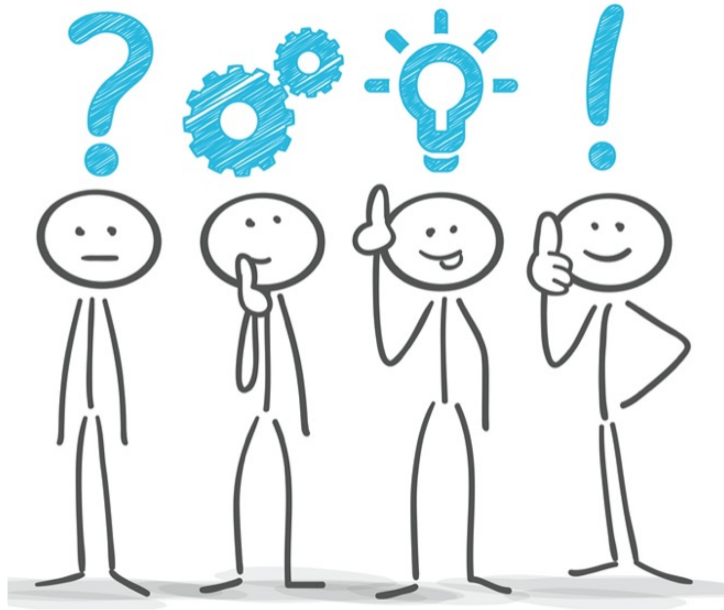
Hindsight

- Is this version of route/2 program very different from the parent/ancestor program ?
  - every program that traverses a **directed graph** will look the same

Ex 3.11 Georgian routing: update 1

```
route(tbilisi, rustavi).
route(tbilisi, mtskheta).
route(tbilisi, gurjaani).
route(tbilisi, akhmeta).
route(mtskheta, gori).
route(gurjaani, telavi).
route(akhmeta, telavi).
route(gori, khashuri).
```

- Write a more sophisticated version that **stores the intermediate cities in a list**



Take your time to search, code and test your own program

Then take your time to understand the following solution

Ex 3.11 Georgian routing: update 1 (bis)

```
route(tbilisi, rustavi).
route(tbilisi, mtskheta).
route(tbilisi, gurjaani).
route(tbilisi, akhmeta).
route(mtskheta, gori).
route(gurjaani, telavi).
route(akhmeta, telavi).
route(gori, khashuri).
```

```
route(X, Y, []) :-
    road(X, Y).
route(X, Y, [ Z | Int]) :-
    road(X, Z),
    route(Z, Y, Int).
```

Hindsight

- The most important design pattern for list processing:

```
do_list([], <base result>).
do_list([Head | Tail], [Head_Res |Tail_Res]) :-
    do_one(Head , Head_Res),
    do_list(Tail, Tail_Res).
```

sometimes
do_list([X], [<base_result>]).

- End result is concatenated **at the end of the recursions**

Recursion

- Replaces iteration of imperative programming
- Much safer to program with
  - … once well understood ☺