Leçon d'informatique : récursivité

S. Benlhajlahsen - PCSI$_1$

Sommaire

Extrait du programme :

Thèmes	Exemples d'activité, au choix du professeur et non exigibles des étudiants. Commentaires
Fonctions récursives.	Version récursive d'algorithmes dichotomiques. Fonctions produisant à l'aide de print successifs des figures alphanumériques. Dessins de fractales. Énumération des sous-listes ou des permutations d'une liste. *On évite de se cantonner à des fonctions mathématiques (factorielle, suites récurrentes). On peut montrer le phénomène de dépassement de la taille de la pile.*

I Un exemple simple : la factorielle

On souhaite calculer $n!$ la factorielle d'un entier naturel. On rappelle que $0! = 1! = 1$ et pour $n \geq 2$:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Versions itératives : On souhaite écrire une fonction `fact(n)` qui prend en argument un entier n et qui renvoie $n!$. On peut le faire avec une boucle `for` ou une boucle `while` (voir figures 1 et 2).
On notera que le nombre d'opérations est en $O(n)$.

```
#version itérative
def fact(n):
    res = 1
    if n==0 or n==1:
        return res
    for i in range(2,n+1):
            res *= i
    return res
print(fact(4))
```

FIGURE 1

```
# version itérative
def fact(n):
    res = 1
    if n==0 or n==1:
        return res
    i = 2
    while i <= n:
        res *= i
        i +=1
    return res
print(fact(4))
```

FIGURE 2

```
#version récursive
def fact(n):
    if n==0 or n==1:
        return 1
    else:
        return n*fact(n-1)
print(fact(4))
```

FIGURE 3

Relation de récurrence : Notons $(u_n)_{n\in\mathbb{N}}$ la suite de terme général $u_n = n!$. On remarque alors que :

$$\forall n \geq 1,\ u_n = n \cdot u_{n-1}$$

Ce qui donne une relation de récurrence entre les valeurs successives de la suite.

Fonction récursive La relation de récurrence précédente permet de comprendre la version récursive donnée en figure 3 :

- si $n = 0$ ou $n = 1$, la fonction renvoie 1 ;
- pour $n \geq 2$, `fact(n)` renvoie `n*fact(n-1)` qui renverra `n*(n-1)*fact(n-2)` …jusqu'à arriver à l'appel `fact(1)`.

Arbre d'appels récursifs : Pour comprendre ce qui se passe, on peut représenter l'arbre des appels récursifs (voir figure 4). Lorsque le programme exécute `fact(4)`, il y a deux phases :

- une phase de descente : on appelle successivement `fact(4)` puis `fact(3)` puis `fact(2)` et, enfin, `fact(1)` ;
- une phase de remontée : le programme renvoie les valeurs des appels successifs précédents. On remonte alors jusqu'à l'appel initial et le programme renvoie le résultat.

Notion de pile : Les différents appels représentés dans l'arbre précédent ont une structure de **pile**. Une pile est une structure de données basée sur le principe « dernier arrivé, premier sorti » (en anglais `LIFO` pour *last in, first out*).
Une bonne image est un pile d'assiettes sales **empilées** avant nettoyage. On lavera souvent la dernière assiette (*last in*) en premier (*first out*).

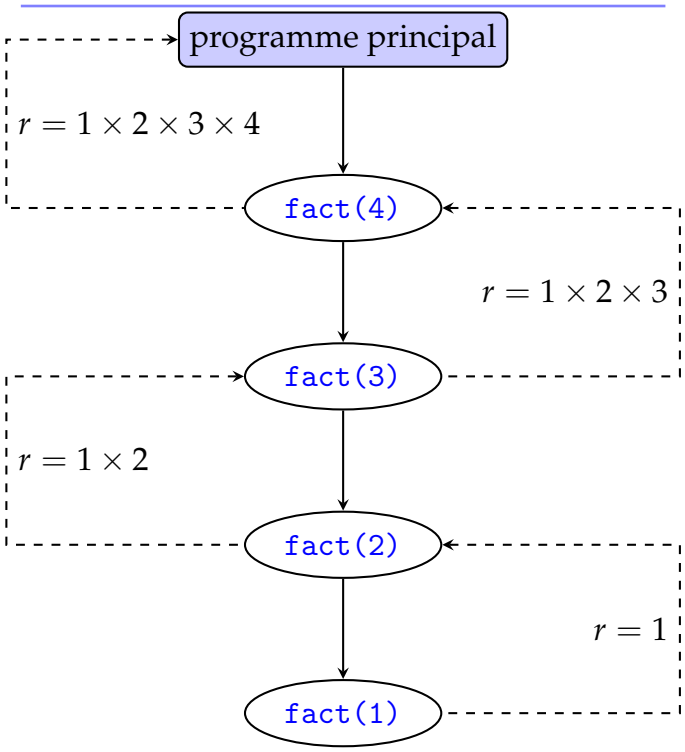

FIGURE 4

II Définitions

À retenir : Une fonction est récursive si, dans sa définition, elle fait référence à elle-même. Il est essentiel de prévoir qu'une fonction récursive se termine ! Pour cela, l'instruction `return` doit être présente deux fois :

- une fois pour la condition d'arrêt ;
- une autre fois pour l'appel récursif.

Suite récurrente Les suites récurrentes telles que $u_n = f(u_{n-1})$ s'expriment naturellement avec un algorithme récursif. Sur la figure 5, on donne une version récursive une implémentation d'une fonction donnant le n-ième terme d'une suite arithmétique de raison r. En effet, $u_n = u_{n-1} + r$.
Ainsi, `suite(5, 1, 2)` renverra `11`.

```
1 #suite arithmétique
2 def suite(n, u0, r):
3     if n==0:       # condition
4         return u0 # d'arrêt
5     else:
6         return suite(n-1, u0, r) + r  # appel récursif
7 print(suite(5, 1, 2))
```

FIGURE 5 – Fonction récursive calculant le n-ième terme d'une suite arithmétique.

Complexité : Le nombre d'opérations réalisées est de l'ordre de $O(n)$. La récursivité est dans cet exemple une façon implicite de réaliser une boucle itérative.

Limitation du nombre d'appels récursifs : En python, le nombre d'appels récursifs est limité **par défaut** à 1000. Le module `sys` et les méthodes `sys.getrecursionlimit` et `sys.setrecursionlimit` permettent respectivement de connaître la limite d'appel récursif et modifier cette limite.

```
import sys
a= sys.getrecursionlimit()
print(a) # renvoie 1000

# on change cette limite
sys.setrecursionlimit(2000)
a = sys.getrecursionlimit()
print(a) # renvoie 2000
```

III Autres exemples

III.A Suite de Fibonacci - récursif

On rappelle que la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ vérifie une relation de récurrence d'ordre 2 avec :

$$F_0 = 0, F_1 = 1 \text{ et } \forall n \geqslant 2,\ F_n = F_{n-1} + F_{n-2}$$

Cette relation de récurrence permet aussi, assez naturellement, d'écrire une fonction récursive (voir figure 6). Pour comprendre ce que donne `fibo_rec(4)`, on peut représenter les appels récursifs sous la forme d'un `arbre binaire`[1].

```
# Fibonacci
def fibo_rec(n):
    if  n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fibo_rec(n-1)+fibo_rec(n-2)
print(fibo_rec(4))
```

FIGURE 6 – Version récursive du calcul du n-ième terme de la suite de Fibonacci.

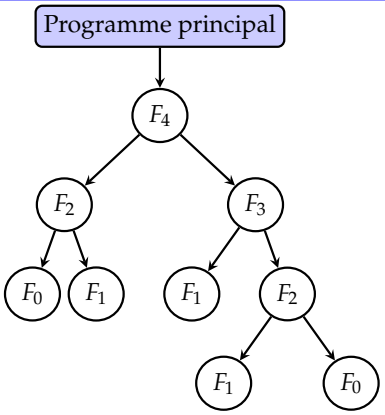

FIGURE 7 – Arbre binaire des appels récursifs.

III.B Exponentiation naïve et exponentiation rapide

On souhaite, de manière récursive, calculer a^n avec n un entier naturel et a un réel.

Exponentiation naïve - versions itérative et récursive : On rappelle que la méthode naïve revient à faire le calcul $a \times a \times a \cdots \times a$ (n fois). Ce calcul peut être fait :

- de manière itérative avec une boucle `for` (voir figure 8) ;
- de manière récursive à l'aide de la relation de récurrence : $a^n = a \cdot a^{n-1}$ (voir figure 9).

Exponentiation rapide - version récursive On peut aussi calculer cette exponentiation de manière plus rapide en constatant que :

- si n est pair, alors n est divisible par 2 et $a^n = \left(a^{n/2}\right)^2$;

1. Voir le chapitre sur les graphes.

```python
# exponentiation naïve itérative
def expo1(a, n):
    res = 1
    for i in range(n):
# on fait n multiplications en tout
        res = res * a
    return res
print(expo1(2, 10)) # renvoie 1024
```

FIGURE 8 – Exponentiation naïve itérative.

```python
# exponentiation naïve - récursive
def expo2(a, n):
    if n==0:
        return 1
    else:
        return a*expo2(a, n-1)
print(expo2(2, 10)) # renvoie 1024
```

FIGURE 9 – Exponentiation naïve récursive.

- si n est impaire, alors $n-1$ est divisible par 2 et $a^n = a \cdot \left(a^{(n-1)/2}\right)^2$.

Cette méthode permet de limiter le nombre de calculs et prendra moins de temps (voir figure 10).

```python
# exponentiation rapide - récursive
def expo3(a, n):
    if n==0:
        return 1
    elif n%2==0:
        return (expo3(a, n//2))**2
    else:
        return a*(expo3(a,(n-1)//2))**2
print(expo3(2, 10)) # renvoie 1024
```

FIGURE 10 – Exponentiation rapide récursive.

Exercice 1 : La suite de Héron est définie par $u_0 = 1$ puis $u_{n+1} = \frac{1}{2} \cdot \left(u_n + \frac{2}{u_n}\right)$. Écrire une fonction récursive `heron(n)` qui retourne le n-ième terme de la suite de Héron. On pourra vérifier que cette suite tend très rapidement vers $\sqrt{2}$.

Exercice 2 : La suite de Syracuse partant de l'entier positif a est définie par $u_0 = a$ puis

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

On notera que tous les termes de la suite sont des entiers. Écrire une fonction récursive `syr_rec(n, a)` qui renvoie le terme d'indice n de la suite de Syracuse. On pourra vérifier que `print(syr_rec(6, 14))` renvoie 52.

III.C Tours de Hanoï

Le problème des tours de Hanoï est un jeu de réflexion imaginé par le mathématicien français Édouard Lucas consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire » tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

Au départ les disques sont placés en respectant la seconde règle sur la tour de départ (voir figure 11) et on veut arriver à la figure 12.

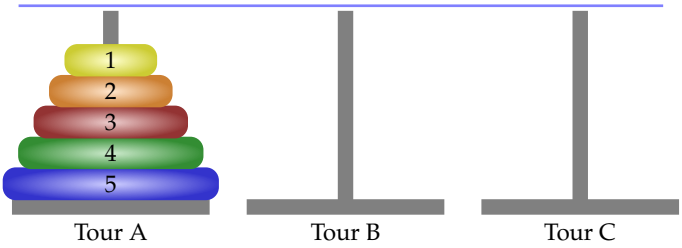

FIGURE 11

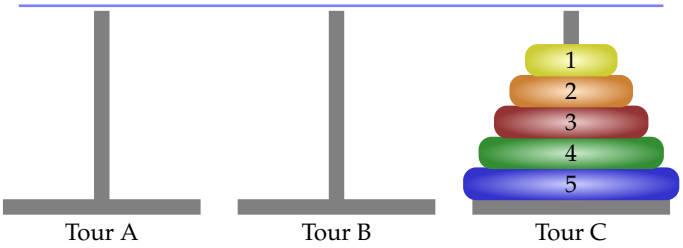

FIGURE 12

Une position intermédiaire possible serait celle de la figure 13.

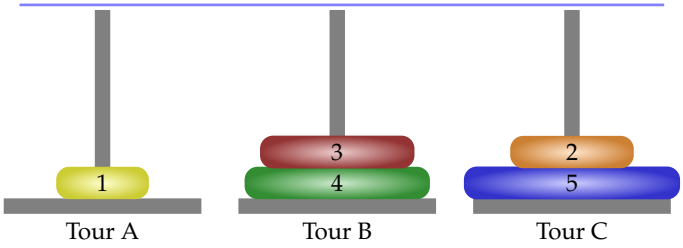

FIGURE 13

Dans cette dernière position il y a 3 déplacements possibles :

- déplacer le disque 1 au-dessus du disque 3,
- déplacer le disque 1 au-dessus du disque 2,
- déplacer le disque 2 au-dessus du disque 3.

La résolution est en fait simple à énoncer en renonçant à écrire la stratégie de manière précise : pour déplacer n disques de la tour A la tour C il suffit :

- de ne rien faire si $n = 0$;
- de déplacer les $n-1$ disques supérieurs de la tour A vers la tour B, puis de déplacer le disque restant vers la tour C : puis enfin de déplacer les $n-1$ disques supérieurs de la tour B vers la tour C.

```
1 def hanoi(n, ch1, ch2, ch3):
2     if n != 0:
3         hanoi(n-1, ch1, ch3, ch2)
4         print("Déplacer le disque supérieur
              de {} vers {}".format(ch1,ch3))
5         hanoi(n-1, ch2, ch1, ch3)
6 hanoi(3, "A", "B", "C")
```

FIGURE 14

```
1 Déplacer le disque supérieur de A vers C
2 Déplacer le disque supérieur de A vers B
3 Déplacer le disque supérieur de C vers B
4 Déplacer le disque supérieur de A vers C
5 Déplacer le disque supérieur de B vers A
6 Déplacer le disque supérieur de B vers C
7 Déplacer le disque supérieur de A vers C
```

FIGURE 15

Une implémentation est donnée en figure 14 et on le teste dans le cas où il n'y a que 3 disques (voir 15).

III.D Dessins

Exercice 3 : On considère les programmes de la figure 17. Prévoir ce qu'affiche l'interpréteur python.

```
1 def triangle1(n):
2     print(n*"X")
3     if n>0:
4         triangle1(n-1)
5 triangle1(4)
```

```
1 def triangle2(n):
2     if n>0:
3         triangle2(n-1)
4     print(n*"X")
5 triangle2(4)
```

FIGURE 16

Dessin de « fractale » Le triangle de Sierpinski (voir figure 17) est une figure fractale, obtenue à partir d'un triangle équilatéral par le mécanisme suivant :

- Relier les milieux des côtés, de façon à créer quatre zones triangulaires.
- Enlever la zone du milieu.
- Recommencer dans chacune des trois zones restantes et poursuivre indéfiniment.

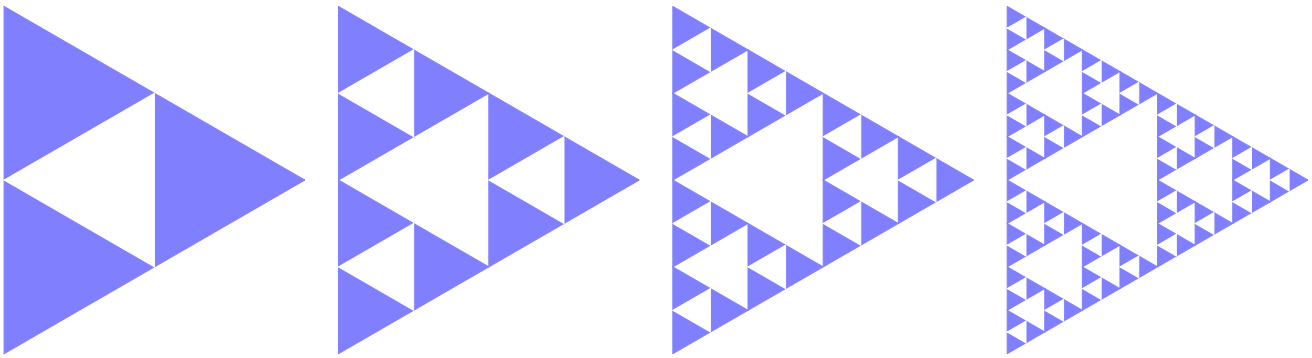

FIGURE 17