IF3 – Fonctions récursives

En programmation et en mathématiques, de nombreux problèmes se décrivent naturellement de manière récursive, c'est-à-dire en se ramenant à des sous-problèmes de même nature mais de taille plus petite. La récursivité permet de définir une fonction en fonction d'elle-même. Les fonctions récursives apparaissent dans de nombreux domaines : calculs arithmétiques (factorielle, puissances), structures de données (listes, arbres), algorithmes de tri (tri fusion, tri rapide), ou encore résolution de problèmes combinatoires (suites, tours de Hanoï…).

Objectifs :

- Comprendre la notion de récursivité :
  - Comprendre le principe d'une fonction récursive
  - Reconnaître un programme récursif et le distinguer d'un programme itératif ;
  - Savoir écrire une fonction récursive (principe du test d'arrêt) ;
  - Parvenir à formuler une solution récursive à un problème.

I – Définition et principe général

I-1) Exemple

```
In [5]: def f():
            f()
```

Lorsqu'on exécute la fonction f, on relance f qui relance f…Cette fonction ne s'arrête jamais donc jamais.
Il convient lors de l'écriture d'une fonction récursive de faire en sorte qu'on puisse sortir de cette boucle infinie. C'est le rôle du « cas de base » ou « test d'arrêt ».

Le cas de base (ou test d'arrêt) est une condition qui permet de stopper la récursion.

I-2) Principe

La récursivité est la base de la stratégie dite « diviser pour régner » :

- Traiter les cas de base ;
- Diviser : le problème à traiter est divisé en sous problèmes plus simples ;
- Régner : on applique récursivement l'algorithme à chaque sous problème ;
- Combiner : on trouve la solution du problème initial en combinant les différents résultats intermédiaires.

On peut ainsi faire le lien avec la récurrence en mathématiques :

- Calcul de f(0) : le cas de base.
- Exprimer f(n) en fonction de f(n-1) : régner.

Une fonction récursive doit respecter les trois critères suivants :

- Contenir un cas de base ;
- Modifier son état pour pouvoir se ramener au cas de base ;
- S'appeler elle-même.

I-3) Exemple de la factorielle

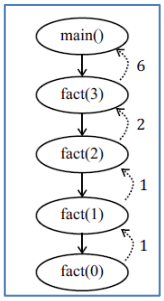
La fonction factorielle peut-être définie ainsi afin de préparer notre algorithme récursif :

$$n! = \begin{cases} 1 \; si \; n = 0 \\ n \times (n-1)! \;\; sinon \end{cases}$$

```
In [2]: def fact(n):
            if n==0:
                return 1 #Le cas de base
            else:
                return n*fact(n-1)

        fact(3)
Out[2]: 6
```

L'arbre ci-contre représente les différents appels de la fonction fact(3).
Les différents appels de la fonction récursive sont stockés dans une pile : c'est la phase de descente. Quand on atteint la condition d'arrêt, on passe à la phase de remontée et les appels sont désempilés jusqu'à retourner à l'appel initial. Au dernier appel de la fonction récursive, $n = 0$. La condition d'arrêt est vérifiée. On passe à la phase de remontée.
Pour calculer la factorielle de n, on applique cet algorithme a un sous-problème (ici factorielle de n–1). Cette méthode de décomposition/recomposition est appelée « diviser pour régner ».



II – Les différents types de récursion

II-1) Récursion simple

```
def puissance(x,n):
    if n==0:
        return 1 #Le cas de base
    else:
        return x*puissance(x,n-1) #Régner/Diviser/Combiner

puissance(2,4)
```

```
16
```

Les propriétés d'une récursion simple :

- Un seul cas de base
- Un seul appel récursif

II-2) Récursion multiple

a) Cas de base multiples

La définition de la fonction puissance(x,n) n'est pas unique. On peut par exemple identifier deux cas de base « faciles », celui pour n = 0 mais également celui pour n = 1.

```
def puissance(x,n):
    if n==0:
        return 1 #Le cas de base
    elif n==1:
        return x #Le second cas de base
    else:
        return x*puissance(x,n-1) #Régner/Diviser/Combiner

puissance(2,3)
```

```
8
```

b) Cas récursif multiples

Il est également possible de définir une fonction avec plusieurs cas récursifs. Par exemple, on peut donner une autre définition pour puissance(x,n) en distinguant deux cas récursifs selon la parité de n.

- Si n est pair, on a alors : $x^n = \left(x^{\frac{n}{2}}\right)^2$
- Si n est impair, on a alors $x^n = x * \left(x^{\frac{n-1}{2}}\right)^2$.

```
def puissance2(x,n):
    if n == 0:
        return 1
    elif n%2== 0:  # n pair
        return puissance2(x,n/2)**2 #1er appel récursif
    else:  # n impair
        return x*puissance2(x,(n-1)/2)**2 #2eme appel récursif
puissance2(3,3)
```

```
27
```

c) Double récursion

Les expressions qui définissent une fonction peuvent aussi dépendre de plusieurs appels à la fonction en cours de définition. Par exemple, la fonction fibonacci(n), qui doit son nom au mathématicien Leonardo Fibonacci, est définie récursivement, pour tout entier naturel n, de la manière suivante :

$$fibonacci(n)\begin{cases} 0\ si\ n=0 \\ 1\ si\ n=1 \\ fibonacci(n-2)+fibonacci(n-1)\ si\ n>1 \end{cases}$$

```
def fibonacci(n):
    if n == 0:
        return 0 #Cas de base 1
    elif n == 1:
        return 1 #Cas de base 2
    else:
        return fibonacci(n - 1) + fibonacci(n - 2) #Double appel récursif

fibonacci(6)
```

```
8
```

III – Efficacité de la récursivité

III-1) Mémoïsation

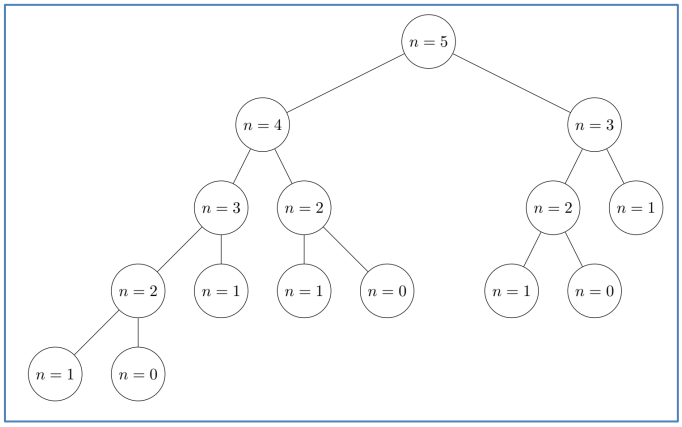
L'intérêt d'une programmation récursive est la simplicité de la mise en œuvre lorsque qu'une récurrence simple apparaît dans le problème à modéliser et la facilité avec laquelle on peut conduire les preuves de programmes. Mais une relation simple, programmée telle quelle, peut conduire à une explosion combinatoire du nombre des appels récursifs.

```
def fibonacci(n):
    global c;c=c+1;print("Le nombre d'appel de la fonction est pour n=",n," ",c)
    #On crée une variable globale pour compter le nombre d'appels de fibonacci...
    if n == 0:
        return 0 #Cas de base 1
    elif n == 1:
        return 1 #Cas de base 2
    else:
        return fibonacci(n - 1) + fibonacci(n - 2) #Double appel récursif
c=0;fibonacci(5)
```

```
Le nombre d'appel de la fonction est pour n= 5   1
Le nombre d'appel de la fonction est pour n= 4   2
Le nombre d'appel de la fonction est pour n= 3   3
Le nombre d'appel de la fonction est pour n= 2   4
Le nombre d'appel de la fonction est pour n= 1   5
Le nombre d'appel de la fonction est pour n= 0   6
Le nombre d'appel de la fonction est pour n= 1   7
Le nombre d'appel de la fonction est pour n= 2   8
Le nombre d'appel de la fonction est pour n= 1   9
Le nombre d'appel de la fonction est pour n= 0   10
Le nombre d'appel de la fonction est pour n= 3   11
Le nombre d'appel de la fonction est pour n= 2   12
Le nombre d'appel de la fonction est pour n= 1   13
Le nombre d'appel de la fonction est pour n= 0   14
Le nombre d'appel de la fonction est pour n= 1   15

5
```

Dans notre programme la fonction Fibonacci est appelé 15 fois…Pour Fibonacci(10) la fonction est appelée 177 fois. Voici l'arbre qui présente le suivi des appels récursifs.



On peut toutefois réécrire la fonction pour la rendre plus efficace et c'est ce que nous allons faire en stockant les Fibonacci(n) déjà calculées dans un dictionnaire.
Ainsi le nombre d'étapes passe pour Fibonacci(10) de 177 à 19. Donc un fort gain de temps de calcul.

```
memo={}
def fibomemo(n):
    global c;c=c+1;print("Le nombre d'appel de la fonction est pour n=",n," ",c)
    if n in memo.keys() : #On teste si fibonacci(n) a déjà été calculé
        return memo[n]
    else : #si non on calcule le nouvel élément
        if n == 0 or n == 1:
            memo[n] =  1
        else :
            memo[n] = fibomemo(n-1) + fibomemo(n-2); #Que l'on stocke dans un dictionnaire
    return memo[n]
c=0;fibomemo(5)
```

```
Le nombre d'appel de la fonction est pour n= 5   1
Le nombre d'appel de la fonction est pour n= 4   2
Le nombre d'appel de la fonction est pour n= 3   3
Le nombre d'appel de la fonction est pour n= 2   4
Le nombre d'appel de la fonction est pour n= 1   5
Le nombre d'appel de la fonction est pour n= 0   6
Le nombre d'appel de la fonction est pour n= 1   7
Le nombre d'appel de la fonction est pour n= 2   8
Le nombre d'appel de la fonction est pour n= 3   9

8
```

Cette technique de stockage est appelé « mémoïsation ».

III-2) Avantages et défauts de la récursivité

Les fonctions récursives ne sont pas toujours intéressantes et notamment ne font pas forcément gagner de temps de calcul puisqu'elles stockent au fur et à mesure les différentes valeurs ce qui est très gourmand en mémoire.

Lorsque ça s'y prête, récurrence assez évidente notamment, on écrit des fonctions récursives mais dans d'autres cas on préfèrera un algorithme itératif.

Les principaux avantages de la récursivité sont que la construction de l'algorithme est en général assez simple et que la preuve et la correction d'un programme récursif sont plus simples à mette en œuvre que pour un code itératif.