TRAVAUX PRATIQUES n° 14
Récursivité

Toutes les fonctions de ce TP doivent être écrites de manière récursive.

Exercice 1 (*Introduction*) : Recopier les fonctions récursives suivantes et les tester :

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

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

Exercice 2 (*Calcul d'une somme*) : Écrire une fonction `somme_bicarree(n)` prenant en argument un entier naturel `n` et renvoyant la valeur de la somme $\sum_{k=0}^{n} k^4$.

Exercice 3 (*Récursivité sur les listes*) :

1) Écrire une fonction `produit(L)` prenant en argument une liste de nombres `L` et renvoyant le produit de tous les éléments de `L`.
2) Écrire une fonction `minimum(L)` prenant en argument une liste de nombres `L` et renvoyant le minimum des éléments de `L`.

Exercice 4 (*Fonction puissance*) :

1) Écrire une fonction `puissance_0(x, n)` non récursive qui étant donnés un réel `x` et un entier naturel `n`, calcule et renvoie la valeur de x^n.
2) Écrire une fonction `puissance_1(x, n)` qui fait la même chose que `puissance_0` mais de manière récursive en utilisant la formule de récurrence :

$$\begin{cases} x^0 = 1 \\ \forall n \in \mathbb{N}^*, \quad x^n = x \times x^{n-1} \end{cases}$$

3) Écrire une fonction `puissance_2(x, n)` qui fait la même chose que `puissance_1` mais en utilisant la formule de récurrence :

$$\begin{cases} x^0 = 1 \\ \forall n \in \mathbb{N}^*, \quad x^n = \begin{cases} (x \times x)^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ x \times (x \times x)^{\frac{n-1}{2}} & \text{si } n \text{ est impair} \end{cases} \end{cases}$$

4) Tester les instructions suivantes et expliquer ce qu'il se passe.

```
>>> puissance_1(5, 2000)
>>> puissance_2(5, 2000)
```

5) Tester le code suivant qui permet de comparer les deux algorithmes itératif contre récursif. Quel est le meilleur ?

```
from time import time

t0 = time()
puissance_0(5, 100000)
t1 = time()
print("La temps d'exécution de l'instruction puissance_0(5, 1000) est {}".format(
    t1-t0))

t0 = time()
puissance_2(5, 100000)
t1 = time()
print("La temps d'exécution de l'instruction puissance_2(5, 1000) est {}".format(
    t1-t0))
```

Exercice 5 (*Fibonacci*) :

1) Écrire une fonction `Fibonacci(n)` qui prend en argument un entier naturel `n` et qui renvoie la valeur du n^e terme de la suite de Fibonacci définie par $F_0 = F_1 = 1$ et pour tout $n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$.
2) Tester votre fonction pour $n = 40$ et pour $n = 100$ (il est conseillé de faire une sauvegarde avant...)

Exercice 6 (*Coefficients binômiaux*) :

1) Écrire une fonction `binom_1(n, p)` qui prend en argument deux entiers naturels `n` et `p` et qui renvoie la valeur du coefficient binomial $\binom{n}{p}$ en utilisant la formule de Pascal :

$$\forall n, p \in \mathbb{N}^*, \qquad \binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

2) Écrire une fonction `binom_2(n, p)` qui prend en argument deux entiers naturels `n` et `p` et qui renvoie la valeur du coefficient binomial $\binom{n}{p}$ en utilisant la formule du chef :

$$\forall n, p \in \mathbb{N}^*, \qquad \binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$$

3) Comparer ces deux fonctions en exécutant les instructions suivantes :

```
>>> binom_1(100, 2)
>>> binom_2(100, 2)
>>> binom_1(100, 50)     # Attention !
>>> binom_2(100, 50)
```

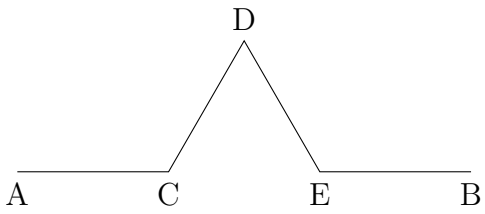
Exercice 7 (*Permutations (difficile)*) :

Écrire une fonction `permuations(L)` qui prend en argument une liste `L` et qui renvoie la liste de toutes les permutations des éléments de `L`.

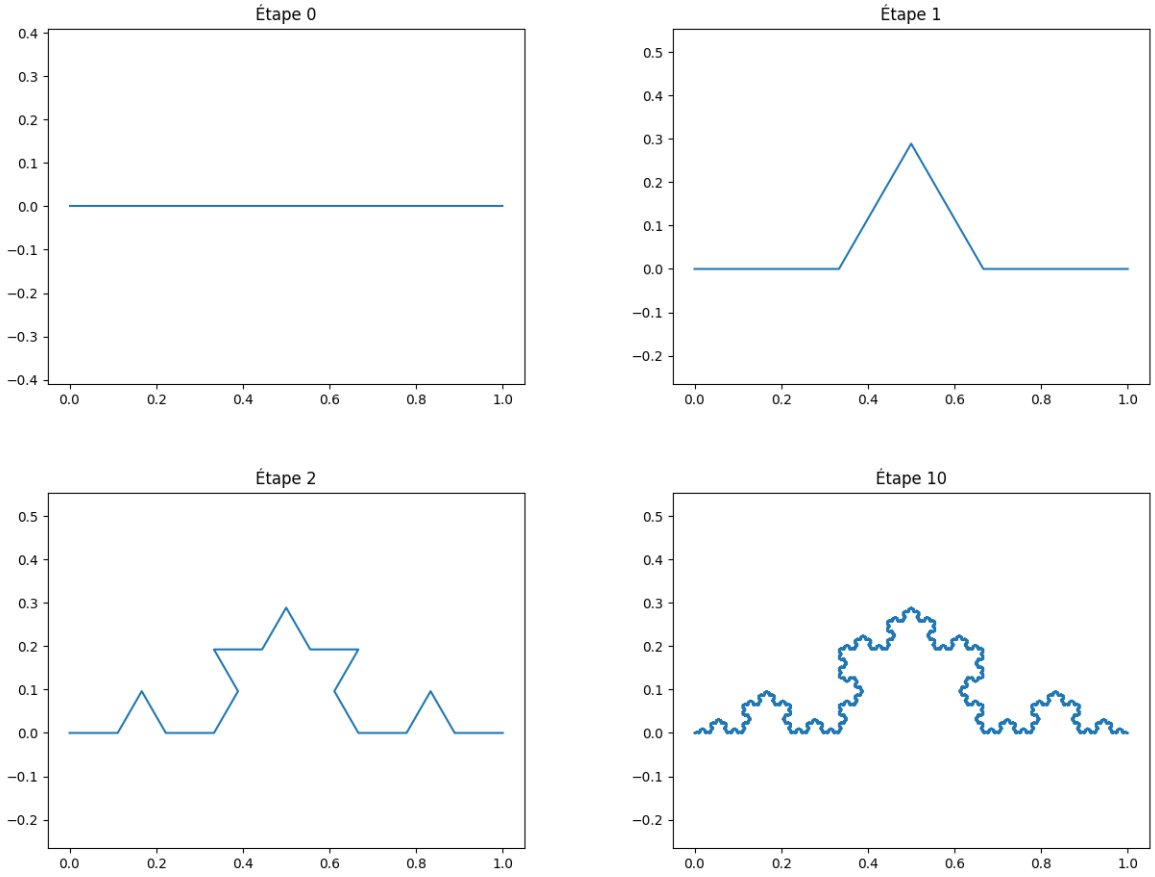
Par exemple `permutations([1, 2, 3])` doit renvoyer la liste `[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2],[3, 2, 1]]` mais pas nécessairement dans cet ordre.

Exercice 8 (*Flocon de von Koch*) : Pour construire le flocon de von Koch, on procède de la manière suivante :

- ⋆ Étape 0 : la courbe est un segment $[AB]$.
- ⋆ Étape 1 : on remplace le segment $[AB]$ par la figure suivante de sorte à avoir $AC = CD = DE = CE = EB$:



- ⋆ Étape 2 : on applique la même transformation à tous les segments $[AC]$, $[CD]$, $[DE]$ et $[EB]$.
- ⋆ On recommence ainsi de suite pendant un certain nombre d'étapes.



1) Définir trois fonctions Python qui, à partir des coordonnées de deux points A et B du plan, calculent les coordonnées des points C, D et E de l'étape 1 respectivement.

   Les coordonnées de ces points sont obtenus par les formules suivantes : si $\overrightarrow{AB}\,(x, y)$, alors

$$\overrightarrow{AC}\left(\frac{x}{3}, \frac{y}{3}\right)$$
$$\overrightarrow{AD}\left(\frac{1}{2}x - \frac{\sqrt{3}}{6}y, \frac{1}{2}y + \frac{\sqrt{3}}{6}x\right)$$
$$\overrightarrow{AE}\left(\frac{2x}{3}, \frac{2y}{3}\right)$$

   On représentera les coordonnées d'un point par une liste de deux éléments.

2) Écrire une fonction récursive `von_koch(A, B, n)` qui prend en arguments deux points du plan `A` et `B` (liste de leurs coordonnées) et un entier `n` et qui renvoie la liste des coordonnées des points de la courbe de von Koch après `n` étapes.
3) Tracer alors la courbe obtenue. On rappelle les instructions pour tracer un graphe :

```
import matplotlib.pyplot as plt
plt.figure()
plt.title("Flocon de von Koch")
plt.plot(abscisses, ordonnees)
plt.show()
```