Récursivité

Fonctions récursives

Une fonction peut s'appeler elle-même au cours de son exécution : on dit alors qu'elle est **récursive**.

Considérons par exemple la fonction suivante :

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

Si on exécute la commande `factorielle(4)` :

– Python voit qu'il doit calculer `factorielle(3)` puis multiplier par 4.

– Pour calculer `factorielle(3)`, Python voit qu'il doit calculer `factorielle(2)` puis multiplier par 3.

– Pour calculer `factorielle(2)`, Python voit qu'il doit calculer `factorielle(1)` puis multiplier par 2.

– Pour calculer `factorielle(1)`, Python voit qu'il doit calculer `factorielle(0)` puis multiplier par 1.

– `factorielle(0)` renvoie 1, puis on remonte les calculs : `factorielle(1)` renvoie $1 \times 1 = 1$, puis `factorielle(2)` renvoie $2 \times 1 = 2$, `factorielle(3)` renvoie $3 \times 2 = 6$ et enfin `factorielle(4)` renvoie $4 \times 6 = 24$.

Le cas $n = 0$ est appelé **cas d'arrêt** de la fonction (on aurait pu prendre $n \leqslant 1$).

Les appels successifs sont stockés en mémoire dans une pile appelée **pile d'exécution**. La taille de celle-ci est limitée :

```
>>> factorielle(1000)
RuntimeError: maximum recursion depth exceeded in comparison
```

Avantages et inconvénients

Considérons par exemple la suite (u_n) définie par $u_0 = 1$ et $u_{n+1} = 2u_n + 1$ pour tout $n \in \mathbb{N}$. Elle peut être programmée en récursif ou en itératif :

```
def u(n):  # version récursive
    if n == 0:
        return 1
    else:
        return 2 * u(n-1) + 1
```

```
def u(n):  # version itérative
    x = 1
    for i in range(n):
        x = 2 * x + 1
    return x
```

La version récursive est particulièrement simple à écrire, mais les appels récursifs successifs prennent de la place en mémoire et leur nombre est limité (cf exercice 5 du TP 9).

Certaines fonctions s'écrivent de manière naturelle en récursif. On peut ainsi produire du code plus court et plus facile à lire qu'avec une méthode itérative.

Considérons par exemple **l'algorithme d'Euclide** qui permet de calculer le pgcd de deux entiers naturels a et b (qu'on note $a \wedge b$) en procédant par divisions euclidiennes successives. Il repose sur le fait que si $a = bq + r$ avec $q, r \in \mathbb{N}$, alors $a \wedge b = b \wedge r$.

Calculons par exemple le pgcd de 336 et de 276. On a :

- $336 = 1 \times 276 + 60$ donc $336 \wedge 276 = 276 \wedge 60$,
- $276 = 4 \times 60 + 36$ donc $276 \wedge 60 = 60 \wedge 36$,
- $60 = 1 \times 36 + 24$ donc $60 \wedge 36 = 36 \wedge 24$,
- $36 = 1 \times 24 + 12$ donc $36 \wedge 24 = 24 \wedge 12$,
- $24 = 2 \times 12 + 0$ donc $24 \wedge 12 = 12 \wedge 0 = 12$.

Le pgcd de 336 et de 276 est donc 12.

On peut programmer cet algorithme récursif de manière très simple :

```
def pgcd(a, b):
    if b == 0:
        return a
    else:
        return pgcd(b, a % b)

>>> pgcd(336, 276)
12
```

On a vu qu'on pouvait calculer efficacement x^n où x est un entier ou un flottant et n un entier naturel en utilisant les égalités $x^{2n} = (x^n) \times (x^n)$ et $x^{2n+1} = x^{2n} \times x$. Par exemple pour calculer 2^{10} on a :

$$2^{10} = 2^5 \times 2^5 \ ; \ 2^5 = 2^4 \times 2 \ ; \ 2^4 = 2^2 \times 2^2 \ ; \ 2^2 = 2^1 \times 2^1 \ ; \ 2^1 = 2^0 \times 2 \ ; \ 2^0 = 1.$$

On peut ainsi écrire très naturellement la fonction récursive correspondante :

```
def puissance_rapide(x, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        p = puissance_rapide(x, n//2)
        return p * p
    return puissance_rapide(x, n-1) * x
```

Cette fonction est presque aussi rapide que la fonction éponyme du cours sur les preuves d'algorithmes (sa complexité asymptotique est $O(\ln n)$) et présente peu de risque de dépassement de taille de la pile (la plus petite valeur de n qui produit ce phénomène est $2^{497} - 1$).

Terminaison, correction et complexité

Une fonction récursive ne se termine pas si on n'arrive pas jusqu'au cas d'arrêt. Par exemple si on essaie de calculer `factorielle(-1)` avec la fonction du premier paragraphe, l'ordinateur va appeler successivement `factorielle(-2)`, `factorielle(-3)`, etc., mais ne retombera jamais sur `factorielle(0)`. Lorsque la pile d'exécution sera saturée il renverra un message d'erreur :

```
>>> factorielle(-1)
RuntimeError: maximum recursion depth exceeded in comparison
```

Pour établir la terminaison d'une fonction récursive on procède comme pour les boucles `while` : on cherche un entier positif qui diminue strictement à chaque appel. Pour la fonction `factorielle` il suffit de prendre `n` (qui diminue de 1 à chaque appel). Pour la fonction `pgcd` précédente, l'entier `b` convient (à chaque appel il est remplacé par `a % b` qui est strictement inférieur à `b` par définition de la division euclidienne).

Pour établir la correction d'une fonction récursive on raisonnera par récurrence (souvent forte).

Pour la fonction `factorielle` la propriété à démontrer est "`factorielle(n)` renvoie $n!$", dont la démonstration (par récurrence simple) est immédiate : l'initialisation correspond au cas d'arrêt et vient de l'égalité $0! = 1$ et l'hérédité découle de l'égalité $n! = n(n-1)!$.

Pour la fonction `pgcd` la propriété à démontrer est "`pgcd(a, b)` renvoie $a \wedge b$", dont la démonstration se fait par récurrence forte sur `b` : l'initialisation vient de l'égalité $a \wedge 0 = a$ et l'hérédité découle de l'égalité $a \wedge b = b \wedge r$ où r est le reste dans la division euclidienne de a par b.

Le calcul de la complexité d'une fonction récursive est souvent délicat. Supposons que cette complexité dépende d'un entier n et notons $C(n)$ le nombre d'opérations élémentaires effectuées par la fonction. On essaie alors de trouver une relation de récurrence qui permet de calculer $C(n)$.

Par exemple, si on note $C(n)$ le nombre de multiplications effectuées par la fonction `factorielle`, on a $C(0) = 0$ et $C(n) = C(n-1) + 1$ pour tout $n \in \mathbb{N}^*$. On en déduit que $C(n) = n$ pour tout $n \in \mathbb{N}$.

Exercice 1 On considère la fonction `puissance_rapide` de la page précédente.

1) Établir sa terminaison et sa correction.

2) On note $C(n)$ le nombre de multiplications que `puissance_rapide(x, n)` effectue.

a) Trouver une relation entre $C(n)$, $C(n/2)$ et $C(n-1)$ (on distinguera les cas n pair et n impair).

b) En déduire $C(2^n)$ et $C(2^n - 1)$.

c) Écrire une fonction récursive `C` qui, recevant un entier naturel n, renvoie la valeur de $C(n)$.

```
>>> C(10)
5
```