Fonctions récursives

En mathématique, certaines propriétés se prouvent en considérant une propriété plus forte que celle qui est demandée, mais qui, elle, peut se pouver par récurrence. De même, en informatique, certaines fonctions doivent être *généralisées* afin de pouvoir les programmer de façon récursive.

Par ailleurs, il faut aussi souvent modifier le schéma récursif initial afin que l'algorithme récursif soit efficace.

Exemple 1) : Algorithme récursif de recherche par dichotomie dans un tableau trié

Le principe est de comparer l'élément cherché x avec le médian du tableau trié a. On se ramène alors en $O(1)$ comparaisons à la recherche de x dans une moitié de tableau.

Mais il est impératif de ne pas avoir à construire explicitement cette moitié de tableau. En effet, le coût de $a[0:m]$ est de coût $O(m)$, car les éléments de a sont alors recopiés.

L'idée est donc de généraliser la fonction et de définir par récurrence une fonction qui cherche l'élément x dans le sous-tableau $a[i:j]$ codé par le tableau a et les indices i et j.

Autrement dit, on définit la fonction récursive auxilaire `aux(x,a,i,j)` :

```
def aux(x,a,i,j) :        # renvoie True ssi x appartient au sous-tableau a[i : j]
    if i = j :  return False          # test d'arrêt associé à un tableau vide
    k = (i+j)//2              # on a toujours i ≤ k < j
    if x == a[k] :  return True
    elif x<a[k] :  return aux(x,a,i,k)     # on a bien k − i < j − i
    else :  return aux(x,a,k+1,j)     # on a bien j − (k + 1) < j − i
```

La fonction demandée est alors :

```
def recherche(x,a) :  return aux(x,a,0,len(a))
```

Exemple 2) : Suite de Fibonacci

Il s'agit d'écrire une fonction `fibo(n)` qui renvoie le n-ième terme f_n de la suite de Fibonacci.

La fonction récursive suivante est de complexité exponentielle :

```
def fibo(n) :
    if n == 0 or n==1 :  return 1
    return fibo(n-1)+fibo(n-2)
```

En effet, le coût $c(n)$ vérifie $c(n) = 1 + c(n-1) + c(n-2)$.

Avec $d(n) = 1 + c(n)$, on a $d(n) = d(n-1) + d(n-2)$, donc $d(n) \sim \alpha\varphi^n$, où $\varphi = 1.61...$

Selon le principe de mémoïsation (consistant à stocker des valeurs pour éviter de les calculer plusieurs fois), on se ramène à un schéma récursif d'ordre 1 en le couple (f_n, f_{n+1}).

```
def fibo2(n) :
```

```
    if n == 0 :  return (1,1)
    x,y = fibo2(n-1) ; return (y,x+y)
def fibo(n) :  return fibo2(n)[0]
```

Exemple 3) : Parcours en profondeur issu d'un sommet dans un graphe

On considère un graphe $G = (S, A)$ codé la liste des listes d'adjacence et un sommet $s \in S$.

On souhaite renvoyer la liste des sommets obtenus par un parcours en profondeur issu de s.

On peut utiliser une pile :

```
def parcours(G,s) :
    n = len(G) ; L = [] ; M = [0]*n ; pile = [s] ; M[s] = 1
    # M est un tableau de marqauge des sommets visités ; la pile donne les sommets à traiter
    while pile :
        x = pile.pop() ;L.append(x)
        for y in G[x] :
            if M[y] == 0 :
                pile.append(y) ; M[y] = 1 ; M[x] = 1
    return L
```

Mais on peut aussi utiliser **une fonction auxiliaire récursive** qui étant donné un sommet x et la liste déjà calculée va ajouter à cette liste la liste des sommets (non encore visités) du parcours issu de x.

```
def parcours(G,s) :
    n = len(G) ; M = [0]*n
    def traite(x) :       # on pourrait aussi écrire : traite(x,G,L,M)
        M[x] = 1 ; L.append(x)
        for y in G[x] :
            if M[y] == 0 :  traite(y)
    L = [] ; traite(s) ; return L
```

Remarque : On peut définir `traite` en dehors du corps de parcours, à condition de mettre en paramètres impérativement tous les arguments utilisés :

```
def traite(x,G,L,M) :
    M[x] = 1 ; L.append(x)
    for y in G[x] :
        if M[y] == 0 :  traite(y)
def parcours(G,s) :
    n = len(G) ; M = [0]*n ; L = [] ; traite(s,G,L,M) ; return(L)
```

Fonctions récursives. *Complément culturel* : Dérécursification

a) En fait, tout programme récursif est "dérécursifié" par le compilateur de PYTHON, c'est-à-dire transformé en un programme itératif. La dérécursification utilise une pile stockant les arguments des appels récursifs successifs.

Dans certains cas (comme les fonctions `fact`, `suite` et `fibo`), l'utilisation d'une pile n'est pas nécessaire :

```
def fact(n) :
    x = 0
    for i in range(1,n+1) :  x = x * i
    return x
```

b) *Etude d'un cas générique*

On considère $(x_n)_{n\in\mathbb{N}}$ définie par $x_0 = 0$ et une récurrence forte : $\forall n > 0$, $x_n = f(n, x_{d(n)})$, avec $0 \leq d(n) < n$.

Dans le cas général, la programmation itérative utilise une pile permettant de stocker les images successives k de n par d jusqu'à obtenir 0, puis à dépiler en calculant un par un les termes x_k correspondant (en commençant par x_0).

Les appels récursifs s'effectuant donc sur les valeurs $x_n \to x_{d(n)} \to x_{d(d(n))} \to x_{d(d(d(n)))} \to ... \to 0$.

Remarque : Dans le cas particulier où $d(n) = n - 1$, c'est-à-dire $x_n = f(n, x_{n-1})$, l'utilisation d'une pile n'est pas nécessaire, puisqu'on connaît d'emblée les différentes valeurs k (en l'occurrence $0 \to 1 \to 2 \to ... \to n$).

Exemple : Considérons la fonction f définie par $f(0) = 0$ et $\forall n \in \mathbb{N}^*$, $f(n) = n + f(\lfloor \frac{n}{2} \rfloor)$.

Une définition récursive est :

```
def f(n) :
    if n == 0 :  return 0
    return n + f(n//2)
```

La programmation itérative s'effectue en deux étapes :

- On commence par construire la pile $(x_0, x_1, ..., x_m)$ définie par $x_0 = n$, $x_{k+1} = \lfloor \frac{1}{2} x_k \rfloor$ jusqu'à obtenir $x_{m+1} = 0$.
- En dépilant la pile, on calcule successivement $f(0)$, $f(x_m)$, $f(x_{m-1}), ..., f(x_0)$ et on renvoie $f(x_0)$.

```
def f(n) :
    x = n
    pile = []
    while x != 0 :
        pile.append(x) ; x = x//2
    y = 0
    while pile != [] :                  # syntaxe allégée :  while pile :
        x = pile.pop() ; y = x + y
    return y
```

c) *Exemple : Algorithme d'exponentiation rapide*

On suppose connue une fonction `produit(a,b)` qui effectue le produit ab dans un ensemble E.

On connaît aussi `neutre()` la fonction renvoyant l'élément neutre de E.

Ecrire une fonction `power(a,n)` qui étant donnés un élément a et un entier n renvoie a^n selon l'algorithme d'exponentiation rapide : il s'agit de l'algorithme récursif basé sur le principe suivant : $a^n = (a^m)^2 a^r$ pour $n = 2m + r$, avec $r \in \{0, 1\}$.

Autrement dit, on utilise $a^n = (a^m)^2$ si $n = 2m$ est pair, et $a^n = (a^m)^2 a$ si $n = 2m + 1$ est impair.

La complexité est en $O(\log n)$ opérations.

```
def power(A,n) :                        # fonction récursive
     if n == 0 :  return neutre()
     r = m%2 ; m = n//2 ; b = power(A,m)
     if r == 0 :  return produit(b,b)
     else :  return produit(a,produit(b,b))
```

La version itérative consiste à stocker dans une pile la liste des restes de la division euclidienne par 2, c'est-à-dire l'écriture en base 2 de l'entier n.

```
def power(A,n) :                        # fonction itérative
     q = n ;
     while q != 0 :  L.append(q%2) ; q = q//2
     # L est la liste [r_0, ..., r_{p-1}] de sorte que n = sum_{k=0}^{p-1} r_k 2^k, avec r_k ∈ {0,1}.
     b = neutre()
     while L != [] :       # la dernière itération correspond au bit des unités
         if L.pop() == 0 :  b = produit(b,b)
         else :  b = produit(a,produit(b,b))
     return b
```