Récursivité

1 Définitions, mode de fonctionnement

Définition 1.1 *Une fonction est dite* récursive *lorsque au sein de celle-ci, un appel est fait à elle-même. Une forme de récursivité est également obtenue lorsque deux fonctions (ou plus) s'appellent l'une l'autre, appelée* récursivité croisée.

Exemple 1.2 *La factorielle est définie mathématiquement par : pour tout* $n \in \mathbb{N}$, *si* $n = 0$, $n! = 1$ *et sinon* $n! = n * (n-1)!$.

Cette définition, par récurrence, conduit à la fonction récursive suivante :

```
def fact(n):
   if n == 0:
       return 1
   return n * fact(n-1)
```

C'est la notion de *pile*, et plus spécifiquement celle de *pile d'exécution* (runtime stack en anglais), qui rend possible la notion de récursivité.

Cette pile d'exécution apparaît en clair dans la console python dès lors qu'un plantage survient.

Voici un petit programme (nommé `plante.py`) qui plante :

```
def a():
    return b()

def b():
    return c()

def c():
    return d()

print(a())
```

Son exécution conduit à l'affichage suivant (un peu écourté de choses inutiles) :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "plante.py", line 17, in <module>
    print(a())
  File "plante.py", line 9, in a
    return b()
  File "plante.py", line 12, in b
    return c()
  File "plante.py", line 15, in c
    return d()
NameError: name 'd' is not defined
```

Le plus important reste la dernière ligne qui indique une erreur du type `NameError`, mais le message complet d'erreur donne l'état de la pile d'exécution au moment où l'erreur est survenue : qui a invoqué quoi pour parvenir à cette erreur... En reprenant la liste des messages en sens inverse, on voit que l'erreur provient de la ligne 15, dans le corps de la fonction `c` pour l'appel de la fonction `d`, la fonction `c` ayant été invoquée à la ligne 12 depuis la fonction `b`...

Comme on l'indiquait dans le message précédent : à chaque appel d'une fonction, une entrée est rajoutée à la pile d'exécution, permettant à la fois de garder trace de l'instruction depuis laquelle la fonction a été invoquée,

et un environnement est créé pour permettre de définir des variables locales à la fonction (et si une même fonction est appelée plusieurs fois, par exemple parce qu'elle est récursive, alors il y aura autant d'environnements créés que d'appels imbriqués à cette fonction)

Lorsque la fonction se termine, l'entrée correspondante (forcément la dernière) est retirée de la pile d'exécution.

A noter qu'on ne saurait empiler des objets sur une pile à l'infini. Pour ce qui concerne la pile d'exécution de Python, sa taille est par défaut fixée à 1000 termes : ainsi le code suivant échoue :

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

identite(1000)
```

(En fait, l'appel à `identite(n)` échoue, par défaut, dès la valeur 998.)

En cas de besoin, il est possible d'agrandir la taille maximale permise de la pile, en faisant appel à la fonction `setrecursionlimit(n)` du module `sys`.

En pratique toutefois, devoir augmenter cette taille maximale est le plus souvent le signe que l'utilisation de la récursivité n'est pas adaptée (comme l'exemple ci-dessus) ou qu'une erreur de programmation a été commise, entrainant une fonction à s'appeler sans fin...

2 Terminaison et correction d'un algorithme récursif

Pour prouver qu'un algorithme récursif termine, on utilise comme pour une boucle conditionnelle un variant de boucle, c'est-à-dire une expression prenant une valeur dans $\mathbb{N}$ et qui à chaque appel de la fonction décroît strictement. Exemple avec l'algorithme d'Euclide :

```
def pgcd(a, b):
    """ calcule le pgcd des entiers naturels a et b """
    if b == 0:
        return a
    else:
        return pgcd(b, a % b)
```

Ici un variant de boucle est donné par `b`.

Pour prouver ensuite qu'un algorithme récursif fournit la réponse qu'on attend de lui, le raisonnement s'apparente, sans surprise, à un raisonnement par récurrence.

Pour revenir à l'exemple précédent : le fait que le pgcd de a et b vaille a lorsque $b = 0$ et que le pgcd de a et b sinon soit égal à celui de b et de r où r est le reste de la division euclidienne de a par b donne la preuve de ce que l'algorithme fournit bien la réponse attendue...

Exercice 1 *Donner la preuve de terminaison et de correction des fonctions récursives suivantes :*

```
def factorielle(n):
    """ pour n un entier positif, retourne la factorielle de n """
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)
```

```
def produit(a, b):
    """ étant donnés a et b des entiers positifs, retourne a * b """
    if b == 0:
        return 0
    else:
        return produit(a, b-1) + a

def exposant(x, n):
    """ retourne x**n où n est un entier positif """
    if n == 0:
        return 1
    else:
        if n % 2:
            return x * exposant(x * x, n // 2)
        else:
            return exposant(x * x, n // 2)

def rechercheDichotomique(L, i, j, val):
    """ En supposant L trié dans le sens croissant,
    retourne k tel que i<=k<j et L[k]==val si val est dans L[i:j], et None sinon """
    if i >= j:
        return None
    k = (i + j) // 2
    if L[k] < val:
        return rechercheDichotomique(L, k+1, j, val)
    elif L[k] > val:
        return rechercheDichotomique(L, i, k, val)
    else:
        return k
```

3 Récursivité et complexité

Un exemple classique de mauvaise utilisation de la récursivité est la fonction permettant le calcul de la suite de Fibonacci suivante :

```
def fibo(n):
    "retourne le terme d'indice n de la suite de Fibonacci """
    if n < 2:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

Ici, l'étude de sa complexité est facile, car en notant $T(n)$ le temps nécessaire au calcul de `fibo(n)`, alors $T(n) = O(1)$ si $n < 2$ et, si $n \geqslant 2$, $T(n) = T(n-1) + T(n-2) + O(1)$.

Si on néglige le temps nécessaire à additionner $fibo(n-1) + fibo(n-2)$, on peut en première approche considérer que $T(0) = T(1) = 1$ et pour tout $n \geqslant 2$, $T(n) = T(n-1) + T(n-2)$, et on a alors pour tout n, $T(n)$ égal au terme d'indice $n+1$ de la suite de Fibonacci et il vient $T(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$ (complexité exponentielle donc...)

Comme on devine qu'un algorithme naïf calculant le n-ième terme de la suite de Fibonacci agit en temps linéaire, on comprend que la fonction récursive donnée ci-dessus est peu recommandable...

Une amélioration possible est d'utiliser la *mémoïzation* dont le principe est de garder en mémoire pour chaque appel à la fonction `fibo` les valeurs en entrée et en retour. Si une valeur demandée a déjà été calculée, on se contente d'en lire la valeur dans une table. Cette technique est bien entendu un peu coûteuse en mémoire, et ne corrige pas un écueil de la fonction récursive `fibo` : pour le calcul d'un terme de la suite de Fibonacci d'un indice

important (ne serait-ce que quelques milliers), le nombre d'appels imbriqués de la fonction conduira assez vite à un dépassement de capacité de la pile d'exécution, et donc à un plantage...

Un exemple d'algorithme récursif efficace est celui du calcul de pgcd par la méthode d'Euclide, comme présenté un peu plus tôt, mais l'étude de sa complexité n'est pas facile : grosso-modo on remarque que le pire des cas est obtenu lorsque les arguments `a` et `b` sont deux termes consécutifs de la suite de Fibonacci (encore elle !), auquel cas les arguments des appels successifs à la fonction seront toujours deux termes consécutifs de cette même suite. (Exemple : le pgcd de F_n et F_{n+1} est aussi celui de F_{n-1} et F_n, lequel est celui de F_{n-2} et F_{n-1} et ainsi de suite...)

Le nombre d'appels imbriqués est alors de l'ordre de n (dans le pire des cas) pour des entrées de l'ordre de ϕ^n où ϕ est le nombre d'or, ce qui conduit à une complexité logarithmique (dans le pire des cas), ce qui est bien sûr nettement plus acceptable que la complexité exponentielle obtenue précédemment.

Exercice 2 *Faire l'étude de la complexité des quatre fonctions récursives décrites dans l'exercice 1.*