Chapitre 11 :
Récursivité

I Généralités

1.1 Définition

Définition

Une fonction récursive est une fonction s'appelant elle même.

Une fonction récursive doit contenir une condition d'arrêt qui doit toujours être atteint, sans cela, la fonction ne s'arrêterait pas. Il arrive fréquemment que la condition d'arrêt soit un cas de base, comme dans les récurrences en mathématiques.

La structure classique d'une fonction récursive est la suivante :

```
def f(arguments) :
    if condition d'arrêt :
        return valeur
    else :
        appel récursif
```

La récursivité permet, dans certaines situations, d'avoir des algorithmes plus naturels et plus concis.

➪ **Exemple 1 :**
On pose : $u_0 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+1} = 2u_n + 3$.
On écrit une fonction récursive d'argument n qui renvoie u_n :

```
def suite(n) :
    if n==0 :
        return 1
    else :
        return 2*suite(n-1)+3
```

La structure de cette fonction est plus proche de la définition de la suite que sa version itérative.

Pour comprendre son fonctionnement, introduisons un balisage :

```
def suite(n) :
    if n==0 :
        print('cas de base : n=0')
        return 1
    else :
        print('appel récursif, n=',n)
        return 2*suite(n-1)+3

>>> suite(5)
appel récursif, n= 5
appel récursif, n= 4
appel récursif, n= 3
appel récursif, n= 2
appel récursif, n= 1
cas de base : n=0
125
```

L'interprétation est la suivante :

- $n = 5$ donc $n \neq 0$, la fonction affiche `appel récursif, n= 5` et doit renvoyer `2*suite(4)+3`, il faut donc calculer `suite(4)`,
- $n = 4$ donc $n \neq 0$, la fonction affiche `appel récursif, n= 4` et doit renvoyer `2*suite(3)+3`, il faut donc calculer `suite(3)`,
- ...
- $n = 0$ la fonction affiche `cas de base : n=0` et doit renvoyer `1`, il n'y a plus de calculs à faire.

➪ **Exemple 2 :**
Etudions un autre exemple :

```
def afficher(debut,fin) :
    if debut<=fin :
        print('ici',debut)
        afficher(debut+1,fin)
        print('la',debut)

>>> afficher(2,6)
ici 2
ici 3
ici 4
ici 5
ici 6
la 6
la 5
la 4
la 3
la 2
```

L'interprétation est la suivante :

- $debut = 2$ et $fin = 6$ donc $debut \leq fin$, la fonction affiche ici 2 et doit renvoyer afficher(3,6),
- $debut = 3$ et $fin = 6$ donc $debut \leq fin$, la fonction affiche ici 3 et doit renvoyer afficher(4,6),
- ...
- $debut = 6$ et $fin = 6$ donc $debut \leq fin$, la fonction affiche ici 6 et doit renvoyer afficher(7,6),
- $debut = 7$ et $fin = 6$ donc $debut > fin$, la boucle est terminée, il faut donc dépiler les exécutions,
- on termine afficher(6,6), la fonction affiche la 6,
- ...
- on termine afficher(3,6), la fonction affiche la 3,
- on termine afficher(2,6), la fonction affiche la 2.

1.2 Pile d'appels récursifs

Lors de l'utilisation d'une fonction récursive, on utilise une pile afin de stocker les appels précédents. La taille de la pile est limitée à 1000 par défaut, il faut donc veiller à ne pas dépasser cette taille.

➪ **Exemple 3 :**
Ecrivons une version récursive de la factorielle :

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

>>> f(6)
720
>>> f(1000)
...
RecursionError : maximum recursion depth exceeded in comparison
```

II Exemple : la suite de Fibonacci

On considère la suite de Fibonacci définie par :

$$u_0 = 1,\ u_1 = 1,\ \forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n.$$

On va écrire des fonctions d'argument n renvoyant u_n.

- Version itérative :

```
def FibonacciIt(n) :
    u=1
    v=1
    for i in range(n) :
        u,v=v,u+v
    return u
```

- Version récursive :

```
def FibonacciRec(n) :
    if n>=2 :
        return FibonacciRec(n-1)+FibonacciRec(n-2)
    else :
        return 1
```

Comparons les temps d'exécution de ces deux fonctions :

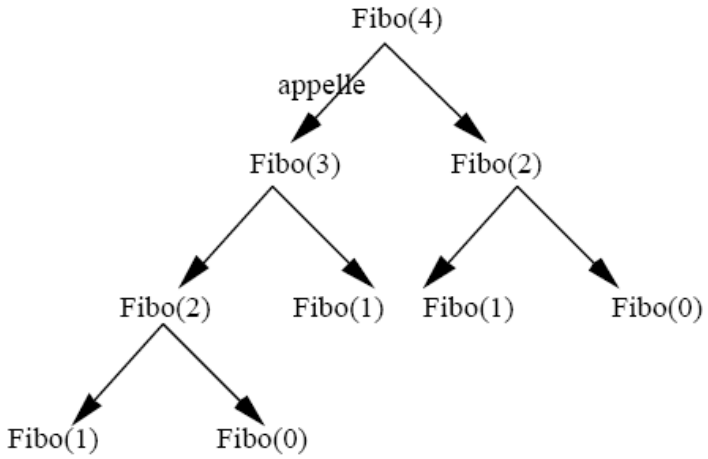
```
from time import time

debut = time()
FibonacciIt(40)
duree=time()-debut
print(int(duree), "secondes pour FibonacciIt(40)")

debut = time()
FibonacciRec(40)
duree=time()-debut
print(int(duree), "secondes pour FibonacciRec(40)")

0 secondes pour FibonacciIt(40)
37 secondes pour FibonacciRec(40)
```

Il faut 37s pour calculer u_{40} en version récursive, ce qui n'est absolument pas efficace!
Le problème est le nombre d'appels récursifs qui est trop élevé.Le nombre d'appel peut être représenté par un arbre :



Il faut donc se méfier de la récursivité car elle peut cacher un coût important.

Dans le cas particulier de la suite de Fibonacci, on peut écrire une fonction récursive efficace en utilisant un stockage mémoire des valeurs précédentes :

```
def FibonacciMemoire(n,u,v) :
    if n==0 :
        return u
    else :
        return FibonacciMemoire(n-1,v,u+v)

debut = time()
FibonacciMemoire(40,1,1)
duree=time()-debut
print(int(duree), "secondes pour FibonacciMemoire(40)")
0 secondes pour FibonacciMemoire(40)
```

Les arguments u et v représentent deux termes consécutifs de la suite de Fibonacci.