Fonctions récursives

T.P. I

Partie I : Récursivité

Un programme *récursif* est un programme qui fait appel à lui-même. Très souvent un algorithme récursif est lié à une relation de récurrence permettant de calculer la valeur d'une fonction pour un argument n à l'aide des valeurs de cette fonction pour des arguments strictement inférieurs à n. Lors de son évaluation, le programme garde en mémoire dans une pile les opérations à effectuer jusqu'à l'appel d'un cas de base. La pile en mémoire est alors dépilée jusqu'à obtention du résultat final. Cette évaluation peut être représentée à l'aide de boîtes imbriquées.
Par exemple, pour $x \in \mathbb{R}$ et $n \in \mathbb{N}$, le réel x^n est défini par récurrence à partir des relations :

$$x^0 = 1 \text{ et } x^n = x \cdot x^{n-1} \text{ si } n > 1$$

```
def puissance(x,n) :
    """calcul recursif de x**n"""
    if n == 0 :
        return 1
    else :
        return (x * puissance(x,n-1))

>>> puissance(2,4)
16
```

```
4 == 0 : False
  Appelle puissance(2, 3)
  3 == 0 : False
    Appelle puissance(2, 2)
    2 == 0 : False
      Appelle puissance(2, 1)
      1 == 0 : False
        Appelle puissance(2, 0)
        0 == 0 : True
        Valeur retournée : 1
      Évaluation de 2 * 1
      Valeur retournée : 2
    Évaluation de 2 * 2
    Valeur retournée : 4
  Évaluation de 2 * 4
  Valeur retournée : 8
Évaluation de 2 * 8
Valeur retournée : 16
```

Lors de l'écriture d'une fonction récursive, il faut vérifier que

- les appels à la fonction s'effectuent sur des arguments qui soient **strictement plus petits** : un entier strictement inférieur, une liste de taille strictement plus petite,...
- la fonction renvoie bien une valeur pour les **cas de base**, i.e. les arguments les plus petits : l'entier 0, la liste vide,...

Pour éviter un trop grand nombre d'appels ou un nombre d'appels infinis qui provoqueraient un débordement de la pile des appels, la profondeur de la récursion (i.e. le nombre d'appels) est limité par une constante. L'accès à cette constante est possible via le module `sys` et la commande `sys.getrecursionlimit()`.

La **complexité** de la fonction s'exprime généralement à l'aide d'une formule de récurrence. Pour la fonction `puissance` précédente, en notant T_n le nombre de multiplications effectuées lors de l'évaluation `puissance(x,`n`)`, alors

$$T_0 = 0 \text{ et } T_n = 1 + T_{n-1}.$$

Ainsi, $T_n = n$ et la complexité de l'algorithme est linéaire.

La **correction** de la fonction se prouve par récurrence. Pour la fonction `puissance` :
Initialisation. `puissance(x,0)` renvoie $1 = \texttt{x}^0$.
Hérédité. Soit $n \in \mathbb{N}$. Supposons que `puissance(x,`n`)` renvoie $\texttt{x}^n$. Alors, `puissance(x,`$n+1$`)` renvoie $\texttt{x} \cdot \texttt{x}^n = \texttt{x}^{n+1}$.

Partie II : Exercices

1. La fonction factorielle est définie par la relation de récurrence :

$$0! = 1 \text{ et } \forall\, n \in \mathbb{N},\ (n+1)! = (n+1) \cdot n!$$

Écrire une fonction itérative `fact_iter` puis une fonction récursive `fact_rec` qui, étant donné un entier naturel `n`, renvoie `n`!.

2. Écrire une fonction itérative `somme_entiers_iter` puis une fonction récursive `somme_entiers_rec` qui, étant donné un entier naturel `n`, renvoie la somme des entiers naturels inférieurs ou égaux à `n`.

3. On considère la fonction définie par

```
def f(n):
  if n == 0:
    return 2
  else:
    return f(n-1) * f(n-1)
```

a) Déterminer, sans utiliser l'ordinateur, les valeurs renvoyées par les appels `f(0)`, `f(1)`, `f(2)`, `f(3)`.

b) Que calcule cette fonction ? Préciser sa complexité en nombre de multiplications.

4. Reprendre les questions précédentes avec la fonction

```
def g(n):
  if n == 0:
    return 2
  else:
    tmp = g(n-1)
    return tmp * tmp
```

5. On considère la fonction, définie sur les couples d'entiers naturels, par

```
def b(n, p):
  if p > n:
    return 0
  elif n == 0 or p == 0:
    return 1
  else:
    return (n * b(n-1,p-1)) // p
```

a) Déterminer, sans ordinateur, les valeurs renvoyées par les appels `b(2, 0)`, `b(2, 1)`, `b(2, 2)`.

b) Que calcule cette fonction ? Préciser sa complexité.

6. Soit f une fonction continue sur un intervalle $[a, b]$ de $\mathbb{R}$ telle que $f(a)f(b) \leqslant 0$. Alors, f possède un zéro c sur $[a, b]$ tel qu'on puisse construire par récurrence deux suites (a_n) et (b_n) telles que

- $a_0 = a$ et $b_0 = b$,
- $\forall\, n \in \mathbb{N},\ b_n - a_n = \frac{b-a}{2^n}$,
- $\forall\, n \in \mathbb{N},\ f(a_n)f(b_n) \leqslant 0$,
- $\forall\, n \in \mathbb{N},\ c \in [a_n, b_n]$.

a) Écrire une fonction itérative `dicho_iter` puis une fonction récursive `dicho_rec` qui prend en arguments une fonction `f`, deux réels `a`, `b` et un réel `eps`, et renvoie une approximation par **défaut** à `eps` près d'un zéro de la fonction `f`.

b) Rechercher une approximation par défaut à 10^{-2} près d'un zéro de la fonction $x \mapsto x^2 - 4$ sur l'intervalle $[0, 3]$.

c) Rechercher une approximation par défaut du réel π à 10^{-10} près.

Votre code pourra utiliser les fonctions trigonométriques mais ne devra pas faire appel à la variable `pi` *disponible dans les modules de Python.*

7. Exponentiation rapide. Soient x un réel et n un entier naturel. En notant $y = x^{\lfloor \frac{n}{2} \rfloor}$, alors

$$\begin{aligned} x^0 &= 1, \\ x^n &= y \cdot y \text{ si } n \text{ est pair}, \\ &= x \cdot y \cdot y \text{ si } n \text{ est impair}. \end{aligned}$$

Écrire une fonction `exponentiation_rapide` qui, étant donnés un réel `x` et un entier naturel `n`, renvoie la valeur de `x**n` en utilisant la relation de récurrence précédente. Cette fonction ne devra pas utiliser l'opérateur `**`. On montre dans la partie suivante que le calcul de x^n nécessite de l'ordre de $\ln(n)$ multiplications.

On veillera, pour diminuer la complexité, à n'effectuer qu'un seul appel récursif. On rappelle que `//` et `%` permettent de calculer respectivement le quotient et le reste de divisions euclidiennes.

Partie III : Des questions de complexité...

8. La suite de FIBONACCI est définie par la relation de récurrence

$$F_0 = F_1 = 1 \text{ et } \forall\, n \in \mathbb{N},\ F_{n+2} = F_{n+1} + F_n.$$

a) Écrire une fonction récursive `fibo_rec` qui, étant donné un entier `n`, renvoie l'entier $F_{\texttt{n}}$.

b) Écrire une fonction itérative `fibo_iter` qui, étant donné un entier `n`, renvoie l'entier $F_{\texttt{n}}$.

c) Calculer F_{35} à l'aide des deux algorithmes précédents. Que constatez vous ? Pourquoi ce phénomène apparaît ?

d) Évaluer la complexité $T_{\texttt{n}}$ du nombre d'additions effectuées par l'appel `fibo_rec(n)`. Donner un équivalent de $T_{\texttt{n}}$ lorsque $\texttt{n} \to +\infty$.

e) Pour tout entier naturel n, notons $U_n = \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}$. Alors, $U_{n+1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} U_n$.

En adaptant l'algorithme d'exponentiation rapide pour des matrices, proposer une fonction `fibonacci` qui renvoie la valeur de F_n.

On pourra utiliser des tableaux `numpy` ainsi que le produit matriciel `dot`.

9. On considère la fonction suivante

```
def h(P, x):
  def aux(d):
    if d == len(P):
      return 0
    else:
      return P[d] + x * aux(d+1)
  return aux(0)
```

a) Quel est le type de l'argument `P` ? Quel est le type renvoyé par l'appel à la fonction `h` ?

b) Quel est le résultat renvoyé par `h(P, x)` ? Préciser la complexité de cette fonction en nombre d'additions puis en nombre de multiplications.

Vous aurez reconnu l'algorithme de HORNER.

10. Tours de Hanoï. On dispose de trois tiges sur lesquelles s'enfilent des disques de tailles différentes. On définit la contrainte suivante : sur chaque tige, on ne peut empiler un disque que si son diamètre est plus petit que ceux des disques déjà empilés sur cette tige. Au départ, tous les disques se trouvent sur la tige n°1. Il faut empiler les disques sur la tige n°3.

Ce jeu a été inventé par É. LUCAS *en 1883. La version originale est disponible à l'adresse :*
edouardlucas.free.fr/pdf/oeuvres/Jeux_3.pdf

a) Écrire une fonction `hanoi` qui prend en argument un entier et trois chaînes de caractères. L'appel `hanoi(n, dep, inter, arr)` affiche la suite d'instructions à effectuer pour déplacer une pile de disques de taille `n` de la tige de départ `dep` à la tige d'arrivée `arr` en utilisant la tige intermédiaire `inter`.

b) Déterminer la complexité de `hanoi`.

11. Complexité de l'exponentiation rapide. On note $T_{\texttt{n}}$ le nombre de multiplications effectuées par l'appel `exponentiation_rapide(x, n)` et on décompose $\texttt{n} = (b_t \cdots b_0)_2$ en base 2.

a) Déterminer l'écriture binaire de `n//2` et de `n%2`.

b) Déterminer la valeur de `t` en fonction de $\log_2(\texttt{n})$.

c) Montrer que $T_{(b_t \cdots b_0)_2} = T_{(b_t \cdots b_1)_2} + b_0 + 1$.

d) En déduire la complexité de l'exponentiation rapide en fonction de `n`. La comparer à l'algorithme itératif classique.

12. Plus grand diviseur commun. Étant donnés deux entiers naturels a et b tels que $a \geqslant b$, et en notant r le reste de la division euclidienne de a par b, le plus grand commun diviseur entre a et b, noté $a \wedge b$ est défini par

$$a \wedge 0 = a \text{ et } a \wedge b = b \wedge r.$$

a) Écrire une fonction `pgcd` qui renvoie le plus grand commun diviseur de deux entiers naturels.

b) Pouvez vous évaluer la complexité de cette fonction ?

Ce théorème a été démontré par G. Lamé *en* 1844. *L'article original est disponible à l'adresse :*
http://gallica.bnf.fr/ark:/12148/bpt6k2978z/f867.item

Mathématiciens

Fibonacci Leonardo (1170 à Pise-1250 à Pise).
Horner William (1786 à Bristol-22 sept. 1837 à Bath).
Lamé Gabriel (22 juil. 1795 à Tours-1er mai 1870 à Paris).
Lucas François Édouard (4 avr. 1842 à Amiens-3 oct. 1891 à Paris).