Informatique — MPSI/PCSI, 2024–2025

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 défint une suite (finie) d'entiers naturels u par $u_0 = a$, $u_1 = b$, et, tant que $u_{n+1} \neq 0$, u_{n+2} est le reste de la division euclidienne de u_n par u_{n+1}. Le pgcd de a et b est alors le dernier terme u_n non nul.

Écrire une version itérative de l'algorithme à l'aide de `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. 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 le reste $r(a, b)$ de la division euclidienne de a par b de manière récursive.
2. Écrire une fonction prenant en paramètres deux entiers naturels a et b ($b > 0$) et calculant le quotient de la division euclidienne de a par b de manière récursive.
   (On pourra reprendre l'idée de la question précédente et introduire un troisième paramètres décomptant le nombre de soustractions déjà effectuées.)

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 tous les cas précédents, 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 $\lfloor x \rfloor$. Rappelons les propriétés suivantes, déjà utilisées cette années au moment de la description de la méthode d'exponentiation rapide : pour tout réel x et pour tout entier naturel n,

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

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 donnée 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 1, on teste si l'unique élément de la liste est égal à x.
— Sinon, on note n la longueur de la liste et on pose $m = \lfloor \frac{n}{2} \rfloor$.

· si $x_{m-1} = x$, x appartient à la liste et on s'arrête.
· si $x_{m-1} > x$, 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 $[x_{m-1}, x_m, \ldots, x_{n-1}]$

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

3 Similitudes et fractales

3.1 Similitudes

Une similitude directe du plan est une transformation multipliant toutes les longueurs par un réel k strictement positif fixé et conservant les angles orientés. On peut montrer (nous ne le ferons pas ici) que si u est une transformation du plan de ce type, seuls deux cas sont possibles :

1. u est une translation
2. u admet point fixe I et elle est la composée d'une homothétie de centre I et de rapport k et d'une rotation de centre I et d'angle θ (voir figure 1). On intéressera par la suite à ce second cas.

   On peut noter que si $k = 1$, u est une rotation, et que si $\theta = 0$, u est une homothétie.

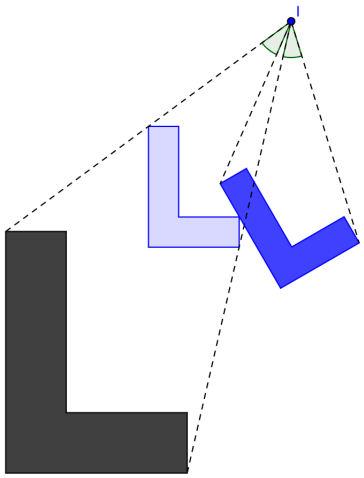

FIGURE 1 – La similitude est la composée d'une homothétie et d'une rotation toutes deux de centre I.

Dans tout ce qui suit, un point sera codé en Python par une liste de deux nombres flottants représentants ses coordonnées.

Exercice 7.7

1. On donne dans le fichier `TP07_rotation.py` le code d'une fonction python prenant en argument les coordonnées de deux points I et M et un angle t et retournant les coordonnées du point image de M par la rotation de centre I et d'angle t.

   Écrire une fonction `simil(I,M,k,theta)` utilisant la fonction précédante et prenant en argument la liste des coordonnées d'un point I et celle d'un point M, deux nombres flottants k et t et retournant l'image du point M par la similitude de centre I, de rapport k et d'angle t.
   (On pourra donner les valeurs par défaut 1 à k et 0 à t.)
2. En déduire une fonction `simil_poly(I,L,k,t)` prenant en entrée un point I, une liste de points L , deux nombres flottants k et *theta* et retournant la liste des images des éléments de L par la similitude de centre I, de rapport k et d'angle *theta*.

3.2 Flocon de von Koch

La flocon de Von Koch est une courbe construite par étapes de la manière suivante.

— On part d'un triangle équilatéral.
— A chaque étape, on transforme tous les segment de la figure de la manière suivante (voir figure 2) :

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.

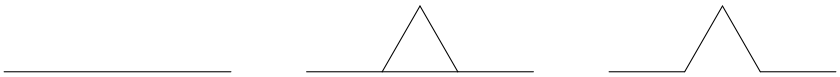

FIGURE 2 – Transformation d'un segment

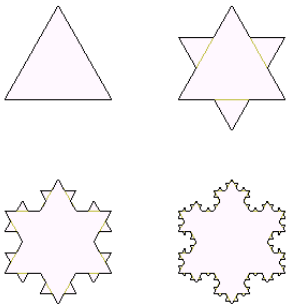

FIGURE 3 – Les quatre premières étapes de la construction du flocon

Le flocon de Von Koch désigne la courbe limite obtenue en itérant "à l'infini" le procédé ci-dessus (voir figue 3) . Chacune des figures obtenues à partir d'un des côtés du triangle initial possède une propriété remarquable : si vous prenez le premier tiers du côté et que vous la grossissez trois fois, vous obtenez à nouveau un côté complet.

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

Exercice 7.8

1. On va commencer par appliquer le processus à un segment $[AB]$. Ecrire une fonction récursive `cote_von_koch_cote(L,n)` prenant en entrée une liste de deux éléments L contenant les couples de coordonnées de deux points A et B et un entier n et retournant la liste des couples de coordonnées des sommets de la ligne polygonale obtenue en appliquant n fois à $[AB]$ la construction décrite ci-dessus.
2. Écrire une fonction `von_koch(n)` prenant en entrée un entier n et représentant la construction le flocon obtenu à l'étape n à l'aide du module `mathplotlib.pyplot`.

3.3 Courbe du dragon

La courbe du dragon est une fractale étudiée la première fois dans les années 60 par John Heighway, Bruce Banks, et William Harter. Elle peut être construite en pliant plusieurs fois une feuille de papier dans le sens de la longueur, puis en la dépliant de façon à ce que les plis forment un angle droit (la figure 4 montre les résultats obtenus après un, deux et trois plis [1]). La courbe du dragon est la courbe obtenue en itérant ce procédé à l'infini.

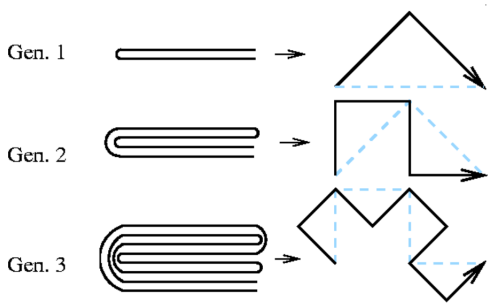

FIGURE 4 – Construction de la courbe du dragon par pliages

Soit n un entier naturel et A et B deux points du plan. La courbe reliant les points A et B obtenue après n plis peut être définie de la manière suivante :

1. Si $n = 0$, la courbe est le segment $[AB]$
2. Si $n > 0$, la courbe est obtenue en prenant l'image de la courbe obtenue à l'ordre $n-1$ par la similitude de centre A, de rapport $k = \frac{1}{\sqrt{2}}$ et d'angle $\frac{\pi}{4}$, puis l'image de cette première courbe par une rotation de centre le dernier sommet de la première image et d'angle $\frac{\pi}{2}$.

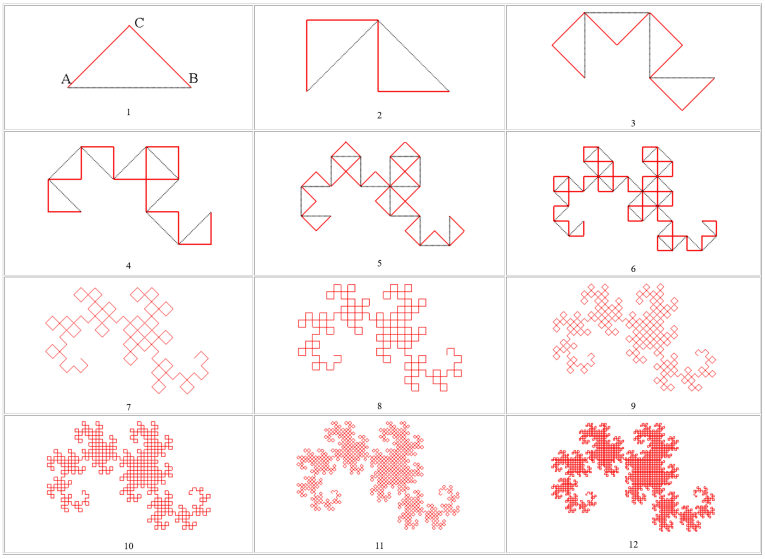

FIGURE 5 – Construction de la courbe du dragon par images par des similitudes successives

Exercice 7.9

1. Écrire une fonction récursive `points_dragon(L,n)` prenant en argument une liste L contenant les coordonnées des points A et B et un entier naturel n et retournant la liste des coordonnées des sommets de la courbe obtenue après n itérations et joignant les points A et B.
2. Écrire une fonction `dragon(n)` traçant la courbe du dragon à l'ordre n. On pourra prendre pour points de départ $A(0;0)$ et $B(1;0)$.

1. Source : https ://mathcurve.com

4 Appels récursifs multiples

On va étudier maintenant le cas ou 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.

Voue 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 7.10
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 morceaux 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 $d(n)$.

* *
*