# TP 5 : tris de listes, une introduction à l'algorithmique

## Listes en Python (fin)

Pour créer des listes, Python nous fournit un outil très pratique, très similaire aux mathématiques, qui définit des listes en compréhension (comme dans les ensembles mathématiques : l'ensemble des $f(x)$ avec $x \in E$). La syntaxe pour définir une telle liste est la suivante :

```
[ f(x) for x in liste ]
```

à rapprocher de la syntaxe mathématique

$$\{f(x)\ ,\ x \in E\}$$

Voici quelques exemples, exécutez les et modifiez les pour vérifier que vous avez bien compris tous les tenants et aboutissants.

```
>>> liste = [2, 4, 6, 8, 10]
>>> [3*x for x in liste]
[6, 12, 18, 24, 30]
>>> [[x, x**3] for x in liste]
[[2, 8], [4, 64], [6, 216], [8, 512], [10, 1000]]
>>> [3*x for x in liste if x > 5]
[18, 24, 30]
>>> [3*x for x in liste if x**2 < 50]
[6, 12, 18]
>>> liste2 = list(range(3))
>>> [x**y for x in liste for y in liste2]
```

**Exercice 1.**

1. En une seule ligne de code Python, générez une liste contenant tous les entiers pairs compris entre 3 et 59.
2. En une seule ligne de code Python, générez une liste contenant tous les nombres entiers naturels inférieurs ou égaux à 1000 qui soient multiples de 4 mais non multiples de 8.
   *Vérifiez que votre liste contient bien exactement* ~~500~~ *éléments.* 125
3. En une seule ligne de code Python, générez la liste des diviseurs de 2621970.
   *Pour vérifier : la liste contient* 96 *éléments en incluant* 1 *et* 2621970 *lui-même.*
4. Écrivez une fonction Python nommée `listediviseurs` qui prend en argument un entier `n` et qui renvoie la liste de ses diviseurs.
   *Votre code devra faire deux lignes. Vérifiez le avec* $n = 2621970$.
5. Écrivez une fonction python `isprime` d'argument `n` qui renvoie le Booléen `True` si le nombre $n$ est premier et `False` sinon.
6. Donner la liste des premiers entiers premiers qui sont de la forme $2^p - 1$ avec $p$ premier (on les appelle les nombres premiers de Mersenne).
   *Pour vérifier :* $2^{23} - 1$ *n'est pas premier alors que* 23 *est bien premier. Attention la puissance de calcul de votre ordinateur sera vite dépassée, allez-y doucement au delà de 23...*
   **Lucas a démontré en 1876 après 19 ans d'acharnement que $2^{127} - 1$ est bien premier... On ne sait toujours pas si il y a une infinité de nombres premiers de Mersenne. Le projet GIMPS s'attelle à la recherche de tels nombres premiers. On en connaît pour l'instant seulement 51...**

# Algorithmes de tris de listes

Un algorithme est une suite finie d'instructions écrite dans le but de résoudre un problème donné, qui en prenant en entrée un nombre fini de données renvoie après un nombre fini d'opérations un résultat sous la forme d'un autre nombre fini de données.

On présente les algorithmes sous plusieurs formes : diagrammes de flot (voir TP 3 page 2), dans un langage de programmation quelconque (comme Python), ou simplement par une suite d'instructions textuelles comme ci-dessous.

On présente ici deux algorithmes simples de tri de liste.

**Algorithme : tri par sélection** *Données d'entrée :* une liste
- On recherche le plus grand élément de la liste
- On le met à la fin de la liste
- On cherche le plus grand élément de la liste sans son dernier élément
- On le place en avant dernière position
- On cherche le plus grand élément de la liste sans ses deux derniers éléments
- On le place en avant-avant dernière position
- Etc...

*Sortie :* la liste triée

Voici un exemple d'exécution de cet algorithme sur la liste $[4, 7, 1, 3, 19, 7, 2, 1]$
(en gras on montre la partie triée de la liste au fur et à mesure qu'elle se constitue) :

| Étape | État de la liste | Maximum de la partie non triée |
|---|---|---|
| 0 | [4,7,1,3,19,7,2,1] | 19 |
| 1 | [4,7,1,3,7,2,1,**19**] | 7 |
| 2 | [4,1,3,7,2,1,**7,19**] | 7 |
| 3 | [4,1,3,2,1,**7,7,19**] | 4 |
| 4 | [1,3,2,1,**4,7,7,19**] | 3 |
| 5 | [1,2,1,**3,4,7,7,19**] | 2 |
| 6 | [1,1,**2,3,4,7,7,19**] | 1 |
| 7 | [1,**1,2,3,4,7,7,19**] | 1 |
| 8 | [**1,1,2,3,4,7,7,19**] | - |

**Exercice 2.**

1. Écrire une fonction Python nommée `maxi(l, n)` qui prend en argument une liste `l` et un entier `n` et qui renvoie l'indice du maximum des `n` premiers éléments de `l`. (Si ce maximum apparaît à plusieurs positions, on reverra la position de la première occurence du maximum dans la liste).
2. À l'aide de cette fonction, écrire une fonction `tri_selection` qui prend en argument une liste et qui renvoie une version triée de la liste via l'algorithme de tri par insertion. On utilisera la fonction `maxi` de la question précédente, une boucle `for` ainsi que l'astuce `x, y = y, x` qui permet d'échanger deux variables.
3. Dans l'interpréteur, entrez la commande `from random import *`. Vous pouvez désormais générer des réels aléatoirement entre 0 et 1 à l'aide de la commande `random()`. Utilisez cette commande pour remplir une liste aléatoire contenant 10000 réels à l'aide de `liste = [random() for k in range(10000)]`. Puis testez votre tri sur cette liste. Comparez la vitesse de votre tri avec celle de la méthode `sort()` en faisant `print(liste.sort())`.

Voici maintenant une autre version du tri d'une liste : le tri par insertion.

**Tri par insertion**
- On laisse le premier élément de la liste à sa place
- On place le deuxième élément de la liste à gauche ou à droite du premier selon si il est plus petit ou plus grand que le premier
- On recommence avec le troisième élément, inséré à la bonne place : position 1 ou 2 ou 3 selon sa valeur
- Etc...

**Exercice 3.**

1. Écrire une fonction `insertion(l, n)` qui prend en argument une liste `l` dont on suppose les `n` premiers éléments déjà triés et qui insère `l[n]` à sa place parmi les `n` premiers éléments de `l`.
   *Indication : on pourra pour ce faire échanger `l[n]` avec son voisin de gauche tant qu'il est strictement plus grand que ce voisin de gauche.*
2. Écrire une fonction `tri_insertion` qui réalise le tri par insertion. Testez le comme dans l'exercice précédent et comparez à nouveau avec la méthode `sort()`.

Pour rivaliser de vitesse avec la méthode de Python, il va nous falloir utiliser un algorithme plus puissant.

## Fonctions récursives et l'approche diviser pour régner

Une fonction **récursive** est une fonction qui s'appelle elle-même.

Par exemple si on définit pour tout $n$ l'entier $n!$ (on dit $n$ « factorielle ») par

$$n! \underset{\text{def.}}{=} 1 \times 2 \times \cdots \times (n-1) \times n \ .$$

On pose par convention $0! = 1$ et on remarque qu'on a alors

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

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 :

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

**Exercice 4.**

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`. 158

On modifie légèrement le programme afin de visualiser précisément ce qui se passe :

```
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
```

**Exercice 5.**

Testez ce programme avec $n = 6$ et méditez sur l'ordre des affichages.

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 n == 0:
        return 1
    else:
        if n % 2 == 0:
            return expo(a, n // 2)**2
        else:
            return a * expo(a, (n-1) // 2)**2
```

## 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.

Par exemple avec la liste $[10, 1, 5, 19, 3, 3, 2, 17]$ :
— on commence par choisir le pivot 10
— on constitue les deux sous-listes $[1, 5, 3, 3, 2]$ et $[19, 17]$
— on appelle récursivement le tri sur ces deux sous-listes
— on concatène les résultats et le pivot pour obtenir la liste triée
L'algorithme est alors

```
def tri_rapide(liste):
    if liste == [] :
        return []
    else:
        liste1 = [x for x in liste[1:] if x <
    liste[0]]
        liste2 = [x for x in liste[1:] if x >=
     liste[0]]
        return tri_rapide(liste1)+[liste[0]]+
    tri_rapide(liste2)
```

**Exercice 6.**

Faites à nouveau le comparatif entre ce tri et celui donné par `sort()` sur une liste aléatoire de taille 100000.

## (bonus) Test de primalité de Lucas-Lehmer

La suite de Lucas est la suite définie par récurrence par

$$s_0 = 4$$
$$\forall n \in \mathbb{N}\ ,\ s_{n+1} = s_n^2 - 2$$

**Théorème**

Soit $p$ un nombre premier différent de 2. On note $M_p$ le nombre de Mersenne $M_p = 2^p - 1$. Alors

$$M_p \text{ est premier} \quad \Leftrightarrow \quad s_{p-2} \textbf{ est divisible par } M_p$$

**Exercice 7.**

1. Vérifiez progressivement la primalité des premiers nombres de Mersenne $M_p$ avec $p$ premier.
2. Parvenez-vous à tester la primalité de $M_{127}$ ? Quel est le plus grand nombre de Mersenne dont vous parvenez à tester la primalité en quelques secondes ?
3. On peut en fait calculer les termes successifs de la suite « modulo $M_p$ »c'est-à-dire qu'on peut remplacer, dans le calcul de $s_{n+1}$, $s_n$ par le reste dans la division de $s_n$ par $M_p$ (à l'aide de l'opérateur % en python).
   Écrivez alors un code qui teste la primalité de $2^{127} - 1$.