Chapitre V

Récursivité 1 : première approche

Figure V.1 – Mise en abyme : un exemple photographique de récursivité

Sommaire

1 Fondements

1.1 "Construisons" la récursivité

Supposons que nous souhaitions fabriquer une fonction affichant la suite des puissances n$^{\text{ième}}$ de 2 dans un ordre décroissant :

Un méthode simple est de faire appel à une boucle inconditionnelle :

Listing V.1 –

```
def deux_exp(n):
    for i in range (n+1):
        print 2**(n-i)

deux_exp(5)
```

La sortie sera la suivante :

```
32
16
8
4
2
1
```

Dans cet exemple, la boucle itérative provoque 6 appels à la fonction `print`.

Nous pourrions limiter le nombre d'itérations en commençant par exemple par afficher 2^5 et en calculant ensuite `deux_exp(4)` :

Listing V.2 –

```
n=5
print 2**n
print deux_exp(n-1)
```

En intégrant ce principe dans une fonction cela donne :

Listing V.3 –

```
def deux_exp_2(n):
    print 2**n
    deux_exp(n-1)
```

Ainsi, nous réalisons l'affichage attendu en deux étapes : $\left[\begin{array}{c} \text{l'affichage "direct" de } 2^n \\ \text{l'affichage par appel à la fonction de } 2^{n-1}, 2^{n-2}, \ldots, 2^0 \end{array} \right.$

En répétant ce processus, nous pourrions ainsi bâtir autant de fonctions `deux_exp` que "nécessaire" pour parvenir au même résultat. L'inconvénient majeur est qu'il existe alors autant de fonctions que de puissances à calculer, soit $n+1$ finalement. Ce défaut interdit naturellement un usage général de cette méthode puisque n n'est jamais connu à l'avance ! ! !

En reprenant l'exemple de calcul de la puissance n$^{\text{ième}}$ de 2, nous pourrions éviter l'appel à la fonction puissance intégrée de Python en exploitant la suite récurrente $u_n = 2 \times u_{n-1}$ de premier terme $u_0 = 1$:

Listing V.4 –

```
def deux_exp_imp(n):
    res=1
    for i in range (n):
        res=2*res
    return res
print deux_exp_imp(10)
```

La sortie donne :

```
1024
```

Il est possible de remplacer ce mode d'évaluation itératif, appelé **programmation impérative**, par une programmation dite **fonctionnelle**, dans laquelle les fonctions s'appellent elles-même. Ce mode de programmation est qualifié de **récursif**. Cela donne avec l'exemple précédent :

Listing V.5 –

```
def deux_exp_rec(n):
    if n==0:
        return 1
    else:
        return 2*deux_exp_rec(n-1)
print deux_exp_rec(5)
```

REMARQUE - (1.1) - 1:

le nombre maximum d'appels récursifs est limité à 999 en python.
Que se passe-t-il si l'on lance le script suivant ?

Listing V.6 –

```
def deux_exp_rec(n):
    if n==0:
        return 1
    else:
        return 2*deux_exp_rec(n-1)
print deux_exp_rec(999)
```

EXERCICE N°1: *Faire tourner le script précédent à la main afin d'écrire les résultats intermédiaires du calcul de* `deux_exp_rec(5)`.

RÉPONSE :

n=5	**32**				
n-1=4	$2 \times \underbrace{\texttt{deux\_exp\_rec(4)}}_{=?}$	↖ 16			
n-2=3	↘	$2 \times \underbrace{\texttt{deux\_exp\_rec(3)}}_{=?}$	↖ 8		
n-3=2		↘	$2 \times \underbrace{\texttt{deux\_exp\_rec(2)}}_{=?}$	↖ 4	
n-4=1			↘	$2 \times \underbrace{\texttt{deux\_exp\_rec(1)}}_{=?}$	↖ 2
n-5=0				↘	$2 \times \underbrace{\texttt{deux\_exp\_rec(0)}}_{=1}$

COMMENTAIRES :

- On constate que les appels récursifs successifs imposent de mettre en suspens chaque étape de calcul jusqu'à la dernière itération.
  La récursivité impose parfois le stockage de nombreux calculs intermédiaires qui conduisent à une forte utilisation de la mémoire $\Longrightarrow$ la récursivité engendre **une forte complexité spatiale**. La partie de la mémoire dévolue au stockage "intermédiaire" porte le nom de PILE D'EXÉCUTION et est de type "LIFO" pour LAST IN FIRST OUT. C'est en effet le dernier calcul inscrit en mémoire qui sera évalué le premier, puis l'avant dernier inscrit évalué en second et ainsi de suite jusqu'à la **remontée**[1] au premier appel.
- Pour la dernière itération, soit pour $n-5=0$, on rencontre ce que l'on appelle **le cas de base**, structure conditionnelle permettant d'évaluer directement la valeur de la fonction à ce rang final $\Longrightarrow$ le cas de base **permet d'assurer la terminaison de la fonction**.

QUESTION :

❶ Quelle est la hauteur de la pile d'exécution dans l'exemple `deux_exp_rec(5)`.

❷ Même question pour le cas général `deux_exp_rec(n)`.

1.2 Définition

DÉFINITION - (1.2) - 1:

Une fonction récursive doit contenir les éléments fondamentaux suivants :

- *un (récursivité simple) ou plusieurs (récursivité multiple)* **appel(s) à la fonction elle-même** *lors de son exécution.*
- *un cas de base, c'est à dire* **une situation conditionnelle présente dans la fonction** *qui assure* **sa terminaison**.

1.3 Principe de conception d'une fonction récursive

La mise au point d'un algorithme comprenant une procédure récursive de traitement T sur des données D passe par les étapes générales suivantes :

- Identifier les variables du problème
- Dégager le cas de base qui doit conduire **à la terminaison de la procédure récursive**.

1. cette notion est importante pour la suite !

- Décomposer le traitement T en un ensemble de sous-traitements identiques au traitement de départ et dans lequel les variables convergent vers le cas de base.
- Ecrire l'algorithme.

1.4 Quelques exemples classiques simples

a - Factorielle

Un cas ultra classique d'usage de la récursivité est le calcul de la fonction factorielle, donc le script Python est le suivant :

Listing V.7 – Fonction factorielle en récursif

```
def fact(N):
    if N==1: #cas de base!!!
        return 1
    else:
        return N*(fact(N-1)) #récurrence convergent vers le cas de base
n=int(input("Entrez un entier positif n: "))
print fact(n)
```

b - PGCD récursif

L'un des algorithmes itératifs de calcul du PGCD de deux nombres a et b s'appuie sur la division euclidienne(algorithme d'Euclide)[2]. Son principe est le suivant :

- on calcule le reste de $a /\!/ b$ que l'on stocke dans r.
- tant que $a\%b \neq 0$ faire : $\begin{cases} r = a\%b \\ a = b \\ b = r \end{cases}$
- On renvoie b.

Un exemple de rotation "à la main" est le suivant : cherchons le PGCD de 96 et 81, donc :

$$\begin{array}{ccrcl l}
a & & b & & r & \\
96 & = & 1 \times 81 & + & 15 & \\
81 & = & 5 \times 15 & + & 6 & \\
15 & = & 2 \times 6 & + & 3 & \text{la boucle s'arrête ici car.....} \\
6 & = & 2 \times \boxed{\mathbf{3}} & + & 0 & \text{sa condition est violée au rang suivant !}
\end{array}$$

EXERCICE N°2: *Proposer un algorithme récursif de calcul du PGCD.*

c - Conjecture de Syracuse

EXERCICE N°3: *On définit la suite de Syracuse par :*

$$\left[\begin{array}{lcc}
x_1 & = & a \in \mathbb{N}^* \\
x_{n+1} & = & \begin{cases} \dfrac{x_n}{2} \ \textit{si } x_n \textit{ est pair} \\ 3 \times x_n + 1 \ \textit{si } x_n \textit{ est impair} \end{cases}
\end{array}\right.$$

2. Une autre version itérative exploite la soustraction. En somme, les étapes de calcul du PGCD sont très proches de celles employées dans la décomposition d'un nombre dans une base.

❶ Proposer une fonction Python `SyracuseRec(a,n)` qui calcule de manière récursive le n[ième] terme d'une suite de Syracuse de premier terme a.

❷ Réaliser un script de programme principal exploitant la fonction `SyracuseRec(a,n)`, et permettant l'affichage des 8 termes qui suivent celui obtenu à un cetain rang lorsqu'il vaut 1. Conclure sur la structure de la suite des nombres suivants.

d - Exponentiation rapide

L'algorithme naïf permettant le calcul de n^p, sans faire appel à la fonction puissance intégrée de Python, consiste à multiplier n par lui-même p fois. Cette manière de faire conduit à une complexité en p. Il est possible d'améliorer sensiblement le calcul en exploitant un algorithme récursif dit **d'exponentiation rapide**.

> Remarque - (1.4) - 2:
>
> Quelques notes sur la complexité :
>
> L'exposant p peut toujours être décomposé en base 2 avec [a] :
>
> $$p = \sum_{i=0}^{d} b_i \times 2^i \qquad \text{avec } b_i = \{0, 1\}$$
>
> on a alors : $n^p = n^{\sum_{i=0}^{d} b_i \times 2^i} = n^{b_0}(n^2)^{b_1}(n^{2^2})^{b_2} \dots (n^{2^d})^{b_d}$
>
> Il faudra d opérations pour calculer les $(n^{2^i})^{a_i}$, puis encore d opérations pour former leur produit. Ainsi, il faut $2d$ opérations au total soit un coût de l'ordre de :
>
> $$\ln p \simeq ln(2^d) \Rightarrow C \sim \frac{\ln p}{\ln 2} = \log_2 p < p$$
>
> a. cf cours révisions 4 "Représentation des nombres en machine"

L'algorithme est le suivant pour tout entier $n > 1$:

- si n est pair alors $x^n = (x^2)^{\frac{n}{2}}$; on calcule alors $y^{\frac{n}{2}}$ avec $y = x^2$
- si n est impair alors $x^n = x \times (x^2)^{\frac{n-1}{2}}$; on calcule alors $y^{\frac{n-1}{2}}$ avec $y = x^2$ que l'on multiplie ensuite par x

En résumé, tout cela devient :

$$puissance(x, n) = \left[\begin{array}{cc} x & \text{si } n = 1 \\ puissance(x^2, n/2) & \text{si } n \text{ est pair} \\ x \times puissance(x^2, n//2) & \text{si } n > 2 \text{ est impair} \end{array} \right.$$

Exercice n°4: *Ecrire le script récursif en Python de la fonction d'exponentiation rapide.*

1.5 Les "dangers" de la récursivité

L'emploi d'une fonction récursive n'est pas toujours judicieux. En particulier, si les opérations d'écriture en pile d'exécution sont trop nombreuses et que certains résultats sont inutilement stockés, le temps de calcul peut s'avérer bien plus long qu'avec une procédure itérative classique.

On propose ci-dessous l'exemple de la **suite de Fibonacci** démarrant à $u_0 = 0$ puis $u_1 = 1$, calculée par une méthode récursive et une méthode itérative. Les deux scripts calculent également les temps d'exécution :

Listing V.8 –

```python
def fiborec(n):
    #algorithme récursif
    if n==1 or n==2 :
        return 1
    return fiborec(n-1)+fiborec(n-2)

for k in range(5):
    print((k+1)*10)
    a=time.clock()
    fiborec((k+1)*10)
    b=time.clock()
    print("temps d'exécution en récursif :", b-a)
```

```
rang : 10
temps d'exécution en récursif : 2.95136093027e-05
rang : 20
temps d'exécution en récursif : 0.00322596581682
rang : 30
temps d'exécution en récursif : 0.449604549715
rang : 40
temps d'exécution en récursif : 53.8201081353
```

Listing V.9 –

```python
import time as t
def fiboiter(n):
    # algorithme itératif
    i=1
    j=1
    k=3
    s=2
    if n==1 or n==2 :
        return 1
    else:
        while k<=n :
            s=i+j
            i=j
            j=s
            k+=1
    return s

for k in range(4):
    print (u"rang:"),(k+1)*10
    a=t.clock()
    fiboiter((k+1)*10)
    b=t.clock()
    print(u"temps d'exécution en itératif :"), b-a
```

```
rang : 10
temps d'exécution en itératif : 4.49120141563e-06
rang : 20
temps d'exécution en itératif : 4.49120141563e-06
rang : 30
temps d'exécution en itératif : 5.77440182009e-06
rang : 40
temps d'exécution en itératif : 7.05760222456e-06
```

2 Les types de récursivité

2.1 Récursivité simple

DÉFINITION - (2.1) - 2:

La récursivité simple correspond au cas le plus classique pour lequel la fonction récursive ne fait qu'un seul appel à elle-même lors de chaque récurrence

On peut citer parmi les algorithme déjà vus : le PGCD récursif, factorielle, suite de Syracuse, exponentiation rapide etc..

2.2 Récursivité multiple

Définition - (2.2) - 3:

La récursivité multiple correspond au cas d'une fonction récursive faisant plus d'un appel à elle-même lors de chaque récurrence.

Remarque - (2.2) - 3: La suite de Fibonacci, traitée plus haut, est déjà un exemple de récursivité multiple.

Un bon exemple de récursivité multiple est le calcul des coefficients binomiaux $C_n^p = \binom{n}{p} = \dfrac{n!}{p!(n-p)!}$ avec $(n,p) \geq (0,0)$

Ainsi on a : $C_0^0 = C_n^n = 1$. En outre on montre facilement la formule de Pascal qui lie les oefficients binomiaux :

$$C_{n+1}^{p+1} = C_n^p + C_n^{p+1}$$

à l'aide du triangle de Pascal, que l'on fabrique en plaçant des 1 sur les extrémités de chaque ligne indicée n puisque $C_0^0 = C_n^n = 1$, les termes inférieurs s'obtenant par sommation des termes adjacents de la ligne supérieure.

$$\begin{array}{c|ccccccccccccc}
n=0 & & & & & & & 1 & & & & & & \\
n=1 & & & & & & 1 & & 1 & & & & & \\
n=2 & & & & & 1 & & 2 & & 1 & & & & \\
n=3 & & & & 1 & & 3 & & 3 & & 1 & & & \\
n=4 & & & 1 & & 4 & & 6 & & 4 & & 1 & & \\
n=5 & & 1 & & 5 & & 10 & & 10 & & 5 & & 1 & \\
n=6 & 1 & & 6 & & 15 & & 20 & & 15 & & 6 & & 1
\end{array}$$

Cette récurrence permet de facilement calculer les coefficients binomiaux C_n^p par une fonction récursive multiple :
On propose l'algorithme suivant :

Listing V.10 –

```python
def C(n,p):
    if (n>p) and (p>0):
        return C(n-1,p-1)+C(n-1,p)
    else: #cas de base
        return 1
```

2.3 Récursivité imbriquée

Définition - (2.3) - 4:

Un fonction récursive dont **l'un des paramètres est un appel à elle-même** *est qualifiée de* **fonction récursive imbriquée**

EXERCICE N°5: *On donne les fonctions d'Ackermann $A(m,n)$ et de Morris $M(m,n)$ définies par :*

$$A(m,n) = \left[\begin{array}{ll} n+1 & si\ m=0\ et\ n\geq 1 \\ A(m-1,1) & si\ n=0\ et\ m\geq 1 \\ A(m-1,A(m,n-1)) & si\ n\geq 1\ et\ m\geq 1 \end{array}\right. \qquad M(m,n) = \left[\begin{array}{ll} 1 & si\ m=0 \\ M(m-1,M(m,n)) & si\ n\geq 1\ et\ m\geq 1 \end{array}\right.$$

Ecrire les scripts récursifs Python de ces deux fonctions.

RÉPONSE :

Listing V.11 – Fonction d'Ackermann

```python
def A(m,n):
    if (m==0) and (n>=1): # cas de base
        return n+1
    elif (n==0) and (m>=1):
        return A(m-1,1)
    else:
        return A(m-1,A(m,n-1))
```

Listing V.12 – Fonction de Morris

```python
def M(m,n):
    if (m==0): # cas de base
        return 1
    else:
        return M(m-1,M(m,n))
```

2.4 Récursivité croisée

DÉFINITION - (2.4) - 5:

Deux fonctions récursives sont dites **mutuellement récursives** *lorsqu'elles s'appellent l'une l'autre. On parle alors* **récursivité croisée**.

- PREMIER EXEMPLE :
  Un exemple très classique de récursivité croisée est l'évaluation de la parité d'un nombre :

Listing V.13 –

```python
def pair(n):
if n==0:
return True
else:
return impair (n-1)
```

Listing V.14 –

```python
def impair(n):
if n==0:
return False
else:
return pair(n-1)
```

QUESTION : Déterminer le résultat de l'appel `pair(2n+1)`.

RÉPONSE :

$2n+1=0$	`pair(2n+1)`						
$(2n+1)-1$		`impair((2n+1)-1)`					
$(2n+1)-2$			`pair((2n+1)-2)`				
$(2n+1)-3$				`pair(2n-1)`			
...	...	...	...	...	...	...	
$2n+1-(2n)=1$						`pair(1)`	
$2n+1-(2n+1)=0$							`impair(0)`

CONCLUSION : le résultat est évidemment le booléen `False`!!!