INFORMATIQUE

TD n°4

Introduction à la récursivité

Définition et exemples

Un algorithme est dit **récursif** s'il s'appelle lui-même. Sinon l'algorithme est dit **itératif**.

⚠ bien veiller à ce qu'un algorithme récursif comporte une **terminaison**, c'est-à-dire un (ou plusieurs) cas où la fonction renvoie directement une valeur sans appel à elle-même.
Exemples classiques :

- fonction qui calcule $n!$:

```
def factorielle(n):
    if n==0:
        return 1                    #0!=1  [terminaison]
    else:
        return n*factorielle(n-1)   #n!=n*(n-1)!
```

- fonction qui calcule le n-ième terme de la suite de FIBONACCI ($f_0 = f_1 = 1$ et $f_n = f_{n-1} + f_{n-2}$) :

```
def f(n):
    if n<=1:
        return 1 #[terminaison]
    else:
        return f(n-1)+f(n-2)
```

Remarque : nous verrons pas la suite que cette fonction `f` pose un problème d'exécution.

Programmation itérative ou programmation récursive ?

Tout programme peut être écrit au choix sous forme itérative ou récursive. Par exemple :

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

renvoie le `n`-ième terme de la suite définie par $u_0 = a$ et $\forall n \in \mathbb{N}^*$, $u_n = 3u_{n-1} + 1$.

Son équivalent itératif est :

```
def suite(n,a):
    u=a
    for i in range(n):
        u=3*u+1
    return u
```

On alors dit qu'on a « dérécursivé » le premier programme.

1 Exercices sur l'écriture récursive

Dans les exemples qui suivent, la récursivité n'apporte rien en terme d'efficacité d'exécution par rapport à l'écriture itérative. Le but est ici de s'entraîner à passer de l'une à l'autre de ces écritures pour constater que l'écriture du code est parfois plus simple dans un cas que dans l'autre.

Q1 Écrire une fonction récursive `reste(n,b)` (resp. `quotient(n,b)`, resp. `division(n,b)`) qui renvoie le reste (resp. le quotient, resp. la liste `[q,r]` formée du quotient et du reste) de la division euclidienne de l'entier naturel `n` par l'entier naturel non nul `b`.
On n'utilisera pas les fonctions `%`, `//` ni `/`. On utilisera le fait que si `n<b`, le reste vaut `n` et le quotient vaut `0`, et que sinon ils s'obtiennent à partir de la division euclidienne de `n-b` par `b`.
Écrire des versions itératives de ces fonctions.

Q2 Écrire une fonction récursive `inverse(L)` qui renvoie la liste obtenue en inversant l'ordre des éléments de la liste `L` ; on remarquera qu'elle s'obtient à partir du dernier élément de la liste et de la liste inverse des autres éléments. *Par exemple :* `inverse([1,2,3,5])` → `[5,3,2,1]`
Écrire une version itérative de la même fonction.

Q3 Sans utiliser la fonction `inverse`, écrire une fonction récursive `estPalindrome(L)` qui renvoie `True` si la liste `L` est un palindrome — *i.e.* `inverse(L)` renvoie `L` — et qui renvoie `False` sinon.
Par ex. : `estPalindrome([1,3,5,5,3,1])` → `True` ; `estPalindrome([2,3,5,3,1])` → `False`
Écrire une version itérative de la même fonction (qui n'utilise pas non plus `inverse`).

Q4 Écrire une fonction récursive `nbDeChiffres(n)` qui renvoie le nombre de chiffres de l'écriture décimale de l'entier non nul `n`. On remarquera que tout entier `n`$\geqslant 10$ a un chiffre de plus que `n//10` ; on n'utilisera pas de chaîne de caractère.
Par ex. : `nbDeChiffres(563)`→`3` ; `nbDeChiffres(12385)`→`5`
Écrire une version itérative de la même fonction.

Q5 Si n est un nombre dont l'écriture binaire est $n = \overline{b_{p-1}b_{p-2}\dots b_1 b_0}$ alors $n = \overline{b_{p-1}b_{p-2}\dots b_1 0} + \overline{b_0} = 2 \times \overline{b_{p-1}b_{p-2}\dots b_1} + b_0$ donc b_0 est le reste dans la division euclidienne de n par 2 et b_1 est le chiffre des unités du quotient de n par 2 (et il suffit donc de diviser ce quotient par 2 pour déterminer b_1).
Écrire une fonction récursive `binaire(n)` qui renvoie une liste correspondant à l'écriture binaire d'un entier n (elle renverra la liste vide pour $n = 0$).
Écrire une version itérative de la même fonction.

Q6 L'algorithme d'EUCLIDE de calcul du PGCD (plus grand diviseur commun) de deux entiers naturels non nuls a et b est fondé sur le fait que si a n'est pas divisible par b, alors le PGCD de a et b est égal au PGCD de b et r, où r est le reste de la division euclidienne de a par b.
Écrire une fonction récursive `pgcd(a,b)` qui calcule le PGCD de a et b.
Écrire une version itérative de la même fonction.

Q7 Préciser ce que renvoie la fonction suivante :

```
def S(n):
    S=0
    for i in range(n):
        S=S+1/(i+1)
    return S
```

Écrire une fonction récursive équivalente à cette fonction.

Q8 Écrire deux fonctions récursives `somme(L)` qui renvoient la somme des éléments de la liste `L` (l'une ramenant la somme d'une liste de longueur n à la somme d'une liste de longueur $n-1$ et l'autre procédant par dichotomie).
Écrire une version itérative de chacune de ces deux fonctions.

Q9 Écrire une fonction récursive `decomposition(n)` qui renvoie la décomposition de `n` en facteurs premiers, rangés dans l'ordre croissant. On rappelle qu'un nombre `n` est premier si et seulement s'il n'est divisible par aucun nombre `d` tel que $2 \leqslant$ `d` $\leqslant \sqrt{\texttt{n}}$.
Par exemple : `decomposition(12)` → `[2,2,3]` ; `decomposition(13)` → `[13]`
Écrire une version itérative de la même fonction.

2 Maniement d'une structure de donnée récursive : listes imbriquées

Les listes `[[1, 2], [9, 10]]`, `[[1], 2]`, `[[1], 2, [3, [[4]]]]` ou plus simplement `[1, 2, 3]` sont des listes imbriquées d'entiers. On voudrait écrire une fonction `S(L)` qui renvoie la somme des entiers contenus dans une liste imbriquée d'entiers `L`. *Par exemple :* `S([[5], 2, [10, 8, [[7]]]])` → `32`

Les listes imbriquées ont une structure naturellement récursive : une liste imbriquée est soit nombre entier, soit une liste dont chacun des éléments est un nombre entier ou une liste imbriquée.

La somme d'une liste imbriquée `L` peut aussi être définie récursivement de la manière suivante : si `L` est un entier, la somme de `L` est égale à l'entier `L` ; sinon `L` est une liste dont la somme est par définition (récursive) la somme de chaque élément (liste ou entier) de cette liste.
Par exemple pour calculer `S([[5], 2, [10, 8, [[7]]]])` :

- `S([[5], 2, [10, 8, [[7]]]])` est égal à `S([5])+2+S( [10, 8, [[7]]])` ;
- `S([5])` est égal à `5` ;
- `S([10, 8, [[7]]])` est égal à `10+8+S([[7]])` ;
- `S([[7]])` est égal à `S([7])` lui même égal à `7` ;
- on en déduit que `S([10, 8, [[7]]])` est égal à `10+8+7` et donc que `S([[5], 2, [10, 8, [[7]]]])` est égal à `5+2+10+8+7=32`.

Lorsqu'on applique récursivement cette définition de la somme, on aboutit en fin de compte à des sommes d'entiers.

Q10 Écrire une fonction `S(L)` qui renvoie la somme d'une liste imbriquée (on utilisera la fonction `isinstance` : `isinstance(x, int)` qui renvoie `True` si la variable `x` est de type entier et `False` sinon).

Q11 Écrire une fonction `contient(L,x)` d'argument une liste imbriquée `L` et un entier `x` qui renvoie `True` si `x` est élément d'une quelconque des listes imbriquées dans `L` et `False` sinon.

3 Décompte du nombre d'appels récursifs et dérécursivation

La récursivité est particulièrement intéressante lorsque la dérécursivation engendrerait l'écriture d'une fonction beaucoup plus compliquée. Cela se produit lorsque le nombre de variables stockées dans les appels récursifs est important. Dans ce cas, la fonction itérative correspondante devra définir des structures de données permettant de stocker les valeurs de toutes ces variables, ce qui est fait automatiquement pour une fonction récursive. Une version itérative est par contre préférable lorsque le nombre d'appels récursifs est très grand.

Q12 En utilisant uniquement la formule du triangle de PASCAL et le fait que $\binom{n}{0} = 1 = \binom{n}{n} = 1$, écrire une fonction récursive `binom(n,p)` qui renvoie le coefficient binomial $\binom{n}{p}$.

Combien faut-il d'appels récursifs pour calculer $\binom{5}{3}$? *(Penser à les représenter sous la forme d'un arbre).*

Q13 Écrire une fonction `binom2(n,p)` qui renvoie le coefficient binomial $\binom{n}{p}$. Cette fonction fera appel à une fonction auxiliaire récursive `Pascal(L)`, où `L` est la liste des coefficients binomiaux $[\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}]$, et qui renvoie la liste $[\binom{n+1}{0}, \binom{n+1}{1}, \ldots, \binom{n+1}{n+1}]$.

Combien faut-il d'appels récursifs pour calculer $\binom{5}{3}$? Combien de sommes ont été effectuées ?

Q14 Écrire une fonction itérative `binom3(n,p)` qui renvoie le coefficient binomial $\binom{n}{p}$.

Q15 Dans la fonction récursive `division` de la question 1, pourquoi l'écriture suivante :

```
...
    else:
        [q,r]=division([n-b,b])
        return [1+q,r]
```

est-elle de très loin préférable (indépendamment de la lisibilité du code) à :

```
...
    else:
        return [1+division([n-b,b])[0],division([n-b,b])[1]]
```

4 Pour ceux qui ont déjà fini

EXERCICE 1

On représente un damier dont les cases sont noires ou blanches par une liste de listes `L` dont les éléments valent 0 (pour noir) ou 1 (pour blanc).

Q16 Écrire une procédure `coloriage(L,i,j)` qui modifie le damier `L` en coloriant en gris (*i.e.* avec la valeur 2) la case `L[i][j]`, ainsi que toute case de la zone de blanc qui contient cette case — ces cases se touchent par un côté au moins. Cette procédure sera récursive, et agira par « contamination », en examinant toutes les cases voisines d'une case déjà grisée.
Par exemple :

```
L=[[1,0,1,1,1],
   [1,1,0,1,0],
   [1,0,0,0,1],
   [1,1,1,0,1]]
coloriage(L,1,0)
print(L)
   [[2,0,1,1,1],
    [2,2,0,1,0],
    [2,0,0,0,1],
    [2,2,2,0,1]]
```

EXERCICE 2

Un commerçant doit rendre la somme de `n` € (`n` entier naturel non nul) à un client. Pour cela, il dispose de stocks illimités de pièces de 1 €, 2 €, 5 €. Par exemple, il y a 7 manières de rendre 8 € :

$8\times 1€$	$4\times 2€$	$6\times 1€+1\times 2€$	$3\times 1€+1\times 5€$
$4\times 1€+2\times 2€$	$1\times 1€+1\times 2€+1\times 5€$	$2\times 1€+3\times 2€$	

Q17 À l'aide d'une fonction récursive auxiliaire, écrire une fonction `monnaie(n)` qui calcule le nombre de manières différentes de rendre `n` € de monnaie.

EXERCICE 3

Q18 Écrire une fonction récursive `chemins(x,y)` qui renvoie le nombre de chemins allant de l'origine, de coordonnées `0,0`, au point de coordonnées entières positives `x,y`, où les chemins considérés sont les lignes brisées reliant des points de coordonnnées entières telles que l'on passe d'un point au suivant par une translation de vecteur $(1,0)$ (vers la droite), $(0,1)$ (vers le haut) ou $(1,1)$ (en diagonale vers le haut et à droite).
Par exemple : `chemin(1,1)→ 3; chemin(1,2)→ 5`

Q19 Écrire une fonction récursive `nb(L,N)` où `L` est une liste de chiffres distincts (de 0 à 9) qui renvoie la liste de tout les nombres inférieurs ou égaux à `N` dont tous les chiffres appartiennent à `L`.
Par exemple : `nb([1,3],32)→[1,3,11,13,31]`

Q20 Écrire une fonction récursive `supprimeDoublon(L)` où `L` est une liste, et qui renvoie la liste `L` où l'on a supprimé les doublons.
Par exemple : `supprimeDoublon([1,2,5,6,6,2,4,1,5])→[1,2,5,6,4]`

Q21 Écrire une fonction `tours(n)` qui renvoie toutes les configurations possibles de placements de `n` tours, numérotées de `1` à `n`, sur un échiquier `n×n` de sorte qu'aucune ne soit en prise avec une autre (*i.e.* placée sur une même ligne ou une même colonne).
Par exemple : `tour(2)→ [[[1,0],[0,2]],[[2,0],[0,1]],[[0,2],[1,0]],[[0,1],[2,0]]]`

INFORMATIQUE

Corrigé du TD n°4

Q1

```
def reste(n,b):
    if n<b:
        return n
    else:
        return reste(n-b,b)

def quotient(n,b):
    if n<b:
        return 0
    else:
        return 1+quotient(n-b,b)

def division(n,b):
    if n<b:
        return [0,n]
    else:
        [q,r]=division([n-b,b])
        return [1+q,r]

#versions itératives

def reste(n,b):
    r=n
    while r>=b:
        r=r-b
    return r

def quotient(n,b):
    r=n
    q=0
    while r>=b:
        r=r-b
        q=q+1
    return r

def division(n,b):
    r=n
    q=0
    while r>=b:
        r=r-b
        q=q+1
    return [q,r]
```

Q2

```python
def inverse(L):
    n=len(L)
    if n<=1:
        return L[:] #renvoi d'une copie de L si elle est sa propre inverse
    else:
        return [L[n-1]]+inverse(L[0:n-1])

def inverse(L):
    LL=[]
    for i in range(len(L)):
        LL=[L[i]]+LL
    return LL
```

Q3

```python
def estPalindrome(L):
    n=len(L)
    if n<=1:
        return True
    elif L[0]!=L[n-1]:
        return False
    else:
        return estPalindrome(L[1:n-1])

def estPalindrome(L):
    n=len(L)
    for i in range(n//2):
        if L[i]!=L[n-1-i]:
            return False
    return True
```