TP 7 : récursivité

On rapelle le code du tri par sélection :

```
def maxi(liste , n):
    sous_liste = liste[ : n]
    return sous_liste.index(max(sous_liste))
def tri_selection(liste):
    n = len(liste)
    for k in range(n, 1, -1):
        ind = maxi(liste , k)
        liste[ind], liste[k-1] = liste[k-1], liste[ind]
    return liste
```

On a constaté que ce code (ainsi que l'autre tri que l'on a vu, le tri par insertion) était bien plus lent que la méthode `sort` utilisée par Python. Voyons par quel procédé on peut drastiquement améliorer la vitesse de calcul.

Fonctions récursives et l'approche diviser pour régner

Une fonction **récursive** est une fonction qui s'appelle elle-même. Par exemple on a $0! = 1$ et

$$\forall n \in \mathbb{N}^* \ , \ n! = (n-1)! \times n \ .$$

Autrement dit si on a déjà calculé factorielle $n-1$, il suffit de multiplier cela par n pour obtenir $n!$.
On en déduit le petit code Python suivant qui calcule la factorielle d'un entier :

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

La fonction `factorielle` s'appelle elle-même : pour calculer `factorielle(3)`, on demande à l'ordinateur de calculer `factorielle(2)`, qu'on multiplie ensuite par 3.

Exercice 1.

1. Testez ce programme en calculant les factorielles des 100 premiers entiers.
2. Donnez à votre professeur le nombre de chiffres de 100!. On utilisera `len(str(p))` pour trouver le nombre de chiffres du nombre entier `p`.
3. Testez le programme suivant, légèrement modifié, avec $n = 6$ et méditez sur l'ordre des affichages.

```
def factorielle(n):
    if n == 0:
        resultat=1
    else:
        print('-'*n+'> appel de factorielle ', n-1)
        resultat = n * factorielle(n-1)
        print('-'*n+'> sortie de factorielle ', n-1)
    return resultat
```

La récursivité est à la base d'un type d'algorithmes appelé « diviser pour régner ». Le principe est de découper les données d'entrée du problème en deux paquets distincts et d'appliquer la procédure souhaitée à chaque paquet séparément. La procédure s'appelle alors elle-même et découpe donc à nouveau chacun de ces deux paquets de données en deux paquets, et ainsi de suite.... Voici un exemple très simple : l'exponentiation rapide.

Étant donné un réel a quelconque et un entier naturel n, on a

$$a^n = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & \text{si } n \text{ est pair} \\ a \times \left(a^{\frac{n-1}{2}}\right)^2 & \text{si } n \text{ est impair} \end{cases}$$

On en déduit un algorithme d'exponentiation de type « diviser pour régner ».

```
def expo(a,n):
    if .......:
        return
    else:
        if .........:
            return ........
        else:
            return ........
```

Un algorithme de tri rapide

Le tri rapide (quicksort en anglais) est une des méthodes de tri les plus rapides. L'idée est de choisir un élément dans la liste (disons le premier élément) puis de le mettre définitivement à sa place en mettant (sans les trier) à sa gauche tous les éléments inférieurs à ce nombre et à sa droite tous les éléments supérieurs. On recommence ensuite le tri sur les deux sous-listes obtenues.

Algorithme : tri rapide
Données d'entrée : une liste `liste` de taille `n`.
- le pivot est `liste[0]`
- à partir de la liste `liste[1:n]` on constitue la sous-liste `liste1` formée des éléments inférieurs ou égaux au pivot et la liste `liste2` formée des éléments supérieurs strictement au pivot.
- on appelle récursivement le tri sur ces deux sous-listes
- on concatène les deux sous-listes désormais triées avec le pivot avec l'opérateur « + » : `liste1+[liste[0]]+liste2` permet de recoller les listes entre elles, en ajoutant le pivot choisi au milieu

Exercice 2.

1. Testez cet algorithme au **PAPIER CRAYON** sur la liste `[-8, 5, 1, 7, -10, -12, -15, 3, -9]`
2. Codez cet algorithme avec le code suivant. Vérifiez votre code sur la liste précédente.
3. Faites à nouveau le comparatif entre ce tri et celui donné par `sort()` sur une liste aléatoire de taille 100000.