Informatique — MPSI/PCSI, 2024–2025

# TP n° 7 – Récursivité-Correction

## 1 Fonction récursive

**Exercice 7.1**
Version itérative de l'algorithme d'Euclide.

```
def euclide_it(a,b):
    while b!=0:
        r=a%b
        a=b
        b=r
    return a
```

Version récursive.

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

**Exercice 7.2**
Le seul point auquel il faut être attentif est l'ordre des instructions : on appelle la fonction à l'indice $n-1$ puis on affiche une nouvelle ligne dans le premier cas, on affiche une nouvelle ligne puis on affiche la fonction dans le second.

1.
```
def affiche_etoiles(n):
    if n>1:
        affiche_etoiles(n-1)
    print('*'*n)
```

2.
```
def affiche_etoiles_inv(n):
    print('*'*n)
    if n>1:
        affiche_etoiles_inv(n-1)
```

**Exercice 7.3** 1. La fonction quotient s'appelle elle-même avec les paramètres $p-d$ et $d$ tant que $d$ n'est pas strictement inférieur ) $p$.

```
def r_rec(p,d):
    if p<d:
        return p
    else:
        return r_rec(p-d,d)
```

2. On écrit une fonction prenant en entrée un troisième paramètre facultatif $n$. On peut ensuite obtenir le quotient en appelant cette fonction sans préciser la valeur de $n$.

```
def q_rec(p,d,n=0):
    if p<d:
        return n
    else:
        return q_recn(p-d,d,n+1)
```

**Exercice 7.4**

```
def permut(chaine):
    if l==1: # Dans le cas ou la chaine est de longuer 1, une seule permutation
        return [chaine]
    else:
        L=[]
        c=chaine[0]
        Li=permut(chaine[1:]) # genere toute les permuations des n-1 derniers caracteres
        for p in Li: # parcourt la liste des permuations precedentes
            for i in range(l): # insere le premier caractere a chacune des places possibles
                L.append(p[:i]+c+p[i:])
        return L
```

## 2 Généralisation

**Exercice 7.5**
Pour $n = 13$, on appelle la fonction successivement pour $n = 6$, $n = 3$, $n = 1$ et $n = 0$. Il faut deux multiplications pour calculer `exp_rapide(x,n)`: à partir de `exp_rapide(x,m)`: (où `m=n//2`) si $n$ est impair et une multiplication si $n$ est pair. On effectue donc au total $2+1+2+2 = 7$ multiplications.

En comparaison, l'algorithme de multiplication naïf demanderait à effectuer 12 multiplications.

**Exercice 7.6**

```
def rech_dicho(n,L):
    l=len(L)
    if l==1: # si la liste a un seul element on test s'il est egal a n
        if L[0]==n:
            return True
        else:
            return False
    else: #sinon on coupe la liste en 2 et on garde la sous-liste adaptee
        m=l//2
        if L[m-1]>=n:
            return rech_dicho(n,L[:m])
        else:
            return rech_dicho(n,L[m:])
```

## 3 Similitudes et fractale

**Exercice 7.7** 1. Si $N$ est l'image de $M$ par la rotation de centre $I$ et d'angle $t$, les coordonnées de l'image $M'$ de $M$ par la similitude de centre $I$, d'angle $t$ et de rapport $k$ sont obtenues en ajoutant à $I$ les coordonées du vecteur $k\overrightarrow{IN}$.
La fonction suivante exprime ce calcul.

```
def simil(I,M,k=1,t=0):
    N=rotation(I,M,t)
    return [I[i]+k*(N[i]-I[i]) for i in range(2)]
```

2. Il suffit d'appliquer la fonction précédente à tous les élélments de $L$ en parcourant cette liste par valeurs.

```
def simil_poly(I,L,k=1,theta=0):
    return [simil(I,e,k,theta) for e in L]
```

**Exercice 7.8** 1. Si $A$ et $B$ désignent les deux extrémités d'un segment rejoignant deux points successifs de $L$ et $n > 0$ la courbe apparaissant après une nouvelle étape de construction rejoints les points $A$, $C$, $E$ et $D$ définis comme suit :

— $C$ et $D$ verifient $\overrightarrow{AC} = \frac{1}{3}\overrightarrow{AB}$ et $\overrightarrow{AE} = \frac{2}{3}\overrightarrow{AB}$
— $E$ est l'image du point $D$ par la rotation de centre $C$ et d'angle $\frac{\pi}{3}$.

Il suffit donc de calculer les coordonnées de chacun des ces trois points, (en utilisant la fonction `simil` pour celles de $E$, puis d'appeller récursivement la fonction à l'ordre $n-1$ pour chacun des segments $[AC]$, $[CE]$, $[ED]$ et $[DB]$. On concatène ensuite les listes obtenues, en tenant compte du fait qu'on peut ne pas tenir compte du premier élément de chaque liste (à part pour la première) qui est le dernier élément de la liste précédente.
Dans le cas $n = 0$, la liste $L$ est laissée inchangée.
La fonction suivante réalise cette construction en faisant appel à la fonction `simil` pour déterminer les coordonnées de $E$.

```
def von_koch_cote(L,n):
    if n==0:
        return L
    else:
        # Determination de coordonnée de C, D et E
        A,B=L[0],L[1]
        AB=[B[i]-A[i] for i in range(2)]
        C=[A[i]+AB[i]/3 for i in range(2)]
        D=[A[i]+2*AB[i]/3 for i in range(2)]
        E=simil(C,D,1,ma.pi/3)
        # On appelle récursivement la fonction su chaque segment
        # et on reconstitue la liste des sommets obtenus
        return von_koch_cote([A,C],n-1)+von_koch_cote([C,E],n-1)[1:]+von_koch_cote([E,D],n-1
```

2. On peut utiliser par exemple comme sommets du triangle équilatéral le points $A(1;0)$ et les points $B$ et $C$ obtenus à partir de $A$ par rotations successives de centre $O$ et d'angle $-\frac{2\pi}{3}$ (attention à bien prendre un angle de rotation négatif pour que les nouveaux triangles soient ajoutés à l'extérieur du triangle initial). Il suffit ensuite d'appeler pour chaque côté la fonction de la question précédente pour obtenir la liste despoints $L$ à relier.
On extrait ensuite les listes des coordonnées des points de $L$ et on utilise le module `mathplotlib.pyplot` (renommé ici `plt`).

```
def von_koch(n):
    # Création des sommets du triangle
    O=[0,0]
    A=[1,0]
    B=simil(O,A,1,-2*ma.pi/3)
    C=simil(O,B,1,-2*ma.pi/3)
    # Liste des points à tracer pour chaque côté
    L=von_koch_cote([A,B],n)+von_koch_cote([B,C],n)+von_koch_cote([C,A],n)
```

```
    # Affichage
    X,Y=[e[0] for e in L],[e[1] for e in L]
    plt.figure(figsize=(6.4,6.4))
    plt.plot(X,Y)
    plt.show()
```

**Exercice 7.9**

1. Si $n = 0$, on retourne $L$. Sinon, on appelle récursivement la fonction à l'ordre $n-1$ sur l'image de $L$ par la première similitude données par l'énoncée, puis on fait une rotation de la figure obtenue. Il faut prendre garde à inverser l'ordre des sommets de $L2$ pour bien parcourir les sommets de celle-ci en partant de l'extrémité de $L1$. Il est par ailleurs inutile de conserver le premier point de $L2$ qui coïncide avec le dernier de $L1$.

```
def points_dragon(L,n):
    if n==0:
        return L
    else :
        L1=(points_dragon(simil_poly(L[0],L,ma.sqrt(2)/2,ma.pi/4),n-1))
        L2=simil_poly(L1[-1],L1,1,ma.pi/2)
        L2.reverse()
        return L1+L2[1:]
```

2. Il reste à appeler la fonction précédente en partant des points $A(0;0)$ et $B(1;0)$ et afficher le résultat à l'aide du module `mathplotlib.pyplot`.

```
def dragon(n):
    L=[[0,0],[1,0]]
    L=points_dragon(L,n)
    X,Y=[e[0] for e in L],[e[1] for e in L]
    plt.figure()
    plt.plot(X,Y)
    plt.show()
```

## 4 Appels récursifs multiples

**Exercice 7.10**

Si $n = 0$ ou $n = 1$, il y a zéro découpe possible. Si $n = 2$ ou $n = 3$, une seule. Sinon, on considère le morceau le plus à droite. Sa longueur est 2 ou 3. Dans le premier cas, il y a $d(n-2)$ découpes possibles pour la partie à gauche, et dans le deuxième cas $d(n-3)$, donc $d(n) = d(n-2) + d(n-3)$.

```
def nb_decoupes(n):
    if n<=1:
        return 0
    elif n<=3:
        return 1
    else:
        return nb_decoupes(n-2)+nb_decoupes(n-3)
```

**Exercice 7.11**

```
def binom(n,k):
    if k>n:
        return 0
    elif k==0 or k==n:
        return 1
    else:
        return binom(n-1,k)+binom(n-1,k-1)
```