TP 5 : Récursivité

I Le principe de récursivité

La récursivité repose sur le même principe que la récurrence : on résout “à la main” les premiers cas, puis pour résoudre un cas donné on se ramène au cas précédent. Selon le type de récurrence choisie, on pourra avoir différents programmes qui codent une même fonction.

Par exemple, pour connaître la parité d'un entier naturel n, on peut faire l'un des deux raisonnements suivants :

— avec une récurrence simple : 0 est pair, et $n > 0$ a la parité différente de $n-1$:

```
>>> def pair1(n):
...     if n==0 :
...         return True
...     return not pair1(n-1)
```

— avec une récurrence simple : 0 est pair, 1 est impair, et $n > 1$ a même parité que $n-2$:

```
>>> def pair2(n):
...     if n==0 :
...         return True
...     if n==1 :
...         return False
...     return pair2(n-2)
```

Notons au passage que les deux algorithmes précédents, s'ils n'ont pas la même complexité (le second fait deux fois moins d'opérations que le premier), les deux complexités sont linéaires et donc on ne parlera pas vraiment d'une amélioration en terme de complexité pour le premier. En revanche, il y a un point important : c'est le nombre d'**appels récursifs** : c'est le nombre de fois qu'une fonction fait appel à une autre fonction. Pour éviter des appels qui n'en finissent pas, et qui lanceraient des calculs infinis, Python a une sécurité qui limite le nombre d'appels récursifs autorisés. Par exemple, la fonction `pair2` pourra calculer la parité de 1000, mais pas la fonction `pair1`.

Un exemple historiquement important est la fonction d'Ackermann–Péter suivante, qui effectue énormément d'appel récursifs, même pour des petites valeurs de m, et qui est à l'origine de cette sécurité sur de nombreux ordinateurs :

```
>>> def A(m,n) :
...     if m==0 :
...         return n+1
...     if n==0 :
...         return A(m-1,1)
...     return A(m-1,A(m,n-1))
```

Exercice 1

Sur les algorithmes précédents, tester à partir de quelles valeurs de n (pour `pair1(n)` et `pair2(n)`) et de m (pour `A(m,2)`, `A(m,10)` et `A(m,1000)`) la sécurité de Python s'active et empêche l'exécution du programme.

Exercice 2

À l'aide d'une fonction récursive, coder la division euclidienne pour les entiers naturels par une fonction `diveuclide` qui prend en argument deux entiers naturels a, b avec $b \neq 0$ et rend la liste $[q, r]$ où q est le

quotient et r le reste dans la division euclidienne de a par b. Plus précisément, pour la division euclidienne de a par b, on pourra utiliser une récurrence d'ordre b.

Exercice 3

En utilisant (ou pas) la fonction `diveuclide` précédente, utiliser une fonction récursive pour coder la fonction `pgcd` qui rend le pgcd de deux entiers naturels. On rappelle que le pgcd de deux entiers se calcule, suivant l'algorithme d'Euclide, comme le dernier reste non nul dans les divisions euclidiennes successives.

Exercice 4

Écrire une fonction récursive `permutation(L)` qui, étant donnée une liste L, donne toutes les listes que l'on peut construire en permutant les éléments de L.

On utilisera que, si on considère toutes les permutations de la sous-liste `L[1:len(L)]`, il suffit d'insérer `L[0]` à une place quelconque de chaque permutation obtenue.

Par exemple, pour $L = [1, 2, 3]$ on obtient :

```
>>> permutation([1,2,3])
[[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]
```

II Versions récursives d'algorithmes itératifs

Exercice 5

Reprendre les algorithmes itératifs suivants pour le calcul de puissance ou de la factorielle, et les réécrire sous forme récursive :

```
>>> def expo(x,n) :
...     a=1
...     for i in range(n) :
...         a=a*x
...     return a
```

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

On pourra commencer par exprimer x^{n+1} en fonction de x^n, et $(n+1)!$ en fonction de $n!$.

Exercice 6

Réécrire l'algorithme de recherche dichotomique dans un tableau trié de manière récursive (qui donne seulement si l'objet est dans une liste, sans s'occuper de l'indice associé). On donne ci-dessous la version itérative. On pourra utilise que, pour deux entiers a, b et une liste L donnée, la commande `L[a:b]` rend la liste obtenue à partir de L en ne gardant que les éléments d'indice compris entre a (inclus) et b (exclu).

```
>>> def recherchedicho(L,l) :
...     if len(L) == 0 :
...        return False
...     a=0
...     b=len(L)-1
...     if (l<L[a]) or (l>L[b]) :
...        return False
...     if (l==L[a]) or (l==L[b]):
...        return True
...     while (b-a) > 1 :
...        m=(a+b)//2
...        if L[m]==l :
...            return True
...        if L[m]>l :
...            b=m
...        else :
...            a=m
...     return False
```

On pourra omettre certaines des comparaisons précédentes grâce à la récursivité, et on ne s'intéressera pas à la complexité de l'algorithme ainsi créé.

III Autour des tours de Hanoï

Le but du jeu des tours de Hanoï est le suivant : on dispose de trois tiges sur lesquelles sont empilés, du plus grand au plus petit, n disques de diamètres différents. L'objectif est de déplacer tous les disques sur la troisième tige, en respectant les contraintes suivantes :

— on ne déplace qu'un seul disque à la fois ;

— on ne peut pas poser un disque sur un disque de diamètre inférieur.

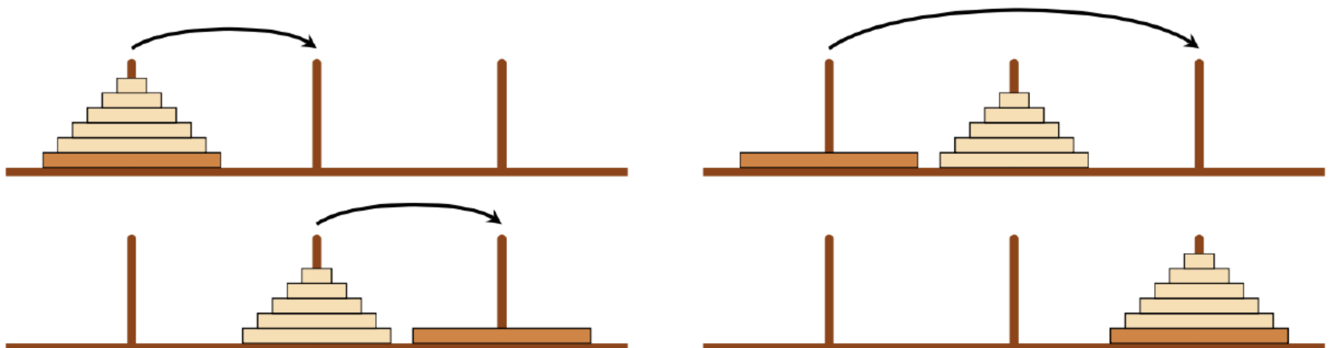
La résolution optimale (en terme de nombre de coups) se fait de la manière suivante :

— on fait, par la succession optimale de coups pour $n-1$ disques, le déplacement des $(n-1)$ plus petits disques de la première à la deuxième tige ;

— on déplace de plus grand disque de la première à la troisième tige ;

— on refait, par la succession optimale de coups pour $n-1$ disques, le déplacement des $(n-1)$ plus petits disques de la deuxième à la troisième tige.

Cette méthode correspond au dessin suivant :



La résolution se fait donc assez naturellement de manière récursive.

Exercice 7

Écrire une fonction récursive `hanoi(n,a=1,b=2,c=3)`, où n désigne le nombre de disques, placés en tige a, que l'on déplace vers la tige c, avec la tige vide b qui aide à faire les mouvements, et rend la liste des déplacements à faire, en nommant le disque à déplacer ainsi que la tige de départ et celle d'arrivée.

On vérifiera que les déplacements pour 3 disques sont donnés de la manière suivante :

```
>>> hanoi(3)
On déplace le disque  1  du sommet de la pile 1  vers le sommet de la pile  3 .
On déplace le disque  2  du sommet de la pile 1  vers le sommet de la pile  2 .
On déplace le disque  1  du sommet de la pile 3  vers le sommet de la pile  2 .
On déplace le disque  3  du sommet de la pile 1  vers le sommet de la pile  3 .
On déplace le disque  1  du sommet de la pile 2  vers le sommet de la pile  1 .
On déplace le disque  2  du sommet de la pile 2  vers le sommet de la pile  3 .
On déplace le disque  1  du sommet de la pile 1  vers le sommet de la pile  3 .
```

Exercice 8

Modifier le programme précédent pour qu'il représente les tiges avec les disques à chaque étape. On représentera les disques par des successions d'`*` (avec $2i-1$ fois le symbole `*` pour le disque de taille i), les tiges par des `|` (sur une hauteur de n), et le support par une succession d'un bon nombre de fois le symbole `-`.

On pourra chercher à créer la liste des disques présents sur chaque tige à chaque étape, et utiliser la fonction suivante qui, étant donnée les listes l_1, l_2, l_3 de taille n comportant dans l'ordre les disques présents sur chaque tige (complétées par des 0) pour tracer la situation correspondante :

```
def dessin(l1,l2,l3):
    n=len(l1)
    for i in range(n):
        k=n-1-i
        a=[l1[k],l2[k],l3[k]]
        b=[]
        for j in a:
            if j==0 :
                b.append('|')
            else :
                b.append('*')
        print(' '*(n-a[0]+1-(a[0]==0))+'*'*(a[0]-1)+b[0]+'*'*(a[0]-1)+' '*(n-a[0]+1-
(a[0]==0)) + ' '*(n-a[1]+1-(a[1]==0))+'*'*(a[1]-1)+b[1]+'*'*(a[1]-1)+' '*(n-a[1]+1
-(a[1]==0)) + ' '*(n-a[2]+1-(a[2]==0))+'*'*(a[2]-1)+b[2]+'*'*(a[2]-1)+' '*(n-a[2]+1
-(a[2]==0)))
    print('-'*(6*n+3))
```

Pour 3 disques, on obtient le résultat suivant :

```
>>> hanoidessin(3)
   *      |      |
  ***     |      |
 *****    |      |
--------------------
   |      |      |
  ***     |      |
 *****    |      *
--------------------
   |      |      |
   |      |      |
 *****   ***     *
--------------------
   |      |      |
   |      *      |
 *****   ***     |
--------------------
   |      |      |
   |      *      |
   |     ***   *****
--------------------
   |      |      |
   |      |      |
   *     ***   *****
--------------------
   |      |      |
   |      |     ***
   *      |    *****
--------------------
   |      |      *
   |      |     ***
   |      |    *****
--------------------
```

IV Dessins de fractales

Les fractales sont des objets mathématiques qui ont une particularité : ils ne changent pas si on zoome dessus. Pour faire une comparaison avec un objet de la vie de tous les jours (ou presque), on pourrait imaginer des poupées russes qui vont jusqu'à l'infiniment petit : à chaque poupée que l'on ouvre, on trouve la même infinité de poupées.

Cet infini n'est évidemment pas possible à avoir de manière exacte, et on ne les représente que de manière approchée. L'idée est donc d'augmenter le niveau de détail au fur et à mesure qu'on zoome sur la fractale. Et donc on peut en amont fixer un certain niveau de détail. Et les niveaux de détail s'adaptent très bien à la récursivité dans de nombreuses fractales célèbre, dont la courbe de von Kock.

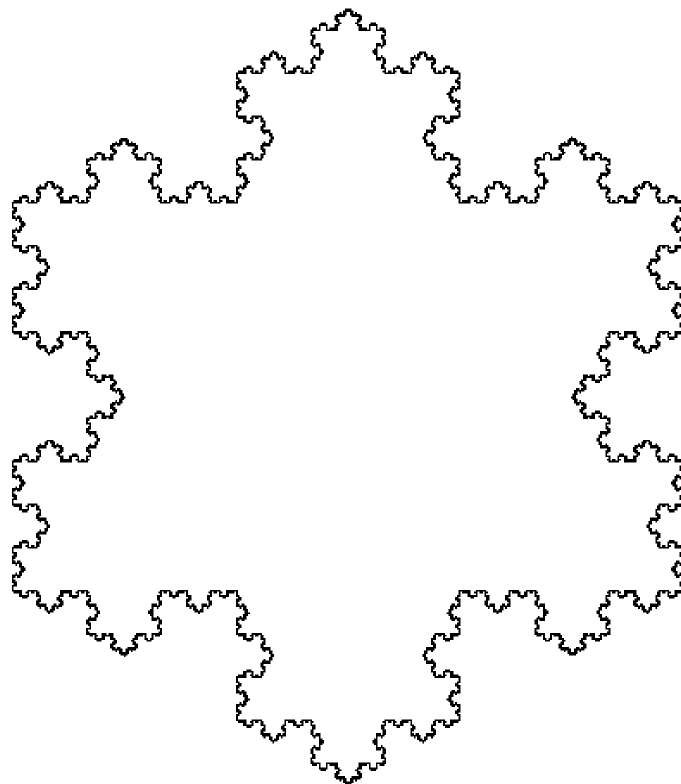
Pour les représentations de fractales, on repèrera les points par leurs coordonnées. On pourra le faire sous forme de listes, ou sous forme de tableaux. L'usage de tableau se fait avec le module `numpy`. On peut passer d'un tableau à une liste avec les commandes suivantes :

```
>>> import numpy as np

>>> A=[1,4]

>>> A=np.array(A)
>>> A
array([1, 4])

>>> A=list(A)
>>> A
[1, 4]
```

L'intérêt principal des tableaux est pour les opérations : on peut additionner élément par élément des tableaux avec l'opération + ce qui permet de faciliter les expressions de transformations géométriques dans le plan (comme les translations).

Exercice 9

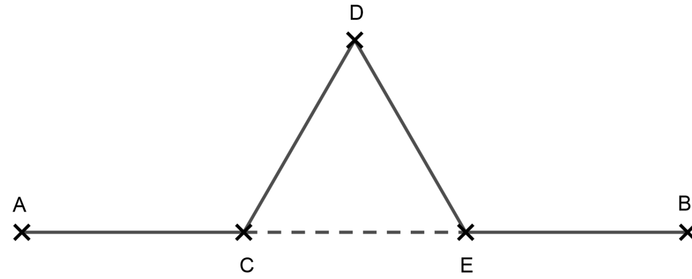
La courbe de von Koch, dont voici une image, est un peu comme un flocon de neige. On se propose de la coder récursivement



On ne va s'intéresser pour le moment qu'à un seul des six côtés du flocon. La récursivité se fait de la manière suivante :

— on part d'un ligne brisée, qu'on assimile à la lise de ses points de jonction $A_1, \ldots, A_n$;

— on coupe chaque segment $[A_i A_{i+1}]$ en quatre segments, ce qui revient à rajouter trois points dans la liste initiale des points (les deux points à 1/3 et 2/3 sur le segment initial, et le sommet de la pointe que l'on rajoute). On donne ci-dessous ce découpage, en partant des deux points $A = A_i$ et $B = A_{i+1}$ pour rajouter les points C, D, E.



Écrire une fonction `transformationvonKoch(A,B)` qui, étant donnés deux points A, B, retourne le quintuplet (A, C, D, E, B) qui correspond au découpage précédent. On fera bien attention que l'ordre des points est important, et que le fait d'inverser A et B revient à faire le symétrique par rapport à la droite (AB).

On rappelle que le module `math` permet de calculer certaines grandeurs mathématiques (comme $\sqrt{3}$) qui seront utiles pour le programme précédent.

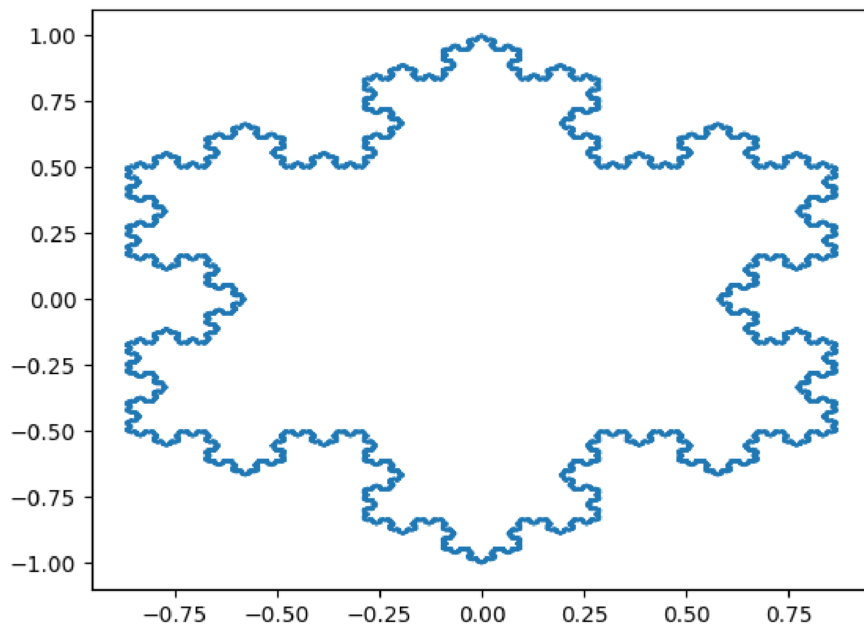
On pourra coder librement avec des points vus comme des tableaux ou des listes à deux éléments, mais on prendra garde à bien écrire le programme de manière cohérente avec ce choix de conventions.

Exercice 10

À l'aide de l'exercice précédent, construire une fonction `vonKoch(L,n)` qui, étant donnée une liste L de points, applique n fois la transformation précédente à la ligne brisée associée à L.

Tracer le flocon de von Koch associé à 5 itérations de `transformationvonKoch`.

On trouve alors la figure suivante :



en exécutant le code :

```
>>> import math as ma

>>> L=[np.array([ma.cos(5*ma.pi/6  - 2*n*ma.pi/3),  ma.sin(5*ma.pi/6  - 2*n*ma.pi/3)]) for

>>> L=vonKoch(L,7)

>>> X=[l[0] for l in L]

>>> Y=[l[1] for l in L]

>>> import matplotlib.pyplot as plt

>>> plt.plot(X,Y)

>>> plt.show()
```

Ou on peut aussi représenter des flocons de plus en plus détaillés, et de plus en plus grands, avec le script suivant :

```
>>> import matplotlib.pyplot as plt

>>> L=[np.array([ma.cos(5*ma.pi/6  - 2*n*ma.pi/3),  ma.sin(5*ma.pi/6  - 2*n*ma.pi/3)]) for

>>> for i in range(5):
...         X=[((1.5)**i) * l[0] for l in L]
...         Y=[((1.5)**i) * l[1] for l in L]
...         L=vonKoch(L,1)
...         plt.plot(X,Y)

>>> plt.show()
```