TP n°6 : introduction à la récursivité

Introduction

La récursivité est une méthode de programmation qui pourrait avoir la morale suivante : on résout quelque chose de compliqué en fonction d'une version un peu moins compliquée de la même chose jusqu'à arriver à quelque chose de très facile, puis on revient en arrière pour résoudre le problème initial [1].

Un algorithme récursif est un algorithme qui fait appel à lui-même dans le corps de sa propre définition, il est constitué de deux éléments remarquables :

- **le cas de base** (correspond à la version facile du problème)
- **la formule de propagation ou d'hérédité** (permet de passer à une version plus simple)

Par exemple l'algorithme d'Euclide pour calculer le pgcd de deux entiers peut s'écrire de manière récursive :

- le cas de base est lorsque $b = 0$, dans ce cas $\text{pgcd}(a, 0) = a$.
- la formule de propagation est $\text{pgcd}(a, b) = \text{pgcd}(b, r)$ où r est le reste de la division euclidienne de a par b. En effet, le problème «trouver $\text{pgcd}(b, r)$» est bien «une version plus simple» du problème «trouver $\text{pgcd}(a, b)$» puisque $0 \leqslant r < b$.

Nous appelons `eucliderec` notre algorithme d'Euclide écrit en récursif.

algorithme : `eucliderec`
Entrées : $a, b \in \mathbb{N}$
Résultat : $\text{pgcd}(a, b)$
  Si $b = 0$ alors
    Renvoyer a
  Sinon
    Renvoyer `eucliderec(b,r)`
où r est le reste de la divison de a par b.

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

1. Pour comprendre la récursivité, il faut comprendre la récursivité :-) :-)

1 Premiers exemples

Exercice 1 On considère la fonction

```
def recursive(x) :
    if x < 1:
        print("je ne m'appelle pas pour x=",x)
        return 1
    else :
        print("je m'appelle pour x=",x)
        return recursive (x/2) + 1
```

1. Sans exécuter le code dans un premier temps deviner la valeur de `recursive(x)` pour $x \in \{4, 8, 10\}$.
2. Vérifier en exécutant le code. Que renvoie la fonction `recursive(x)` ?

Exercice 2 (Un exemple) On considère la fonction suivante :

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

1. Que calcule la fonction suivante ?
2. Quels sont les avantages ou les inconvénients de cette version récursive, par rapport à la version itérative ?

Exercice 3 (Dessins récursifs) On considère les trois fonctions suivantes :

```
def triangle1():
    print('*')
```

```
def triangle2():
    print(2*'*')
    triangle1()
```

```
def triangle3():
    print(3*'*')
    triangle2()
```

1. Que se passe-t-il si l'on exécute `triangle3()` ? Réfléchir puis vérifier avec votre machine.
2. Écrire deux fonctions récursives `triangle(n)` et `trianglebis(n)` qui prennent en argument un entier naturel $n \in \mathbb{N}$ et affichent repectivement le dessin de gauche et celui de droite) avec n lignes. Par exemple, `triangle(5)` et `trianglebis(5)` afficheront :

```
*****
****
***
**
*
```

```
*
**
***
****
*****
```

Exercice 4 (Verlan) Écrire une fonction récursive `verlan(mot:str)-> str` qui prend en argument un mot et renvoie le mot écrit à l'envers. Par exemple si `mot` est la chaîne `'volley'`, `verlan(mot)` renverra `'yellov'`. Attention, on pourra utiliser le slicing et notamment l'instruction `mot[ 1: ]` .

2 Avec des entiers...

Exercice 5 (Coefficients binomiaux) On sait que pour $n \geqslant p \geqslant 1$, on a $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$.

1. Calculer $\binom{8}{4}$ avec cette méthode.
2. Compléter le code suivant pour écrire une fonction récursive `binom_rec(n, p)` qui renvoie le coefficient binomial $\binom{n}{p}$.

```
def binomial(n, p):
    # cas particulier
    if not 0 <= p <= n:
        return .....
    # cas de base
    if p == 0:
        return ....
    # formule de propagation
    else:
        return ........
```

Exercice 6 (Exponentiation) Dans cet exercice, on étudie la complexité de deux algorithmes prenant en argument deux entiers x et $n \geqslant 0$ et renvoyant l'entier x^n. On s'autorise le produit des entiers mais on s'interdit bien sûr l'utilisation de la fonction puissance, on cherche ainsi ce qui se cache lorsqu'on tape `x**n`.

On peut calculer x^n de la façon suivante : $x^n = x \times x \times \cdots \times x$.

1. Écrire une fonction `expo_naif(x, n)` non récursive, reprenant cette méthode prenant en argument deux entiers x et n et renvoyant x^n.
2. Écrire une version récursive de cet algorithme naïf. On pourra appeler `exponaif_rec(x, n)` cette fonction.
3. Déterminer le nombre de multiplications effectuées par `expo_naif(x, n)` (resp. `exponaif_rec(x, n)`), en déduire sa complexité.

Nous allons voir maintenant une autre méthode appelée «exponentiation rapide».

Elle repose sur le principe suivant où l'on décompose l'entier n en base 2 :

$$\begin{cases} x^0 &= 1 & \\ x^n &= (x^2)^{\frac{n}{2}} & \text{, si } n \text{ est pair et } n > 0 \\ x^n &= x \times (x^2)^{\frac{n-1}{2}} & \text{, si } n \text{ est impair} \end{cases}$$

4. Soit x un réel. Appliquer l'algorithme d'exponentiation rapide pour calculer x^8 puis x^{14}. Indiquer dans les deux exemples le nombre de multiplications effectuées puis comparer avec la méthode «naïve».

5. Écrire une fonction récursive `exporap(x, n)` qui prend en argument x un nombre flottant, n un entier naturel et renvoie le nombre x^n selon l'algorithme d'exponentiation rapide.

6. On exécute `exporap(x, n)`. On note n_k la valeur de l'exposant n après k appels récursifs de exporap. On convient que $n_0 = n$. Comparer n_{k+1} et $\frac{n_k}{2}$, en déduire que $n_k < 1$ dès que $k \geqslant \log_2(n)$.

On en déduit que le nombre de multiplications qu'effectue `exporap(x,n)` est inférieur ou égal à $2(\log_2(n) + 1)$. Il s'agit donc d'une complexité logarithmique.

Exercice 7 (Somme des chiffres) Écrire une fonction récursive `somme_chiffre(n)` prenant en argument $n \in \mathbb{N}$ et qui renvoie la somme de ses chiffres.

Par exemple `somme_chiffre(45)` renvoie $4 + 5 = 9$.

3 Le côté obscur de la récursivité

Exercice 8 (Le côté obscur de la récursivité) On considère la fonction suivante :

```
def u(n):
    if n == 0:
        return 1
    else:
        return u(n-1) + 1/u(n-1)
```

1. Que calcule la fonction récursive `u` ?

2. On note a_n le nombre d'appels récursifs à la fonction `u` lorsque l'on exécute `u(n)`. Exprimer a_n en fonction de a_{n-1}. En déduire la complexité de la fonction `u`.

3. Modifier le code précédent de façon à obtenir une complexité linéaire.

Exercice 9 (Fibonacci, le côté obscur de la récursivité, mais un espoir aussi) On rappelle que la suite de Fibonacci (F_n) est définie par $F_0 = 0, F_1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geqslant 2$.

1. Écrire une fonction récursive `fiborec(n)` qui renvoie le nombre F_n.

2. Tester votre fonction pour des petits entiers puis pour $n = 50$. Commenter. Dessiner l'arbre des appels pour $n = 4$.

On note a_n le nombre d'additions qu'effectue `fiborec(n)`. On a donc $a_0 = a_1 = 0$.

3. Soit $n \geqslant 2$. Exprimer a_n en fonction de a_{n-1} et a_{n-2}.

4. En déduire par récurrence que $n \geqslant 2$, on a $a_n \geqslant F_n$.

On a donc (cours de Mathématiques) pour $n \geqslant 2$, $a_n \geqslant F_n \sim \frac{\phi^n}{\sqrt{5}}$. C'est une «catastrophe algorithmique». Il s'agit d'une complexité exponentielle.

Remarque : on peut réparer cette fonction récursive en mémoïzant les résultats à l'aide d'un dictionnaire. C'est le principe de programmation dynamique.

Exercice 10 (Complexité spatiale cachée) Voici une fonction récursive qui calcule la somme des éléments d'une liste.

```
def somme(L : [float]) -> float:
    if len(L) == 0:
        return 0
    else:
        return L[0] + somme(L[1 : ])
```

1. Quelle est la complexité temporelle de cette fonction ?

Lorsque l'on appelle la fonction avec L de longueur n, on recopie une nouvelle liste $L[1:]$ qui est de longueur $n-1$. Au total on recopie donc $(n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2}$ éléments. Il s'agit donc d'une complexité spatiale quadratique :-(

Pour contrer ce problème, on utilise l'astuce suivante : on rajoute à la fonction un paramètre qui est l'indice jusqu'au quel on somme.

2. Écrire une fonction récursive `somme_aux(L : [float], ind:int) -> float` qui calcule la somme des éléments de la liste L jusqu'à l'indice *ind* inclus, c'est-à-dire $L[0]+L[1]+...+L[ind]$. Si par exemple, $L = [20, 40, 50, 80]$, `somme_aux(L, 2)` renverra $20+40+50 = 110$.
3. En déduire une nouvelle fonction utilisant la précédente qui calcule la somme des éléments d'une liste mais avec une complexité spatiale linéaire.

4 Dessiner des fractales avec la tortue

Pour cette section, on utilise le module `turtle`, que l'on importe en exécutant `from turtle import *`. Si on utilise Pyzo, il est préférable d'utiliser TK (Tkinter) comme GUI (Graphical User Interface). Pour cela aller dans l'onglet Shell, puis Edit Shell Configurations et choisir TK comme GUI. Relancer alors votre shell.

L'instruction `forward(50)` permet d'avancer de 50 pixels et `right(30)` fait tourner la tortue de 30 degrés vers la droite. L'instruction `reset()` perùet de réinitialiser le dessin.

Exercice 11 (Fractales) L'objectif est de dessiner la célèbre fractale appelée flocon de Von Koch. On peut la créer à partir d'un segment de droite, en modifiant récursivement chaque segment de droite de la façon suivante :

- On divise le segment de droite en trois segments de longueurs égales.
- On construit un triangle équilatéral ayant pour base le segment médian de la première étape.

FIGURE 1 – Les trois premières étapes du flocon de Von Koch

- On supprime le segment de droite qui était la base du triangle de la deuxième étape.

On répète alors ces étapes...

1. Écrire une fonction `koch1(l)` qui dessine la première étape du flocon où l est la longueur du segment.
2. A l'aide de la fonction précédente, écrire une fonction `koch2(l)` qui dessine la deuxième étape du flocon.
3. A l'aide de la fonction précédente, écrire une fonction `koch3(l)` qui dessine la troisième étape du flocon.
4. Généralisation : écrire une fonction récursive `koch(l, n)` qui dessine la n-ième étape du flocon de Von Koch.