Récursivité - Bilan

La définition d'une fonction récursive est plutôt simple : c'est une fonction qui s'appelle elle-même. Prenons un exemple classique : le calcul de la factorielle. On définit, pour $n \geqslant 0$, $n! = \prod_{i=1}^{n} i$. Informatiquement, on peut donc définir la factorielle comme :

```
def fact(n):
   assert n>=0, "n doit etre positif"
   f=1
   for i in range(1,n+1):
      f=f*i
   return f
```

Une autre définition mathématique classique de la factorielle se fait par récurrence :

$$\begin{cases} 1 \text{ si } n = 0 \\ n! = n \cdot (n-1)! \text{ sinon.} \end{cases}$$

En reprenant quasiment mot pour mot cette dernière définition, on obtient la fonction Python suivante

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

Pile d'execution

En informatique, la pile d'exécution (ou pile d'appels, call stack en anglais) est une structure de données de type pile, qui sert à enregistrer des informations au sujet des fonctions actives dans un programme. Une fonction active est une fonction dont l'exécution n'est pas encore terminée.
En particulier, lors d'appels imbriqués c'est à dire lorsqu'une fonction f appelle une fonction g, ce qui est relatif à l'appel de la fonction g est placé juste au dessus de ce qui est relatif à la fonction f. Lorsque g termine son exécution, ce qui est relatif à l'exécution de g est dépilé. Comme l'adresse de retour est contenu dans la pile d'appel, l'exécution de f peut reprendre juste après l'endroit où g a été appelée.

Autres exemples

Algorithme d'Euclide

Soient a et b deux entiers naturels, avec $a > 0$. Si $b \neq 0$, on a la relation $PGCD(a, b) = PGCD(b, r)$, où r est le reste dans la division euclidienne de a par b. L'algorithme usuel de calcul du PGCD consiste donc à faire des divisions euclidiennes :

```
def PGCD(a,b):
    while b>0:
        a,b=b,a%b
    return a
```

Calcul de puissances

Calcul de puissances. On a vu en première année deux méthodes pour calculer x^n, un algorithme d'exponentiation naïf (faisant usage d'une boucle for, calculant toutes les puissances de x entre 1 et n), et un algorithme d'exponentiation rapide (basé sur la décomposition en binaire de n). Les deux admettent des équivalents récursifs :

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

def expo_rapide(x,n):
    if n==0:
        return 1
    else:
        y=expo_rapide(x,n//2)
        if n%2==0:
            return y*y
        else:
return y*y*x
```

Limites de la récursivité

L'usage de la récursivité présente deux inconvénients : il faut faire attention à ce que les appels récursifs ne se chevauchent pas, et prendre garde à ne pas faire un trop grand nombre d'appels récursifs imbriqués.
Considérons l'exemple de la suite de Fibonacci : soit $(Fn)_n \in \mathbb{N}$ la suite définie par 1 si $n = 0$ ou 1 et $F_n = F_{n-2} + F_{n-1}$ sinon
Une transcription récursive en Python s'obtient aisément :

```
def fib_rec(n):
   assert n>=0
   if n==0 or n==1:
      return 1
   return fib_rec(n-1)+fib_rec(n-2)
```

Le problème de l'algorithme précédent est le nombre d'appels récursifs effectués. Notons A_n le nombre d'appels récursifs nécessaires pour le calcul de F_n. Alors, la suite (A_n) vérifie la relation de récurrence $A_0 = A_1 = 0$ et pour tout $n \geqslant 2$, $A_n = 2 + A_{n-1} + A_{n-2}$, soit $A_{n+2} = (A_{n-2} + 2) + (A_{n-1} + 2)$. Autrement dit, la suite $(A_{n+2})_{n\in\mathbb{N}}$ coïncide avec la suite $(2F_n)_{n\in N}$.
On peut montrer que $A_n \sim C \cdot \left(\frac{\sqrt{5}+1}{2}\right)^n$...
ll vaut mieux utiliser une méthode itérative qui s'exécute en temps linéaire dans ce cas :

```
def fib(n):
    a,b=1,1
    for i in range(n-1):
        a,b=b,a+b
return b
```

Mais on peut aussi écrire matriciellement que

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix}$$

Poser $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ et en remarquant que

$$\begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = A^n \begin{pmatrix} F_0 \\ F_1 \end{pmatrix}$$

calculer A^n par exponentiation rapide : le calcul de F_n se fait alors en temps logarithmique !

Terminaison et correction

Montrer la terminaison d'une fonction récursive signifie démontrer que le processus récursif s'arrête pour tous paramètres. Démontrer sa correction signifie prouver que la fonction a bien le comportement attendu. Donner sa complexité signifie estimer son coût en ressources (temporelles et spatiales).
Pour montrer la terminaison d'une fonction récursive, il suffit d'exhiber une quantité, dépendant des paramètres de la fonction, à valeurs dans $\mathbb{N}$, dont les valeurs décroissent strictement au cours des appels récursifs successifs. Pour la fonction factorielle, il suffit de prendre le paramètre n lui-même. Considérons un exemple où la fonction en question est (un peu) moins évidente : la recherche dichotomique dans une liste triée, qui est explicitement au programme (au moins dans sa version itérative).

```
def DichoRec(L,x):
     """L liste triée dans l'ordre croissant, x un élément. On renvoie True si x est dans L, False sinon"""
     n=len(L)
     if n==0:
         return False
     m=n//2
     if L[m]==x:
         return(True)
     elif L[m]<x:
         return DichoRec(L[m+1:],x) #la partie à droite de L[m].
     else:
         return DichoRec(L[:m],x) #la partie à gauche.
```

La fonction $DichoRec(L, x)$ retourne un booléen caractérisant le fait que x est dans L ou non. L'idée de la recherche dichotomique est simple :

- Si la liste est vide, x n'y est pas ;
- Sinon, on regarde l'élément situé au milieu de la liste (d'indice $n/2$ avec n la taille de la liste). Si c'est x, on a terminé, sinon le fait que la liste soit triée nous permet de chercher x uniquement dans la partie droite (éléments d'indices au moins $m+1$, si $x > L[m]$), ou gauche (éléments d'indices au plus $m-1$, si $x < L[m]$).

La terminaison de la fonction *DichoRec* est alors facile à montrer : une quantité à valeurs dans N, dépendant des paramètres de la fonction, qui décroît strictement à chaque appel récursif est la longueur de la liste L. Ainsi, la fonction termine.
Il n'est pas toujours évident d'exhiber une quantité qui décroît dans une fonction récursive. Le faire pour la fonction de Syracuse suivante permettrait de résoudre un problème ouvert.

```
def Syracuse(n):
    assert n>=1
    print(n)
    if n==1:
        return(1)
    elif n%2==0:
        return Syracuse(n//2)
    else:
        return Syracuse(3*n+1)
```

Correction

Pour prouver la correction d'une fonction récursive, on procède en général par récurrence : on prouve d'abord que la sortie de la fonction est correcte pour les cas terminaux, ce qui correspond à l'initialisation de la récurrence, et on montre ensuite que les appels récursifs se ramènent à des instances plus petites (dans le même sens que la terminaison), ce qui constitue l'hérédité. La plupart du temps, la correction est facile à prouver. Par exemple pour la fonction *DichoRec* définie plus haut, on considère la proposition suivante :
Si L est une liste triée dans l'ordre croissant et x un élément comparable à ceux de L, alors $DichoRec(L, x)$ retourne True si et seulement si x est dans L, False sinon.

• Initialisation : si la longueur du tableau est 1, alors la sortie de la fonction est correcte.
• Hérédité : si L est de longueur au moins 2, un et un seul des cas suivants se produit :

- L[m] est égal à x, auquel cas la sortie de la fonction est correcte.
- L[m] est inférieur strictement à x, auquel cas x ne peut se trouver qu'après m puisque L est trié dans l'ordre croissant. Le tableau L[m :] correspond aux éléments placés après m (inclus), triés également dans l'ordre croissant. Par hypothèse de récurrence, la sortie de la fonction *DichoRec* sur l'instance (L[m :],x) est correcte, donc également sur (L,x).
- On procède de même,si L[m] est strictement supérieur à x avec le tableau L[:m].

Par principe de récurrence, la fonction *DichoRec* est correcte. On montre de même la terminaison et la correction des fonctions récursives introduites plus haut dans le chapitre.

Conclusion

On a vu dans ce cours les avantages et les inconvénients de la récursivité.
Inconvénients :

- Il faut faire attention à ne pas faire trop d'appels récursifs imbriqués.
- Il faut faire attention à ne pas faire plusieurs fois les mêmes calculs, sous peine de voir la complexité exploser.
- Lorsque deux solutions sont équivalentes, l'une en récursif, l'autre en itératif, il est en général préférable d'utiliser la version itérative.

Avantages :

- Dans l'ensemble, il est plus facile de prouver un algorithme récursif que son équivalent itératif.
- L'écriture d'un programme récursif est souvent plus claire (plus mathématique), que son équivalent itératif.
- Parfois, et c'est là où la récursivité est vraiment importante, il n'est pas du tout aisé de transformer un algorithme récursif en algorithme itératif. C'est en général le cas des algorithmes diviser pour régner .