LA RÉCURSIVITÉ

PSI

2021-2022

Résumé

Ce chapitre aborde un nouvel aspect de la programmation : les fonctions récursives. Ces sont des fonctions qui s'appellent elles-mêmes dans leurs propres code.

Table des matières

1 Écriture de fonctions récursives

1.1 Exemple d'introduction

La suite des factorielles des entiers naturels peut être définie par récurrence :

$$\begin{cases} 0! & = & 1 \\ (n+1)! & = & (n+1)\cdot n! \end{cases}$$

La fonction ci-dessous exploite cette définition :

```
1 def fact(n):
2   if n == 0:
3     return 1
4   else :
5     return n * fact(n -1)
```

On remarque que la fonction `fact` s'appelle elle-même.

Une fonction Pyhon qui s'appelle elle-même est dite *récursive*, ces fonctions sont naturellement adaptées pour étudier les structures définies par récurrence.

Arbre d'appels

Étudions par exemple l'appel à `fact(5)`. Il peut être représenté par le schéma suivant appelé *arbre d'appels*.

```
fact(5)
  |
5 * fact(4)
     |
   4 * fact(3)
        |
      3 * fact(2)
           |
         2 * fact(1)
              |
            1 * fact(0)
```

Comme nous pouvons le constater, l'appel `fact(5)` attend le résultat de `fact(4)`, qui lui-même attend le résultat de `fact(3)`, etc. Le seul appel qui termine directement est `fact(0)`. Il renvoie la valeur 1 à `fact(1)` qui à son tour peut calculer sa valeur de retour $1 \times 1 = 1$ et la renvoyer à `fact(2)`, etc.

Remarque 1.1. L'occupation de la mémoire augmente avec le nombre d'appels récursifs. La mémoire est gérée sous la forme d'une pile appelée *pile d'exécution* dans laquelle à chaque appel récursif, les variables nécessaires à l'exécution de la fonction sont ajoutées au sommet de la pile. Ces variables sont supprimées (dépilées) lorsque la fonction retourne sa valeur.
Si le nombre d'appels récursifs est trop important la mémoire allouée au programme peut être dépassée. On dit que la pile d'exécution est débordée.

En Python, le nombre maximum d'appels récursifs (profondeur de l'arbre d'appel) en Python a été arbitrairement fixé à 1000. Si ce nombre est dépassé le message d'erreur suivant apparaît : `RuntimeError: maximum recursion depth exceeded in comparison` . On testera l'appel `fact(1200)` par exemple.

Cette limitation est suffisante dans de nombreux algorithmes. Cependant elle peut devenir gênante pour mettre en œuvre certains algorithmes qui par nature effectuent un grand nombre d'appels récursifs.

Remarque 1.2. Il est toujours possible d'écrire une version itérative d'un algorithme. On peut par exemple peut s'écrire avec une boucle `for` :

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

Dans le code de la fonction `fact`, on a décrit la fonction à calculer. On parle de programmation *fonctionnelle*. Dans le code de la fonction `fact_it` on a indiqué comment les calculs doivent être faits. On parle de programmation *impérative*. Notons l'absence d'affectation dans le code de la fonction `fact`, contrairement au code de la fonction `fact_it`.

1.2 Autre exemple : suite de Fibonacci

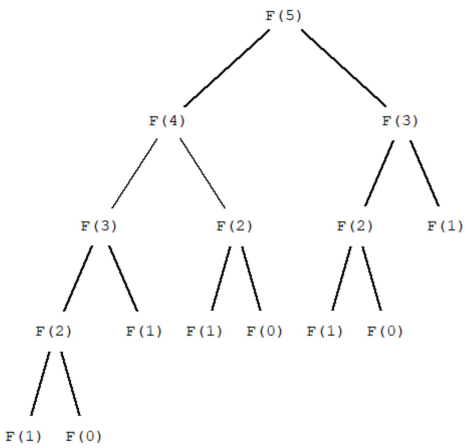
Il est possible qu'une fonction fasse plusieurs appels récursifs. Prenons l'exemple de la suite définie sur **N** :

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \text{ pour } n \geqslant 2. \end{cases}$$

La fonction ci-dessous exploite la définition de cette suite :

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

Remarque 1.3. On pourra constater sur ordinateur que l'exécution de la fonction `F` est très lente. Cela est dû à des problèmes de complexité qui seront étudiés dans un prochain paragraphe et peut se voir sur l'arbre d'appel de `F(5)` où on remarque des calculs redondants :



2 Terminaison d'une fonction récursive

Pour garantir la terminaison d'une fonction récursive il faut

— S'assurer qu'on a traité les cas de base qui n'effectuent pas d'appel récursif.

— Vérifier qu'à chaque appel récursif, les valeurs passées en arguments sont plus "petites" que les précédentes, en terme de taille par exemple.

Les preuves de terminaison, de correction et les calculs de complexité se font par récurrence (voir l'annexe).

3 Exercices

1. On considère la fonction `S` suivante :

```
1 def S(n) :
2     """ n entier naturel """
3     if n == 0 :
4         return 0
5     else :
6         return n+S(n-1)
```

(a) Dessiner l'arbre d'appel de `S(5)` et préciser la valeur renvoyée.

(b) Plus généralement, que renvoie l'appel `S(`n`)`, $n \in \mathbb{N}$?

2. On considère la fonction `f` suivante :

```
1 def f(n) :
2     if n==0 :
3         return 0
4     else :
5         return f(n-2)
```

(a) Que renvoie l'appel `f(12)` ?

(b) L'appel `f(`n`)` se termine-t-il pour toutes les valeurs de n ?

(c) Ajouter des lignes de code à la fonction `f`, après la troisième ligne, pour que `f(`n`)` renvoie 0 si n est un entier naturel pair et 1 si n est un entier naturel impair.

3. **Calcul du PGCD avec l'algorithme d'Euclide**
L'algorithme d'Euclide permet le calcul du PGCD de deux entiers naturels, en utilisant la méthode des divisions euclidiennes successives décrite ci-dessous :
Soient n et m deux entiers naturels.

— Si $m = 0$, alors $\text{PGCD}(n, m) = n$

— Si $m \neq 0$, notons r le reste de la division euclidienne de n par m. Alors $\text{PGCD}(n, m) = \text{PGCD}(m, r)$.

Écrire une fonction récursive calculant le PGCD de deux nombres entiers naturels exploitant cette définition.

4. **Exponentiation rapide**
Pour accélérer la performance de certains algorithme il est utile de disposer d'une méthode de calcul rapide des puissances entières. On exploite l'idée suivante :

— $x^0 = 1$

— $x^{2p} = (x^p)^2$, $x \in \mathbb{R}$, $p \in \mathbb{N}^*$.
— $x^{2p+1} = x.(x^p)^2$, $x \in \mathbb{R}$, $p \in \mathbb{N}$.

(a) Écrire une fonction `pr(x,n)` qui exploite cette idée.

(b) Quel est le nombre de multiplications effectuées pour calculer a^n lorsque $n = 2^p$, $p \in \mathbb{N}^*$.

5. Complexité

On considère les deux fonctions récursives suivantes :

```
1  def g(n):
2    if n == 0:
3      return 2
4    else :
5      return (g(n-1)+2/g(n-1))/2
6  def h(n):
7    if n == 0:
8      return 2
9    else :
10     x= h(n-1)
11     return (x+2/x)/2
```

(a) Est-ce que les deux fonctions `g` et `h` renvoient les mêmes résultats ?

(b) Comparer la complexité de ces deux fonctions (on comptera le nombre d'opérations arithmétiques et le nombre de comparaisons.

6. Suite de Fibonacci

(a) Calcul de la complexité pour la suite de Fibonacci : Revenons à l'exemple 1.2. On veut évaluer ici le coût de l'appel `F(n)` pour un entier naturel n. Appelons $C(n)$ le nombre de comparaisons et d'additions effectuées par cet appel. Établir une relation de récurrence entre $C(n)$, $C(n-1)$ et $C(n-2)$, puis vérifier que la suite A_n définie pour tout entier n par $A_n = C(n) + 2$ vérifie la relation de récurrence $A_n = A_{n-1} + A_{n-2}$ pour tout entier $n \geqslant 2$.

(b) En déduire que $C(n) = K.\phi^n + K'.\phi'^n - 2$ avec $\phi = \dfrac{1+\sqrt{5}}{2}$, $\phi' = \dfrac{1-\sqrt{5}}{2}$, $K = \dfrac{3(5+\sqrt{5})}{10}$ et $K' = \dfrac{3(5-\sqrt{5})}{10}$.

(c) Afin d'accélérer le calcul de la suite de Fibonacci tout en gardant une programmation récursive, on écrit une fonction auxiliaire de la façon suivante :

```
1    def F_aux(n,f1,f2):
2    if n==0:
3    return f1
4    else :
5    return F_aux(n-1,...,...)
```

Compléter cette fonction afin que l'appel `F_aux(`n`,0,1)` renvoie F_n pour tout entier naturel n. Calculer la complexité de `F_aux(`n`,0,1)` en fonction de n (on comptera le nombre d'additions et de comparaisons).

Remarque 3.1. Dans la fonction `F_aux` aucun traitement n'est fait sur la valeur de retour de l'appel récursif. On dit dans ce cas que la récursion est *terminale*.

(d) Écrire une fonction itérative permettant de calculer les termes de la suite de Fibonacci et calculer sa complexité.

Principe "diviser pour régner"

On dit qu'un algorithme obéit au principe "diviser pour régner" s'il permet de résoudre un problème en suivant les étapes suivantes :

— résolution immédiate d'un ou plusieurs cas d'arrêts ;

— découpage du problème en plusieurs sous-problèmes dont le traitement est fait par l'algorithme ;

— combinaison des solutions des sous-problèmes pour résoudre le problème.

Les algorithmes de ce type sont naturellement récursifs.
L'algorithme du calcul rapide des puissances est du type "diviser pour régner", ainsi que celui du TP utilisé pour résoudre le problème des tours de Hanoï.

3.1 Exercice

— On veut trouver la position d'un élément `e` dans une liste `L` triée. Expliquer en quoi la recherche par dichotomie ne suit pas le principe du "diviser pour régner"

— On suppose maintenant que la liste `L` n'est pas triée, proposer un algorithme récursif utilisant le principe "diviser pour régner" pour trouver la position de cet élément.

— On suppose finalement que l'élément `e` peut se trouver plusieurs fois dans la liste non triée `L`. Adapter l'algorithme pour qu'il renvoie la liste des positions de `e`.

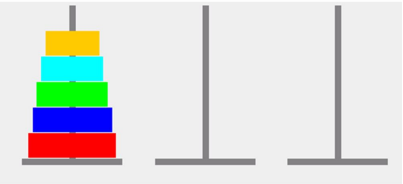
4 TD : Les tours de Hanoï

Le jeu des tours de Hanoï a été inventé par Édouard Lucas en 1883. Il comporte 3 colonnes dans lesquelles on peut empiler des anneaux. Les anneaux sont au nombre de cinq et sont tous de diamètres différents. A l'état initial, la première colonne comporte les 5 anneaux empilés dans l'ordre décroissant de leurs diamètres. On déplace les anneaux en respectant les règles suivantes :

— on ne peut déplacer qu'un seul anneau à la fois (déplacement élémentaire) et il doit être situé au sommet d'une colonne ;

— L'anneau déplacé peut être déposé soit à la base d'une tour si elle est vide, soit sur un autre anneau de taille supérieure.

Le but du jeu est alors de déplacer tous les anneaux de la première vers la troisième colonne.



1. Écrire une fonction récursive `hanoi(c1,c2,c3,n)` qui affiche une liste de déplacements élémentaires permettant de déplacer `n` anneaux de la colonne `c1`, à la colonne `c3` en utilisant la colonne intermédiaire `c2`. Un déplacement sera décrit par une chaine de caractères. Par exemple `"12"` voudra dire qu'on retire l'anneau du sommet de la colonne 1 pour le placer sur la colonne 2.

   Indication : Pour déplacer n anneaux d'une colonne a vers une colonne b, on peut procéder de la façon suivante :

   — **Résolution des cas d'arrêts**
   Si $n = 1$, on fait un seul déplacement de la colonne a vers la colonne b.

   — **Découpage en sous problèmes**
   Si $n > 1$, on déplace les $n - 1$ plus petits anneaux de la colonne a vers la troisième colonne c en utilisant la colonne b.

   — **Combinaison des solutions**
   On déplace le n^{ime} anneau de la colonne a vers la colonne b. On déplace enfin les $n - 1$ plus petits anneaux de la colonne c vers la colonne b en utilisant la colonne a.

2. Afficher la solution du jeu en effectuant un appel à la fonction `hanoi`. On testera la solution sur le site `http://championmath.free.fr/tourhanoi.htm`

3. Calculer la complexité de la fonction `hanoi` en fonction de la valeur du paramètre `n`. On comptera le nombre de déplacements effectués.

5 Annexe : preuves de terminaison et complexité

Preuve de terminaison et de correction de l'exemple 1.1

Appelons H_n la propriété suivante :

$$H_n : \texttt{fact}(n) \text{ se termine et renvoie la valeur } n!.$$

— Initialisation : H_0 est vérifié puisque l'appel `fact(0)` se réduit à l'instruction `return 1`. Il se termine et renvoie bien 0!.

— Hérédité : Soit n un entier naturel non nul. Supposons que H_{n-1} est vérifiée. `fact(`n`)` commence par un appel récursif à `fact(`$n-1$`)`. Par hypothèse de récurrence, cet appel se termine et renvoie la valeur $(n-1)!$. L'appel de `fact(`n`)` se poursuit donc par le calcul du produit $n.(n-1)!$. Il se termine donc et renvoie la valeur $n!$.

Remarque 5.1. La preuve de terminaison et de correction de l'exemple 1.2 se fait en utilisant une récurrence double.

5.1 Complexité en temps

On expose ici une méthode pour calculer le coût d'une fonction récursive. Reprenons l'exemple 1.1. Appelons $C(n)$ le nombre de multiplications effectuées par l'appel `fact(`n`)` pour un entier naturel n. Si $n = 0$ aucune multiplication n'est effectuée. Si $n \geqslant 1$, `fact(`n`)` commence par l'appel à `fact(`$n-1$`)`, qui nécessite $C(n-1)$ multiplications, et se termine par une multiplication. On en déduit les équations suivantes :

$$\begin{cases} C(0) &= 0 \\ C(n) &= C(n-1)+1 \end{cases}$$

Il s'agit d'une suite arithmétique de raison 1. On en déduit que $C(n) = n$.
L'étude de la complexité du calcul de la complexité de la suite de Fibonacci de l'exemple 1.2 est proposé dans l'exercice **6**.a.

5.2 Complexité en espace

Comme nous l'avons vu dans la description de l'algorithme l'exemple 1.1 du calcul de factorielles, chaque appel récursif alloue de la mémoire pour les paramètres et les variables locales de cet appel. Le nombre de cases mémoires allouées par une fonction récursive a donc pour majorant le produit du nombre d'appels récursifs par le nombre de variables allouées à chaque appel. Dans l'exemple 1.1, le nombre de variables créées pour chaque appel récursif est 2 (`n` et `x`). L'appel `fact(`n`)` effectue n appels récursifs et donc le nombre de cases mémoire nécessaires à cet appel est $2n$.
En revanche, pour un appel `F(`n`)` de l'exemple 1.2 de la suite de Fibonacci, le calcul de l'espace mémoire nécessaire est moins évident. En effet, la mémoire allouée lors de l'appel récursif `F(`$n-1$`)` peut être réutilisé pour l'appel `F(`$n-2$`)` (ce qui est réalisé en pratique).

5.3 Exercices

— Prouver la terminaison de la fonction `F` définie dans l'exemple 1.2.

— On considère la fonction `p` suivante :

```
1 def p(x,n) :
2     """ x de type float
3         n entier naturel """
4     if n==0 :
5         return 1
6     else :
7         return x*p(x,n-1)
```

1. Que renvoient les appels `p(4,2)`, `p(2,6)` ?
2. Plus généralement, que renvoie l'appel `p(`x`,`n`)`, $x \in \mathbb{R}$ et $n \in \mathbb{N}$?
3. Faire une preuve par récurrence de la formule obtenue à la question précédente.