Informatique en CPGE (2018-2019)
Récursivité

1 Introduction

Considérons les deux procédures suivantes **compte1** et **compte2** :

```
def compte1(n):
    if n>=0:
        print(n)
        compte1(n-1)

def compte2(n):
    if n>=0:
        compte2(n-1)
        print(n)
```

Un sous-programme est dit **récursif** s'il fait appel à lui-même, comme ici les deux procédures.
Détaillons la procédure **compte1** :

- le test $n \geq 0$ est une condition d'arrêt, obligatoire sinon le sous-programme peut boucler indéfiniment ;
- la valeur de n est affichée si n est un nombre positif ou nul puis il y a un appel à la procédure elle-même en passant le paramètre $n-1$, cet appel est dit récursif ; la valeur $n-1$ passée en paramètre permet de faire décroître la valeur de n pour que le sous-programme ne boucle pas indéfiniment ;
- la valeur de $n-1$ est affichée si $n-1$ est un nombre positif ou nul puis il y a un appel à la procédure elle-même en passant le paramètre $n-2$;
- les appels récursifs continuent jusqu'à ce que le paramètre passé à la procédure prenne la valeur -1 ;
- la dernière valeur affichée est donc 0.

Par exemple l'instruction `compte1(4)` donne :
affichage de 4, appel de `compte1(3)`,
affichage de 3, appel de `compte1(2)`,
affichage de 2, appel de `compte1(1)`
affichage de 1, appel de `compte1(0)`,
affichage de 0, appel de `compte1(-1)`
Le test n'est plus vérifié, c'est terminé.

L'analyse de la procédure **compte2** se fait de la même manière mais cette fois l'affichage ne peut se faire que lorsque la procédure appelée a été exécutée.
Donc l'instruction `compte2(4)` donne :
appel de `compte2(3)`, (l'affichage de 4 est en attente),
appel de `compte2(2)`, (l'affichage de 3 est en attente),
appel de `compte2(1)`, (l'affichage de 2 est en attente),
appel de `compte2(0)`, (l'affichage de 1 est en attente),
appel de `compte2(-1)`, (l'affichage de 0 est en attente),
le test n'est plus vérifié, donc

affichage de 0,
affichage de 1,
affichage de 2,
affichage de 3,
affichage de 4.

Dans le premier cas, la procédure **compte1**, le sous-programme est dit **récursif terminal** car la dernière instruction exécutée est un appel récursif, ce qui n'est pas le cas pour la procédure **compte2**.

La procédure **compte1** peut être remplacée par une procédure **itérative**, c'est-à-dire un programme utilisant des boucles plutôt que la récursivité :

```
def compte1_iteratif(n):
    while n>=0:
        print(n)
        n=n-1
```

La récursivité, en évitant de faire des boucles, facilite souvent l'écriture d'un programme plus facile à implémenter de manière récursive qu'itérative.

Mais cependant, du point de vue de la complexité, un programme itératif peut être beaucoup plus efficace qu'un programme récursif.

2 Fonction récursive

Définition

Une fonction est dite récursive si elle s'appelle elle-même. Ici encore, il faut toujours faire attention à ce que la fonction ne s'appelle pas indéfiniment.

Voici un exemple d'une fonction qui calcule n ! :

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

La condition d'arrêt est testée dès le début de l'exécution de la fonction ; elle permet d'arrêter les appels récursifs et la valeur 1, (0 ! ou 1 !), est retournée. En effet, si n est le paramètre initial, la valeur passée en paramètre dans l'appel récursif est $n-1$ et la séquence des valeurs passées en paramètre dans la suite des appels récursifs est donc $n \to n-1 \to \ldots \to 2 \to 1$, suite qui décroit strictement vers 1, valeur satisfaisant la condition d'arrêt.

Remarque 1 : cet algorithme n'est pas récursif terminal car l'appel récursif est suivi d'une multiplication par n.

Remarque 2 : la fonction factorielle peut s'écrire avec une boucle **for** :

```
def factorielle(n):
    f=1
    for i in range(2,n+1):
        f=f*i
    return f
```

ou avec une boucle **while** :

```
def factorielle(n):
    f=1
    while n>0:
        f=f*n
        n=n-1
    return f
```

Et en général, les fonctions récursives de la forme

```
if(condition):
    appel récursif
else:
    pas d'appel
```

peuvent s'écrire simplement sans récursivité avec une boucle **while** :

```
while(condition):
    instructions
```

Dans l'exemple qui suit, nous définissons une fonction récursive **somme** qui renvoie la somme des termes d'une liste de nombres et nous allons détailler le travail de l'interpréteur lors de l'appel de la fonction.

```
def somme(liste):
    if liste==[]:
        return 0
    else:
        return liste[0]+somme(liste[1:])
```

S nous appelons cette fonction avec l'instruction `somme([1,3,5,7])`, le déroulement est le suivant :

1 + somme([3, 5, 7])
1 + (3 + somme([5, 7]))
1 + (3 + (5 + somme([7])))
1 + (3 + (5 + (7 + somme([ ]))))
fin des appels récursifs
1 + (3 + (5 + (7 + 0)))
1 + (3 + (5 + 7))
1 + (3 + 12)
1 + 15
16

3 Sous-programmes récursifs terminaux

Un sous-programme non terminal peut souvent être réécrit de façon terminale. Pour cela, on ajoute un paramètre r, appelé accumulateur, et il faut faire en sorte que le paramètre r contienne le résultat lorsque la condition d'arrêt est vérifiée.

```
def fonction(...,r):
    if condition d'arrêt:
        return r
    else
        instructions
```

Par exemple :

```
def factorielleT(n,r):
    if n==0:
        return r
    else:
        return factorielleT(n-1,r*n)
```

La fonction **factorielleT** est bien une fonction récursive terminale.

Nous montrerons plus loin, lors de l'étude de la correction d'un algorithme récursif, que factorielleT(n, r) = r (n !) pour tout réel r et tout entier naturel n et donc que factorielleT(n, 1) = n ! en prenant r = 1.

4 Récursivité et notion de pile

Considérons la fonction **factorielle** définie récursivement par :

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

Pour calculer factorielle(10) par exemple, les produits s'effectuent de la manière suivante :

$$10 \times (9 \times (8 \times (7 \times (\ldots (2 \times 1) \ldots))))$$

Ceci revient à utiliser une pile sur laquelle nous empilons successivement tous les opérandes 10, 9, 8, . . . , 2, 1 ; ensuite, neuf fois, nous dépilons deux opérandes et nous empilons leur produit.

En notation polonaise inverse, qui a été rencontrée au chapitre 1, l'expression s'écrit : 10 9 8 . . . 2 1 × × . . . ×

Considérons maintenant la fonction **factorielleT** qui est une fonction récursive terminale :

```
def factorielleT(n,r):
    if n==0:
        return r
    else:
        return factorielleT(n-1,r*n)
```

La suite des appels pour calculer $10\,!$ est :
factorielleT(10,1), factorielleT(9, 10), factorielleT(8, 90), ...

Cette fois, les produits sont effectués de la manière suivante :

$$((\ldots(((10\times 9)\times 8)\times 7)\times\ldots)\times 2)\times 1$$

Donc si nous utilisons une pile, la procédure est la suivante : les opérandes 10 et 9 sont empilés, puis ils sont dépilés et leur produit est empilé ; ensuite, huit fois de suite, l'opérande qui suit est empilé, deux opérandes sont dépilés et leur produit est empilé.

La taille de la pile ne dépasse donc jamais deux et en notation polonaise inverse, l'expression à calculer s'écrit : $10\ 9\ \times\ 8\ \times\ 7\ \times\ \ldots\ 2\ \times\ 1\ \times$

Il est évident, en pratique, qu'il s'agit aussi de stocker les appels récursifs et les éventuelles opérations à exécuter. Et un problème survient naturellement si le nombre d'appels récursifs est trop grand, s'il dépasse la capacité maximale prévue. Dans ce cas, le programme récursif ne termine pas et nous recevons le message d'erreur : "*RuntimeError : maximum recursion depth exceeded in comparison*".

Attention : par défaut, le nombre d'appel récursif est limité à 1000 ; nous pouvons le modifier avec les instructions :

```
import sys
sys.setrecursionlimit(5000)
```
(par exemple)

Mais bien sûr le nombre maximal d'appels récursifs est limité !

5 Validité d'un algorithme récursif

Nous allons étudier la terminaison et la correction d'un algorithme récursif. Pour la terminaison, nous avons déjà parlé de la condition d'arrêt des appels récursifs, et il faudra éventuellement compléter le code par un test qui permet de traiter tous les cas possibles.

En ce qui concerne la correction, une démonstration par récurrence peut être envisagée dans de nombreux cas.

5.1 Terminaison

C'est la condition d'arrêt qui doit assurer la terminaison.

Reprenons l'exemple de la fonction **factorielleT** définie comme suit :

```
def factorielleT(n,r):
    if n==0:
        return r
    else:
        return factorielleT(n-1,r*n)
```

Si un utilisateur teste cette fonction avec un entier naturel n quelconque, la condition d'arrêt et la décroissance de la suite des valeurs passées en paramètres assurent la terminaison après n appels récursifs. Par contre, si le paramètre entré par un utilisateur est négatif ou si ce n'est pas un entier naturel, les appels récursifs vont se poursuivre indéfiniment, ou plutôt tant que la machine le permet. Il est donc nécessaire de compléter le code pour traiter ce cas, par exemple avec un test.

```
def factorielleT(n,r):
    if n<0:
        print("Erreur sur n")
    elif n==0:
        return r
    else:
        return factorielleT(n-1,r*n)
```

Maintenant la terminaison est assurée pour n'importe quel nombre n passé en paramètre.

Remarque : le langage Python propose aussi différentes possibilités comme l'utilisation de **assert** ou de **raise**, ou encore le bloc **try/except**, mais ceci dépasse le cadre de cet ouvrage.

Notons quand même que nous pourrions écrire par exemple `assert n>=0`.

Si la condition qui suit **assert** n'est pas vérifiée, alors le programme s'arrête en signalant une erreur. Donc **assert** est plutôt utilisé en phase de test ou de "debug".

5.2 Correction

La correction d'un programme récursif se démontre en général avec un raisonnement par récurrence.

Considérons à nouveau l'exemple précédent et démontrons la correction par une récurrence sur n. Soit la propriété H_n : "pour tout entier naturel n et tout réel r, factorielleT(n,r) renvoie r $\times$ n ! ".

- Initialisation : si n = 0 alors, pour tout réel r, factorielleT(0, r) renvoie r, soit r (0 !). La propriété est vraie pour n = 0.
- Hérédité : supposons que pour un entier k quelconque et pour tout réel r, factorielleT(k, r) renvoie r (k !), alors :
  factorielleT(k + 1, r) renvoie factorielleT(k, r (k + 1)), c'est l'appel récursif, alors, d'après notre hypothèse, factorielleT(k + 1, r) renvoie r (k + 1)(k !), soit r ((k+1) !). La propriété est vraie au rang k + 1.
- Conclusion : la propriété est vraie pour tout entier naturel n et tout réel r.

6 Complexité d'un algorithme récursif

L'étude de la complexité d'un algorithme récursif consiste principalement à déterminer le nombre d'appels récursifs en fonction de n, le nombre ou la taille de l'objet en entrée. En règle générale, cette complexité peut s'exprimer par une relation de récurrence.

6.1 Calculs exacts

Commençons par un exemple simple et notons u_n le nombre d'instructions élémentaires exécutées pour une taille égale à n. Supposons ensuite que la condition d'arrêt est n = 0 et que pour cette valeur l'algorithme s'exécute en un temps u_0 constant. Supposons enfin que si n $\neq$ 0, l'appel récursif nécessite u_{n-1} opérations, et que si d'autres opérations sont exécutées, elles le sont en temps constant k. La suite (u_n) est alors définie pour n $>$ 0 par : $u_n = u_{n-1} + k$.

Nous pouvons bien sûr exprimer u_n en fonction de u_0, n et k dans ce cas précis, puisque la suite (u_n) est une suite arithmétique de raison k : $u_n = u_0 + nk$.

La complexité est donc linéaire, $u_n \in \mathcal{O}(n)$.

Considérons maintenant la suite de Fibonacci définie par $f_n = f_{n-1} + f_{n-2}$ pour $n > 1$ avec $f_0 = f_1 = 1$ et une fonction **fibonacci** définie de manière récursive :

```
def fibonacci(n):
    if n<=1:
        return 1
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Déterminer la complexité ici est un peu plus difficile. Le calcul du nombre f_n, pour n > 1, demande un test, deux appels de la fonction et une addition. Le calcul du nombre total de tests, d'appels de la fonction et d'additions se traite de la même manière.

Soit k > 1 ; pour obtenir le nombre f_k, nous appelons la fonction avec le paramètre k, qui appelle alors la fonction avec les paramètres k -1 puis k -2. Donc si u_k est le nombre d'appels de la fonction pour un paramètre k, alors u_k vérifie la relation : $u_k = 1 + u_{k-1} + u_{k-2}$ pour $k \geq 2$ avec $u_0 = u_1 = 1$.

L'objectif est d'exprimer u_k en fonction de k. La relation de récurrence nous donne : $u_k + 1 = 1 + u_{k-1} + u_{k-2} + 1$ et en posant $v_k = u_k + 1$, nous obtenons $v_k = v_{k-1} + v_{k-2}$ avec $v_0 = v_1 = 2$. Il est facile maintenant de montrer par récurrence que $v_k = 2f_k$ et nous en déduisons que $u_k = 2f_k - 1$.

De même si nous notons a_k le nombre d'additions pour un paramètre k, alors $a_k = 1 + a_{k-1} + a_{k-2}$ pour $k \geq 2$ avec $a_0 = a_1 = 0$. Nous posons $b_k = a_k + 1$ et nous obtenons $b_k = b_{k-1} + b_{k-2}$ pour $k \geq 2$ avec $b_0 = b_1 = 1$. Donc $b_k = f_k$ et $a_k = f_k - 1$.

Nous procédons de même pour le nombre de tests.

Le temps d'exécution du calcul de f_n est donc de l'ordre de la valeur de f_n.

Or le nombre f_n est égal à l'entier le plus proche de $\dfrac{1}{\sqrt{5}}\left(\dfrac{1+\sqrt{5}}{2}\right)^{n+1}$.

La démonstration (par récurrence) est laissée au lecteur.

Nous pouvons alors en déduire que le calcul de f_n avec la fonction **fibonacci** a une complexité exponentielle.

Un calcul exact n'est pas toujours possible et souvent il peut être très compliqué de trouver une expression de u_n en fonction de u_0 et de n ; nous emploierons alors la méthode suivante.

6.2 Vérification par récurrence

Une manière de classer une suite (u_n) dans un ordre de grandeur est de conjecturer cet ordre, ou un encadrement de cet ordre, ou une majoration de cet ordre, puis de démontrer cette conjecture par récurrence. Maintenant, faire une "bonne" conjecture est parfois plus difficile qu'en faire sa démonstration par récurrence.

Dans l'exemple précédent, l'observation des différents appels de la fonction **fibonacci** pour la calcul de f_n, peut conduire à conjecturer l'encadrement suivant (où Ent est la partie entière) : $2^{\text{Ent}(n/2)+1} - 1 \leq u_n \leq 2^n - 1$.

Par exemple pour f_5 : $2^0 + 2^1 + 2^2 \leq u_5 \leq 2^0 + 2^1 + 2^2 + 2^3 + 2^4$, soit $2^3 - 1 \leq u_5 \leq 2^5 - 1$. (Encadrement très large : $u_5 = 15$ et $7 \leq u_5 \leq 31$).