Informatique — MPSI/PCSI, 2023–2024

TP n° 7 – Récursivité

1 Fonction récursive

On rappelle que la factorielle d'un entier naturel n, notée $n!$ est définie par

$$\begin{cases} 0! &= 1 \\ n! &= 1 \times 2 \times \ldots \times n \text{ si } n > 0 \end{cases}$$

On peut écrire une fonction Python prenant n en paramètre et calculant $n!$ de la manière suivante.

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

Une manière alternative de procéder est de définir $n!$ par récurrence de la manière suivante.

$$\begin{cases} 0! &= 1 \\ \text{Pourtout } n > 0,\ n! &= n \times (n-1)! \end{cases}$$

Cela peut se traduire par la fonction Python suivante.

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

La première implémentation est appelée **itérative**. La seconde est une implémentation **récursive**, dans laquelle la fonction s'appelle elle-même.

De manière générale, une fonction récursive est une fonction qui retourne directement le résultat dans un cas élémentaire, et s'appelle elle-même sur une taille des entrées plus petite sinon. Cette méthode permet souvent d'écrire du code de façon particulièrement concise et élégante.

Il est à noter que pour garantir l'arrêt d'une fonction récursive, l'appel de la fonction par elle-même doit se faire à l'intérieur d'une instruction conditionnelle. Il est de plus nécessaire de vérifier que la fonction s'arrête bien quelle que soient les valeurs des paramètres d'entrée. C'est le cas par exemple pour la fonction `factorielle` définie ci-dessus : en effet, le paramètre passé en appel diminue de 1 à chaque fois, jusqu'à atteindre la valeur 0, pour laquelle la fonction cesse de s'appeler.

Un appel d'une fonction récursive ne se termine pas avant que tous les sous-problèmes soient résolus. Pendant tout ce temps, les paramètres et variable locales de cet appel sont stockés dans une pile d'appels (en anglais *stack*), ce qui peut être gourmand en place mémoire. La taille maximale de la pile d'appel est bornée et son ordre de grandeur est de 1000 (la valeur exacte dépend de votre plate-forme informatique). Il est possible (avec prudence) de modifier cette valeur par

```
import sys
sys.setrecursionlimit(n)
```

Exercice 7.1
On rappelle l'algorithme d'Euclide permettant de déterminer le pgcd de deux entiers naturel a et b : on définition une suite de restes r_n de la manière suivante

$$\begin{cases} r_0 = a \\ r_1 = b \\ \text{Pour tout entier } n, \text{ si } r_n > 0, r_{n+1} = r_{n-1} \% r_n \end{cases}$$

Le pgcd de a et b est alors le dernier reste non nul de la suite.

Écrire une version itérative de l'algorithme à l'aide d'une boucle `while`, puis une version récursive.

Exercice 7.2

1. Écrire une fonction Python récursive prenant en entrée un entier strictement positif n et affichant n lignes d'étoiles sur le modèle suivant.

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

2. Écrire une fonction Python récursive prenant en entrée un entier strictement positif n et affichant n lignes d'étoiles sur le modèle suivant.

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

Exercice 7.3

On désire écrire deux fonctions permettant d'obtenir le quotient et le reste de deux entiers naturels **sans utiliser les opérateurs Python // et %**.

1. Soit a et b deux entiers naturels, avec $b \neq 0$. Si $r(a, b)$ désigne le reste de la division euclidienne de a par b, on peut remarquer que $r(a, b) = a$ si $a < b$ et $r(a, b) = r(a - b, b)$ sinon.
   Écrire une fonction prenant en paramètres deux entiers strictement positifs a et b et calculant $r(a, b)$ de manière récursive.
2. On note désormais $q(a, b)$ le quotient de la division euclidienne de a par b. On peut remarquer que $q(a, b) = 0$ si $a < b$ et $q(a, b) = q(a - b, b) + 1$ sinon.
   Écrire une fonction prenant en paramètres deux entiers naturels a et b ($b > 0$) et calculant $q(a, b)$ de manière récursive.

Exercice 7.4

On désire écrire une fonction prenant en entrée une chaîne de caractères deux à deux distincts non vide et retournant une liste de toutes les permutations possibles de la chaîne. (Par exemple, une liste des permutations possibles de 'abc' est ['abc', 'acb', 'bac', 'bca', 'cab', cba'].) On peut utiliser pour cela l'algorithme suivant :

1. Si la chaîne est de longueur 1, elle admet une unique permutation.
2. Sinon, on procède de la manière ci-dessous.
   (a) Isoler le premier caractère de la chaîne
   (b) Appliquer récursivement l'algorithme pour générer l'ensemble des permutations de la sous-chaîne de caractères restante.
   (c) Pour chacun des éléments de la liste obtenue, générer toutes les permutations de la chaîne initiale obtenues en insérant le premier caractère de la chaîne à un emplacement quelconque. (Par exemple, si le premier caractère est 'a' et la permutation de la sous-chaîne de deux caractères restants est 'bc', on génère les permutations 'abc', 'bac' et 'bca'.)

Écrire une fonction récursive Python répondant au problème posé en utilisant l'algorithme ci-dessus.

2 Généralisation

Dans notre premier exemple de fonctions récursives, une fonction dépendant d'un paramètre entier n s'appelait elle-même avec le paramètre $n - 1$ lorsque n était strictement positif. On peut généraliser ce principe à des cas où la fonction s'appelle elle-même à un indice compris entre 0 et $n - 1$. (Cette idée est analogue à la définition par récurrences forte en mathématiques.)

Dans tous ce qui suit, pour tout réel x, la partie entière de x sera notée $\lfloor x \rfloor$. La méthode d'exponentiation rapide est fondée sur les propriétés suivante : pour tout réel x et pour tout entier naturel n,

$$\begin{cases} x^0 &= 1 \\ x^n &= \left(x^{\left\lfloor \frac{n}{2} \right\rfloor}\right)^2 & \text{si } n \text{ pair} \\ x^n &= x \times \left(x^{\left\lfloor \frac{n}{2} \right\rfloor}\right)^2 & \text{si } n \text{ impair} \end{cases}$$

La procédure d'exponentiation rapide peut donc s'implémenter de la façon suivante.

```
def exp_rapide(x,n):
    if n==0:
        return(1)
    else:
        m=n//2
        y=exp_rapide(x,m)
        if n%2==0:
          return(y*y)
        else:
          return(y*y*x)
```

Exercice 7.5
Appliquer « à la main » la fonction ci-dessus pour $n = 13$. Combien d'appel à la fonction sont-ils réalisés et avec quels paramètres ? Combien de multiplication sont-elles effectuées au total ? Comparer avec le nombre de multiplications effectuées avec la définition usuelle $x^{13} = x \times x \times \ldots \times x$.

Exercice 7.6
Étant donné un nombre réel x et une liste (non vide) de n entiers **classées dans l'ordre croissant** $[x_0, x_1, \ldots, x_{n-1}]$ on désire déterminer si x appartient à la liste. Pour cela, on utilise l'algorithme de recherche dichotomique suivant.

— Si la liste est de longueur 0, x n'appartient pas à la liste.
— Sinon, on note n la longueur de la liste et on pose $m = \lfloor \frac{n}{2} \rfloor$.
  - si $x_m = x$, x appartient à la liste et on s'arrête.
  - si $x_m > x$, on itère l'algorithme en l'appliquant à x et à la liste $[x_0, x_1, \ldots, x_{m-1}]$
  - sinon, on itère l'algorithme en l'appliquant à x et à la liste $[x_{m+1}, \ldots, x_{n-1}]$

Implémenter l'algorithme en Python au moyen d'une fonction récursive.

— Complément —
pour les plus rapides

3 Tortue et fractales

Le module `turtle` est un ensemble de fonctions permettant d'animer une tortue qui se déplace sur un écran en avançant, reculant ou tournant et peut dessiner en laissant une trace dernière elle. (Ce module est inspiré du langage Logo, utilisé pour l'initiation à la programmation dans les années 60.)

Pour importer ce module, il faut ajouter au début de votre fichier la commande

```
import turtle
```

On peut utiliser par la suite toutes les fonctions du module en les faisant précéder du nom du module suivi d'un point. Par exemple, pour utiliser la fonction `clear()`, qui efface tous les dessins, on tape `turtle.clear()`.

Par défaut, les dessins se font dans une fenêtre de $950 = 2 \times 475$ pixels de large et $800 = 2 \times 400$ pixels de haut, muni d'un repère orthonormé tel que le point $(0; 0)$ soit au centre de l'écran. Au départ, la tortue est en $(0, 0)$, orientée à 0 degré par rapport à l'axe des abscisses.

Les principales fonctions disponibles sont listées ci-dessous.

`reset()`	Efface l'écran
`goto(x, y)`	Déplace la tortue au point de coordonnées (x, y)
`forward(distance)`	Avance de distance
`backward(distance)`	Recule de distance
`up()`	Relève le crayon (pour pouvoir avancer sans dessiner)
`down()`	Abaisse le crayon (pour recommencer à dessiner)
`color(couleur)`	Change la couleur du tracé (`'red'`, `'blue'`, etc.)
`left(angle)`	Tourne à gauche de angle (exprimé en degrés)
`right(angle)`	Tourne à droite de angle
`width(épaisseur)`	Choisit l'épaisseur du tracé
`write(texte)`	Écrit la chaîne de caractères texte

Amusez-vous par exemple à exécuter le code suivant.

```
turtle.reset()
for i in range(4):
    turtle.forward(150)
    turtle.left(90)
```

Exercice 7.7
Le flocon de Von Koch est une courbe construite par étapes de la manière suivante.

— On part d'un triangle équilatéral.
— À chaque étape, on transforme tous les segments de la figure de la manière suivante.

1. On divise le segment de base en trois segments de longueurs égales.
2. On construit un triangle équilatéral ayant pour base le segment médian de la première étape.
3. On supprime le segment de droite qui était la base du triangle de la deuxième étape.

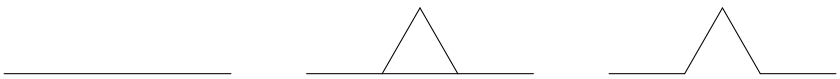

FIGURE 1 – Transformation d'un segment.

Les quatre premières itérations nous donne la liste des figures suivantes.

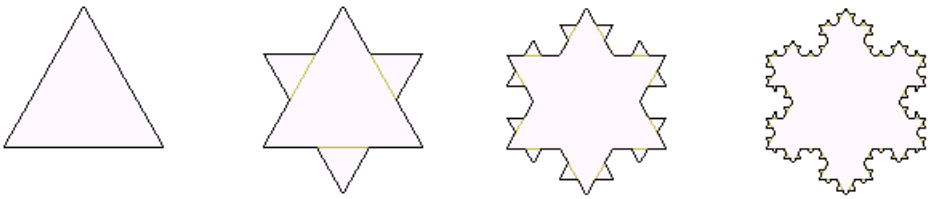

FIGURE 2 – Les quatre premières étapes de la construction du flocon.

Écrire une fonction qui, prenant en entrée un entier naturel n, dessine le résultat obtenu au bout de n itérations.

Le flocon de Von Koch désigne la courbe limite obtenue en itérant « à l'infini » le procédé ci-dessus. Chacune des figures obtenues à partir d'un des côtés du triangle initial possède une propriété remarquable : si vous prenez la partie de la figure correspondant au premier tiers du segment, et que vous la grossissez trois fois, vous obtenez à nouveau la figure complète.

Les objets de ce type sont appelés *fractales*. Des exemples de fractales sont connues depuis longtemps. Ces objets ont été étudiés notamment par le mathématicien français Benoît Mandelbrot (1924-2010).

4 Appels récursifs multiples

On va étudier maintenant le cas où une fonction peut faire plusieurs appels à elle-même.

Commençons par un exemple bien connu. La suite de Fibonacci est définie par

$$\begin{cases} u_0 = u_1 & = & 1 \\ \text{Pour tout } n \geq 2, u_n & = & u_{n-1} + u_{n-2} \end{cases}$$

Le calcul des termes de la suite de Fibonacci peut se faire de manière itérative.

```
def fibo_it(n):
    if n<=1:
        return(1)
    else:
        u=1
        v=1
        for i in range(n):
            w=u+v
            u=v
            v=w
        return(w)
```

Il est également possible d'implémenter une version récursive du calcul des termes.

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

Il est important de préciser que cette façon de procéder est très mauvaise en pratique, car elle implique de refaire plusieurs fois les mêmes calculs. Supposons par exemple que vous appeliez `fibo_rec(5)}` : la fonction va appeler `fibo_rec(3)` et `fibo_rec(4)`, mais `fibo_rec(4)` va à son tout appeler `fibo_rec(3)`, et ainsi de suite jusqu'à effectuer de nombreux appels redondants.

Vous verrez en seconde année un procédé appelé *memoïsation*, consistant à conserver en mémoire les calculs déjà effectués, permettant d'éviter ce problème. Pour l'instant, nous nous contenterons de donner quelques exemples classiques de récursion de ce type.

Exercice 7.8
Soit n un entier positif. On cherche à déterminer de combien de manière il est possible de découper une barre de n mètre en morceaux de 2 ou 3 mètres *en tenant compte de l'ordre*. Une barre de 8 m pour par exemple se découper de quatre manières différentes : $8 = 2+2+2+2 = 2+3+3 = 3+2+3 = 3+3+2$.

Soit $d(n)$ le nombre de découpes possibles pour un morceau de longueur n. On a $d(0) = d(1) = 0$ et $d(2) = 1$. Si $n \geq 3$, on peut séparer la situation en deux cas : soit le premier morceau est de taille 2, et il y a $d(n-2)$ façon de découper le morceau restant, soit il est de taille 3, et il y a $d(n-3)$ façon de découper le morceau restant. On en déduit que pour tout $n \geq 3$, $d(n) = d(n-2) + d(n-3)$.

Écrire une fonction récursive prenant en entrée un entier n et retournant $d(n)$.

Exercice 7.9
On rappelle que les coefficients binomiaux vérifient les propriétés suivantes.

— Pour tout entier naturel n, $\left(\begin{array}{c} 0 \\ n \end{array} \right) = \left(\begin{array}{c} n \\ n \end{array} \right) = 1$.

— Pour tout entier naturel $n \geq 1$ et pour tout entier k tel que $1 \leq k \leq n-1$, $\left(\begin{array}{c} k \\ n \end{array} \right) = \left(\begin{array}{c} k \\ n-1 \end{array} \right) + \left(\begin{array}{c} k-1 \\ n-1 \end{array} \right)$.

En utilisant ces deux propriétés, écrire une fonction prenant en paramètres deux entiers n et k et calculant le coefficient binomial $\left(\begin{array}{c} k \\ n \end{array} \right)$.

* *
*