5 – Fonctionnement d'un programme recursif

Les fonctions récursives permettent de simplifier l'écriture de nombreux problèmes.
Nous verrons ici leur fonctionnement et une méthodologie d'écriture et d'analyse de ces fonctions.

1 – Introduction à la récursivité

1 – Définition et premiers exemples

Une fonction récursive est une fonction qui s'appelle elle même.
Son opposé s'appelle une fonction itérative.

Une fonction récursive simple s'écrit sous la forme :

```
def func(args):
    if condition d'arrêt:
        return valeur
    appel récursif
```

Pour écrire une fonction récursive il faut :
- Déterminer le type de données à renvoyer
- Déterminer pour quelle(s) valeur(s) de l'argument le problème est résolu, puis écrire la condition d'arrêt
- Déterminer de quelle manière la taille du problème est réduite
  (arguments entier qui décroit strictement, liste dont la taille diminue, etc...)
- Ecrire l'appel récursif en prenant garde à ce que le type de données qu'il renvoie soit cohérent avec celui renvoyé par la condition d'arrêt

Si un problème donné peut se ramener à un problème plus "petit", jusqu'à un problème élementaire que l'on sait traiter, alors ce problème peut être traiter par une programmation récurrsive.

Un premier exemple serait la detection d'un palindrome (mot qui se lit indifféremment de gauche à droite comme de droite à gauche).

Ex : elle / ressasser

la détection d'un palindrome est un problème recursif. Une chaine de caractèrs a est un palindrome si et seulement si :

- elle comporte zéro ou une seule lettre :
  - ex : "" ou "a"

- sa première et sa dernière lettre sont identiques, et le reste de la chaine – plus petite – forme un palindrome

```
def estPalindrome(s):
    if len(s) <= 1: #terminaison
        return True
    else:
        return s[0] == s[-1] and estPalindrome(s[1 : -1])
        # s[0] == s[- ] vérifie si la première et la dernière lettre sont identiques
        # s[1 : - ] est la chaîne s privé de sa première et dernière lettre
```

La question que l'on pourrait se poser se poser est :
Quand est-ce que le programme s'arrête ?

Il lui faut une terminaison, autrement dit, un cas ou le programme ne s'appelle lui même
Il faut aussi que la taille du problème diminue à chaque étape

Prenons l'exemple de la factorielle.
"Produit des nombres entiers strictement positifs inférieurs ou égaux à *n*"
5! = 1 x 2 x 3 x 4 x 5 = 120

Si je vous demande d'écrire la fonction factorielle.

Ex :

```
def facto_moi(n):
    if n == 0:
        return 1
    else:
        res = 1
        for i in range(2, n + 1):
            res = res * i
        return res
```

Fonction factorielle en récursif.

Ex :

```
def facto(n):
    if n == 0:
        return 1
    else:
        return n * facto(n-1)
```

Un 2° exemple simple pour expliquer la récursivité, c'est l'inversion

Ex : inversion([2, 6, 4, 9] → [9, 4, 6, 2])

Pour inverser une liste de longueur 0 ou 1, il n'y à rien à faire, la liste est déjà sont propre inverse

Pour inverser une liste de longueur n, il suffit de savoir le faire avec une liste de longueur n-1. On inverse la liste des n-1 derniers éléments et on ajoute le premier à la fin

```
def inversion(L) :
    if len(L) <= 1 :
        return L
    else :
        return inversion(L[1 :]) + [L[0]])
```

Si la longueur est 0 ou 1, on renvoie la liste elle même

2 – Pile d'exécution

Lors de l'appel d'une fonction recursive, une structure de pile est utilisée en interne.

Une pile fonctionne comme un empilement d'objets (une pile d'assiette). Il est possible d'ajouter une assiette sur le haut de la pile, ou en enlever une (on dit dépiler), mais on ne peut pas accéder a une assiette qui n'est pas en haut de la pile. C'est pareil en informatique

La pile utilisée, aussi appelée pile d'exécution, est de taille limitée. Au-delà d'un certain nombre d'appel récursif (1000 par défaut en python), il y à une erreur de dépassement de la taille de la pile (remplacer le n-1 par n+1 dans l'ex sur la factorielle), stack overflow.
Cette pile contient une trace de tous les appels de fonction qui ne sont pas pas encore terminés.

Description de l'exécution de inversion([9, 2, 7]) :

```
inversion([9, 2, 7]) :
    inversion([2, 7]) :
        inversion([7]) :
        → [7] #terminaison
    → [7] + [2]
→ [7, 2] + [9]
```

Dans un premier temps, la fonction est plus grande que 1, on va donc passer au else ;
Il va travailler sur la sous chaîne [2, 7], puis sera rappeler, la longueur n'est toujours pas égale a 0 ou 1 donc on va rappeler la fonction. Cette fois, la fonction est égale à 1, elle va donc renvoyer [7]
Le L[0] va donc être le 2, on concatène et on obtient la liste [7, 2], on le renvoie puis on concatène à nouveau et l'on obtient bien la liste [7, 2, 9].

Cette exemple s'appelle une récursivité linéaire.

Toutes boucle FOR peut être remplacée par une fonction récursive, car l'on va répéter un certains nombres de fois l'appel de la fonction, comme le fait une boucle FOR

Passer d'une fonction récursive à une fonction itérative s'appelle la dérécursivation

On va parcourir la liste à l'envers (ligne du FOR)

```
def inversion(L) :
    res = []
    for i in range(len(L)-1, -1, -1) : # de n-1 à 0
        res = res + [L[i]]
    return res
```

2 – La programmation récursive est-elle efficace ?

La réponse n'est pas très clair, car cela va dépendre...
Les algos récursif sont souvent plus gourmands en ressources (mémoire) que leurs équivalents itératifs
Pour la fonction inversion, par exemple, il va falloir stocker pleins d'informations intermédiaires en mémoire, afin de pouvoir les redérouler pour arriver à la fin.
Pire encore, si l'on ne fait pas attention au nombre d'appels.

Un exemple va être le triangle de Pascal

	p=0	1	2	3	4	5
n=0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

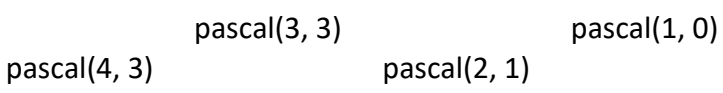
Chaque ligne commence et finit par 1, tout autre coefficient est la somme de son voisin du dessus et du voisin de gauche de ce dernier.

ligne n 0 à 5
colonne p 0 à 5

On voit bien une programmation récursive évidente

```
def pascal(n, p) : # coeff en ligne n et colonne p
    if p == 0 or n == p :
        return 1
    else :
        return pascal(n-1, p) + pascal(n-1, p-1)
```

Voilà ce qu'il se passe quand on va effectuer le calcul :



```
                                pascal(3, 2)                    pascal(1, 1)
                                                pascal(2, 2)

pascal(5, 3)                                    pascal(2, 0)
                                pascal(3, 1)                    pascal(1, 0)
                                                pascal(2, 1)
                                                                pascal(1, 1)
                        pascal(4, 2)
                                                                pascal(1, 0)
                                                pascal(2, 1)
                                pascal(3, 2)                    pascal(1, 1)
                                                pascal(2, 2)
```

pascal(5, 3) va être la somme de pascal(4, 3) + pascal(4, 2), pascal(3, 3) va être la somme de pascal(3, 3) + pascal(3, 2)
pascal(3, 3) ne va appeler personne car il est en bout de ligne, mais pascal(3, 2) va appeler pascal(2, 1) + pascal(2, 2) (se termine car en bout de ligne)

Ce que l'on peut remarquer c'est qu'il y'à beaucoup de répétitions par ex pascal(1, 0), pascal(3, 2) est calculé 2 fois...

Si l'on devait utiliser des valeurs plus grande, on se rends bien compte que le nombre d'appels va exploser et ralentir énormément notre programme.

Une programmation itérative va être moins simple, mais plus efficace.
On construit le triangle ligne par ligne :

```
def pascalIt(n, p)
    Cn = [1] #ligne de départ
    for i in range(1, n+1) :
        Cn_plus_un = [1] #premier coeff
        for j in range(1, i) :
            #coeff de la somme des 2 de la ligne précédente
            Cn_plus_un.append(Cn[j]+Cn[j-1])
        Cn_plus_un.append(1) #dernier coeff
        # pour ne pas encombrer la mémoire la ligne Cn_plus_un
        # va devenir la ligne Cn du prochain tour de boucle
        Cn = Cn_plus_un
    return Cn[p] # on extrait le coeff p de la ligne Cn
```

Cn est la première ligne, et Cn_plus_un est la ligne suivante

Si l'on compare :

```
recursif :      pascal(32, 16)          =       371s
                pascal(1024, 512)       =       RecursionError
```

itératif : pascal(32, 16) = 0,001s
pascal(1024, 512) = 0,2s

En python, le nombre d'appel récursif est limité

Il y'à quand même des cas ou la récursivité est plus efficace que leur version itérative

Par exemple, le tri fusion (version récursive du tri)

Ex : tri([5, 1, 8, 3, 0, 2, 7]) → [0, 1, 2, 3, 5, 7, 8]

Forme récursive :

Une liste de 1 ou 0 nombre est déjà triée
Si la liste est scindée en 2 sous-listes déjà triées, il est rapide d'obtenir la liste complète triée par « fusion »

C'est un exemple de la stratégie « diviser pour régner »

La fusion, c'est comment 2 sous-listes triées donnent une liste complète triée.
Ex : fusion([1, 3, 5, 8, 9], [0, 2, 7]) → []

Les listes étant déjà triées, afin des les fusionner, on va regarder le premier éléments de chaque listes et on va les comparer entre eux :

fusion([1, 3, 5, 8, 9], [0, 2, 7]) → [0]

Entre 0 et 1, le plus petit est 0, on le met au début de notre liste.
Une fois mis en début de liste, on a plus a le prendre en compte, on va décaler

fusion([1, 3, 5, 8, 9], [0, 2, 7]) → [0, 1]
fusion([1, 3, 5, 8, 9], [0, 2, 7]) → [0, 1, 2]
fusion([1, 3, 5, 8, 9], [0, 2, 7]) → [0, 1, 2, 3]
fusion([1, 3, 5, 8, 9], [0, 2, 7]) → [0, 1, 2, 3, 5]
fusion([1, 3, 5, 8, 9], [0, 2, 7]) → [0, 1, 2, 3, 5, 7]
Une des 2 listes est complètement utilisée, je peux ajouter la fin de l'autre.
Fusion([1, 3, 5, 8, 9], [0, 2, 7]) → [0, 1, 2, 3, 5, 7, 8, 9]

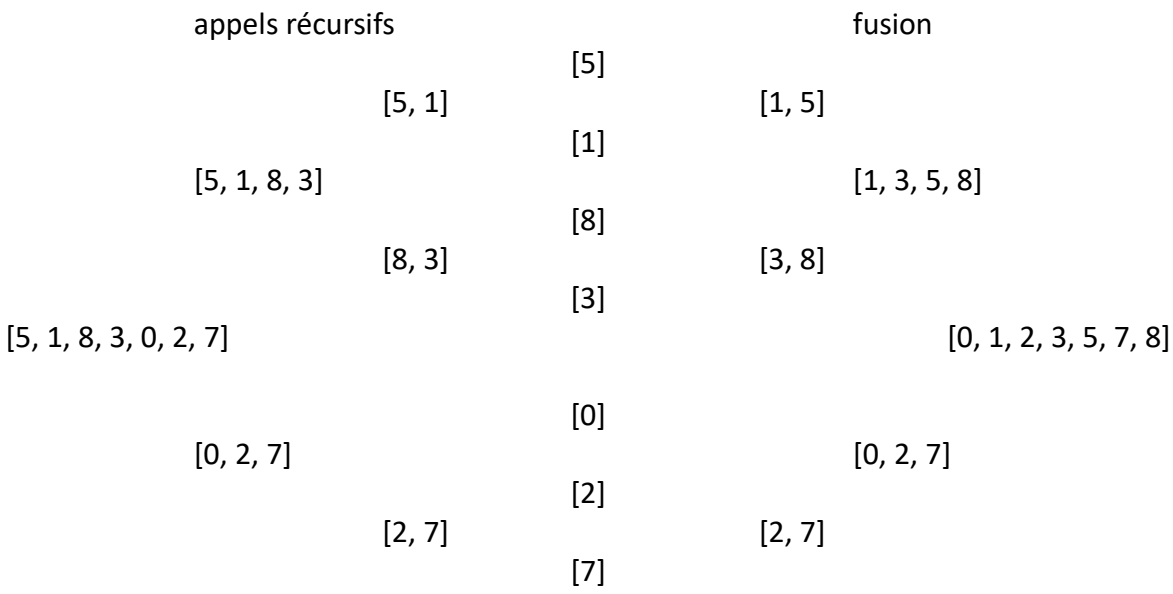
Sans entrer dans les détails :

```
def tri(L) :
    n = len(L)
    if n <= 1 :
        return L
    else :
        return fusion(tri((L[0 : n//2]), tri(L[n//2 : n]))
        # On coupe la liste en 2
```

L = [2, 3, 8, 9, 7] # n = 5
L[0 : n//2] → [2, 3, 8]
L[n//2 : n] → [9, 7]

L = [2, 3, 8, 9, 7, 5] # n = 6
L[0 : n//2] → [2, 3, 8]
L[n//2 : n] → [9, 7, 5]

// reste de la division euclidienne



3 – Pourquoi utiliser la récursive ?

La programmation récursive est-elle plus simple que la programmation itérative ?

Sur les exemples vue, a partir du moment ou le problème est expliqué de manière récursive (triangle de Pascal), oui, car elle est plus simple, plus naturel

Peut-on toujours passer facilement de l'une à l'autre ?

Ça dépends mais en général : non

La plupart des traitements itératifs simples sont facilement traduisible sous forme récursive (exemple du FOR)
Il arrive même qu'un problème ait une solution récursive très simple alors qu'il est très difficile d'en trouver une solution itérative

Voir exemple de la recherche d'un chemin dans un labyrinthe.

https://www.lumni.fr/video/une-introduction-a-la-recursivite