1 Exemple introductif : la somme des n premiers entiers

On connaît bien le problème consistant, étant donné un entier positif n, à calculer la somme des n premiers entiers. En général, on y répond avec une boucle for :

```python
def somme(n):
    s = 0
    for i in range(n):
        s=s+i+1
    return s
```

Nous allons voir une façon de résoudre différemment ce problème, plus proche de la définition mathématique de la somme.

La définition mathématique de la somme des n premiers entiers est la suivante : il s'agît d'une suite S_n définie par récurrence par :

$$\begin{cases} S_0 & = & 0 \\ S_n & = & n + S_{n-1} \end{cases}$$

Cela se voit en détaillant les étapes de calcul :

— $S_0 = 0$
— $S_1 = 1 + S_0 = 1 + 0 = 1$
— $S_2 = 2 + S_1 = 2 + 1 = 3$
— $S_3 = 3 + S_2 = 3 + 3 = 6$
— ...

Cette définition mathématiques ne fait pas appel à un accumulateur tel que celui utilisé dans la fonction précédente.

En fait, on peut se passer d'un accumulateur en python en écrivant directement une définition, dite récursive, de la fonction somme :

```python
def somme(n):
    if n == 0:
        return 0
    else:
        return n+somme(n-1)
```

Analysons ce qu'il se passe lors de l'appel de cette fonction sur l'entier 3 :

1. `somme(3)` : $3 \neq 0$, on calcule `3 + somme(2)`
2. `somme(2)` : $2 \neq 0$, on calcule `2 + somme(1)`
3. `somme(1)` : $1 \neq 0$, on calcule `1 + somme(0)`
4. `somme(0)` : $0 = 0$, on renvoie 0
5. le calcul de l'étape 3 peut être fait : `0 + 1 = 1`
6. le calcul de l'étape 2 peut être fait : `2 + 1 = 3`
7. le calcul de l'étape 1 peut être fait : `3 + 3 = 3`

On peut aussi représenter cette exécution par un arbre d'appels :

```
somme(3) = 3 + somme(2)
                 |
               somme(2) = 2 + somme(1)
                                |
                              somme(1) = 1 + somme(0)
                                               |
                                             somme(0) = 0
```

On appelle l'appel de `somme(n-1)` dans la définition de `somme(n)` un appel récursif.

2 Formulations récursives

Un formulation récursive est toujours constituée de plusieurs cas :

— un ou des cas de base : des cas pour lesquels le résultat est facile à calculer ;
— un ou des cas récursifs : des cas qui nécessitent des appels récursifs.

Voyons un deuxième exemple : le calcul de $x^n = x \times ... \times x$. Une façon de définir récursivement x^n est la suivante :

$$\left\{ \begin{array}{rcl} x^0 & = & 1 \\ x_n & = & x.x^{n-1} \end{array} \right.$$

Ici le cas de base est la cas $n = 0$ puisque quelque soit x, on renvoie 1. En revanche le cas $x^n = x.x^{n-1}$ fait appel au calcul de puissance x^{n-1} déjà effectué. On peut écrire en Python :

```python
def puissance(x,n):
    if n == 0:
        return 1
    else:
        return x*puissance(x,n-1)
```

On peut ajouter des cas de bases. Par exemple, dans la définition précédente, l'appel de `puissance(x,1)` va effectuer l'opération inutile $x \times 1$. On peut modifier notre code pour éviter cela :

```python
def puissance(x,n):
    if n == 0:
        return 1
    elif n==1:
        return x
    else:
        return x*puissance(x,n-1)
```

On pourrait aussi rajouter les cas de base $n = 2$ pour renvoyer x^2, $n = 3$ pour renvoyer x^3 mais cela ne nous fera pas économiser de calculs et ne sert donc à rien ici.

En remarquant que $x^{2k} = (x^k)^2$ et que $x^{2k+1} = x.(x^k)^2$ on peut obtenir la méthode dite d'exponentiation rapide, constituée récursivement de plusieurs cas récursifs :

```python
def puissance(x,n):
    if n == 0:
        return 1
    else:
        p = puissance(x, n//2)
        if n%2==0:
            return x*p*p
        else:
            return p*p
```

Cette définition permet un gain énorme de complexité car on passe d'une complexité linéaire à une complexité logarithmique.

3 Programmer avec des fonctions récursives

On peut également définir plusieurs fonction faisant appel l'une à l'autre récursivement. Par exemple :

```python
def a(n):
    if n == 0:
        return 1
    else:
        return 2*b(n-1)
def b(n):
    if n == 0:
        return 1
    else:
        return 3*a(n-1)-7
```

On voir qu'ici, la fonction `a` fait appel à la fonction `b` faisant elle même appel à la fonction `a`.

Pour concevoir une fonction récursive, il est important d'appliquer plusieurs principes :

1. avoir des cas de base connus et corrects ;
2. supposer qu'un appel récursif donne le bon résultat ;
3. s'assurer que, lors des appels successifs, on finira toujours par tomber sur un cas de base.

Par exemple, dans le cas de la première définition de la fonction `puissance(x,n)` :

1. le cas de base est correct $x^0 = 1$;

2. pour que le résultat de `x*puissance(x,n-1)` soit correct, on suppose que `puissance(x,n-1)` renvoie bien x^{n-1} ;
3. on est assuré de tomber sur le cas de base puisque chaque appel baisse de un la valeur de n.

Le point 3 est particulièrement important, notamment pour être sûr de ne pas générer d'arbre d'appels infini. Voyons un exemple de fonction mal conçue :

```
def infini(n):
    if n==0:
        return 0
    else:
        return n+infini(n+1)
```

Exécutons cette fonction sur l'entrée 1 :

```
infini(1) = 1 + infini(2)
                  |
                infini(2) = 2 + infini(3)
                                  |
                                infini(3) = 3 + infini(4)
                                                  |
                                                 ...
```

Puisque la valeur de n augmente toujours et que le cas de base est $n = 0$, cette fonction s'appelle elle même à l'infini.

Reprenons notre fonction `somme(n)` :

```
def somme(n):
    if n == 0:
        return 0
    else:
        return n+somme(n-1)
```

Cette fonction a pour vocation d'être appelée uniquement sur des entiers positifs. Mais rien n'interdit d'appeler `somme(-2)` ou même `somme(2.5)` et ces appels vont générer des arbres d'appels infinis.

1. le premier sur $n = -2, -3, -4$...
2. le second sur $n = 2.5, 1.5, 0.5, -0.5, -1.5$...

Pour éviter cela, on a l'habitude en Python de forcer le type de la variable et / ou d'utiliser la syntaxe `assert`. On peut donc réécrire :

```
def somme(n:int):
    assert n>=0, "n doit etre positif"
    if n == 0:
        return 0
    else:
        return n+somme(n-1)
```

L'inconvénient de cette solution est qu'un test de type et de positivité vont être effectués à chaque appel récursif alors que cela n'est pas nécessaire. En effet si n est un entier strictement positif (cas d'appel récursif) alors $n - 1$ aussi. Pour résoudre ce problème, une solution consiste à décomposer notre fonction en deux : une fonction auxiliaire récursive faisant les calculs sans test et une fonction principale, testant le type et le domaine puis appelant la fonction auxiliaire :

```
def somme_aux(n):
    if n == 0:
        return 0
    else:
        return n+somme_aux(n-1)

def somme(n:int):
    assert n>=0, "n doit etre positif"
    return somme_aux(n)
```

Lorsqu'on exécute un programme, la mémoire utilisée est organisée sous forme de pile où sont stockés les contextes d'exécution de chaque appel de fonctions (variables, constantes, emplacement mémoire, etc.). À chaque nouvel appel de fonction, un nouvel étage est ajouté à cette pile et cet étage est supprimé une fois le résultat retourné.

Un des inconvénients des fonctions récursives est qu'elle génèrent un nouvel étage à la pile d'exécution pour chaque appel récursif ce qui peut rapidement saturer la mémoire allouée au programme.

En l'occurence, Python limite le nombre d'appels récursifs d'une fonction à 1000 et affiche en cas de dépassement l'erreur :

```
ReccursionError: maximum recursion depth exceeded.
```

On peut cependant modifier cette limite en exécutant le code suivant :

```python
import sys
sys.setrecursionlimit(2000)
```