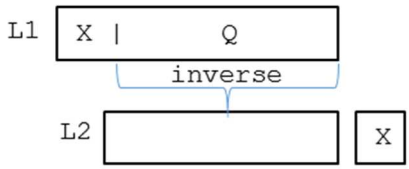
4.4. Application à la manipulation de listes *(VIDEO-PPT 3.3)*

➢ *Inversion d'une liste*

Version enveloppée :

Le prédicat à deux arguments `inverse(L1, L2)` est vrai si `L2` est la liste inverse de `L1`.

- Si `L1` est vide : `L2` est vide
- Si `L1` est non vide, `L1` est de la forme `[X|Q]`



D'où le programme :

```
1.  inverse([], []).
2.  inverse([X|Q], R) :- inverse(Q, QL), conc(QL, [X], R).
```

Des questions :

```
| ?- inverse([0, 1, 2], L).
L = [2,1,0]

| ?- inverse(L,[2,1,0]).
```

Du fait du fonctionnement de l'interpréteur, cette dernière question génère une exécution qui boucle. La récursivité est prévue pour que le premier argument diminue à chaque appel pour atteindre `[]` et la sélection de la règle 1. Il faut donc que le premier argument soit instancié.

La complexité temporelle de ce programme a été étudiée en cours d'algorithmique. Pour une liste à *n* éléments, le prédicat `conc` est exécuté *n* fois, or celui-ci ajoute un élément en queue de liste, cet ajout étant linéaire, l'algorithme est en $O(n^2)$.

Version terminale :

Dans cette approche, la liste inverse de `L` est construite au fur et à mesure (processus itératif) du parcours de `L`. Il faut donc un argument auxiliaire pour propager la liste partielle progressivement construite. À chaque étape, le 1er élément de la liste est inséré en tête dans la liste auxiliaire :

Liste	Liste auxiliaire
[1, 2]	[]
[2]	[1]
[]	[2, 1]

D'où le programme :

```
1.  inverse([], L, L).
2.  inverse([X|Q], L, R) :- inverse(Q, [X|L], R).
```

Lors du 1er appel du prédicat `inverse`, le 1er argument est unifié à la liste à inverser et le second argument (liste auxiliaire) doit être initialisé à `[]`.

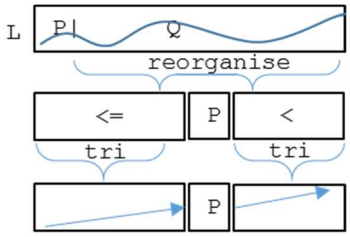
```
| ?- inverse([0,1,2], [], L).
L = [2,1,0]
```

Cet algorithme est en complexité linéaire. La différence de complexité entre la version enveloppée et la version terminale est dûe au fait que la construction de la liste inverse à une liste `L` est effectuée dans le premier cas en traitant les élements de `L` de la droite vers la gauche (ce qui induit des insertions en queue dans la liste inverse) alors que dans le second cas les éléments de `L` sont traités de la gauche vers la droite (ce qui entraine des insertions en tête dans la liste auxiliaire).

- *Tri rapide*

Version enveloppée :

Le schéma suivant décrit le principe de construction du tri rapide par induction structurelle :



D'où le programme :

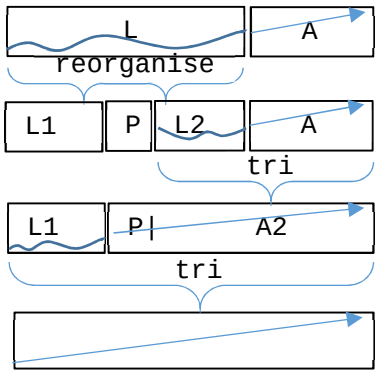
```
1.  triRapide([], []).
2.  triRapide([P|Q], R) :- reorganise(Q, P, L1, L2),
                           triRapide(L1, L1trie),
                           triRapide(L2, L2trie),
                           conc(L1trie, [P | L2trie], R).
```

Avec :

```
1.  reorganise([], P, [], []).
2.  reorganise([X|Q], P, [X|L1], L2) :- X =< P, reorganise(Q, P, L1, L2).
3.  reorganise([X|Q], P, L1, [X|L2]) :- X > P, reorganise(Q, P, L1, L2).
```

Version terminale :

Cette construction est probablement plus difficile car moins intuitive. Il faut se placer dans une situation où une partie du travail a été réalisée. Elle est caractérisée par une liste auxiliaire triée `A` dont les éléments sont supérieurs ou égaux à ceux à la liste `L` des éléments restants non triés. Le principe du tri se schématise de la manière suivante :



D'où le programme :

```
1.  triRapide([], L, L).
2.  triRapide([P|Q], A, R) :- reorganise(Q, P, L1, L2),
                              tri(L2, A, A2),
                              tri(L1, [P|A2], R).
```

Question :

```
| ?- triRapide([2,0,4], [], R).
R = [0,2,4]
```

➢ *MiniTP : Fibonacci*

Définir les versions récursives enveloppée et terminale du prédicat `fib(N,R)` qui calcule dans `R` le résultat du terme F_n de la suite de Fibonacci définie par :

$F_0 = F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ *pour* $n \geq 2$

Tester les deux formes de récursivité aux rangs 5 et 25…

Note $_1$: En passant par un prédicat auxiliaire :

- l'accumulateur peut être un couple décomposé sur ses deux atomes
- le cas d'arrêt à 0 est plus simple à tester qu'à *n*

Note $_2$: Vous pourrez utiliser les pseudo-prédicats :

`var is expr` qui unifie la variable `var` à `expr` évaluée

`expr1 =\= expr2` qui réussit si `expr1` est différente de `expr2`