TNSI	Langages & programmation Récursivité	TP – **Récursivité**

Important : Vous devez vérifier à chaque fois que vous écrivez du code que celui-ci fonctionne. A vous d'écrire les tests unitaires et/ou les assertions nécessaires (ces tests n'ont pas besoin de se retrouver dans la version finale du travail que vous rendrez : ils peuvent être retirés ou mis en commentaire).

I – Analyse d'une fonction récursive

Soit le programme Python suivant :

```
def f(a, b) :
    """ a et b sont deux entiers naturels non nuls """
    if b == 1 :
        return a
    return a + f(a, b-1)
print(f(3, 5))
```

1) Déterminer, sans utiliser d'ordinateur, le résultat affiché par ce programme.
2) Identifier le point d'arrêt de cette fonction récursive.
3) Démontrer que le programme finira par s'arrêter.
4) Que retourne `f(a, b)` (a et b étant des entiers naturels non nuls) ?

II – Occurrences

Réaliser **de manière récursive** une fonction `nb_occurrences(s, c)` qui renvoie le nombre d'occurrences du caractère `c` dans la chaîne `s`.

III – Parcours d'une arborescence de fichiers

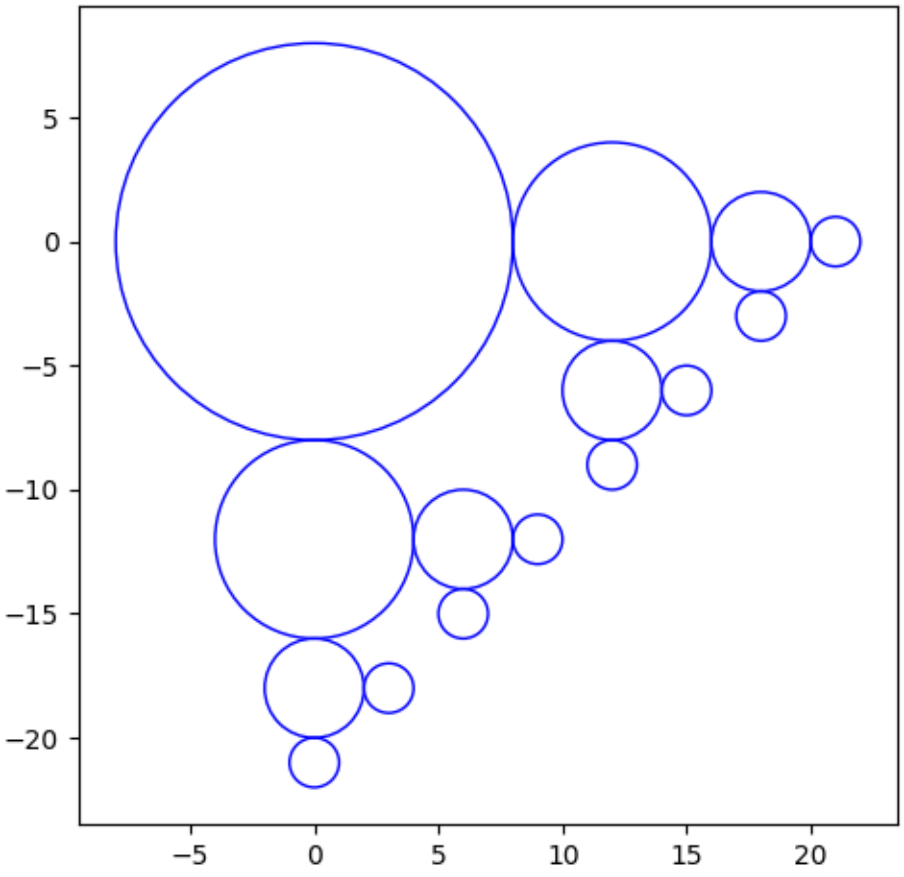
- La fonction `os.listdir(path)` (du module `os`) permet d'obtenir la liste de tous les fichiers et dossier placés directement dans le dossier désigné par le chemin `path`.
- La fonction `os.path.isfile(path)` (du module `os.path`) renvoie `True` si le chemin `path` désigne un fichier (et donc `False` s'il s'agit d'un dossier).
- La fonction `os.path.splitext(path)` permet de séparer l'extension du reste du chemin `path`, sous la forme d'une liste [racine, extension].
- La fonction `os.path.join(path1, path2, ...)` permet de concaténer des éléments de chemin sans trop se soucier de la syntaxe, qui rappelons-le, varie d'un OS à l'autre !

En utilisant ces seules fonctions, écrire une fonction `liste_fichiers(path, ext)` qui renvoie la liste de tous les fichiers ayant l'extension `ext` situés dans d'un dossier désigné par le chemin `path`, ainsi que dans tous ses sous-dossiers.

IV - Figures récursives

On peut décrire la figure suivante de façon *récursive* :

La figure est formée d'un cercle et de deux copies de ce cercle ayant subies une réduction d'un facteur 2, ces deux petits cercles étant tangents extérieurement au cercle initial et tels que les lignes des centres sont parallèles aux axes du repère. Ces deux petits cercles deviennent à leur tour "cercle initial" pour poursuivre la figure.



On peut traduire avec Python ce descriptif récursif de la façon suivante :

```
import matplotlib.pyplot as plt

def cercleRec(x, y, r):
    cercle = plt.Circle((x, y), r, color='b',fill=False)
    axe.add_patch(cercle)
    if r > 1:
        old_r = r
        r = r//2
        cercleRec(x+old_r+r, y, r)
        cercleRec(x, y-old_r-r, r)

axe = plt.gca()
cercleRec(0,0,8)
plt.axis('scaled')
plt.show()
```

1) Qu'est ce qui garantit que cette fonction ne s'appellera qu'un nombre fini de fois ?
2) Dans l'appel initial, si l'on change `cerclesRec(0, 0, 8)` par `cerclesRec(0, 0, 64)`, qu'obtiendra-t-on ?

V – Tri fusion

On souhaite programmer une implémentation en python du tri fusion (voir la page wikipédia) qui est un algorithme du type « diviser pour régner ».

1) Programmer et tester une fonction `fusion(t1, t2)` qui prend en argument deux tableaux triés `t1` et `t2` d'entiers et renvoie un tableau trié contenant les éléments de `t1` et de `t2`. Vous pouvez programmer cette fonction de manière itérative ou récursive au choix (mais la version récursive augmente sensiblement le temps de calcul sur des grands tableau).

Par exemple `fusion([4,7,14,21], [3,7,18,23])` renverra `[3,4,7,7,14,18,21,23]`.

2) Quelle est la complexité de cet algorithme ?

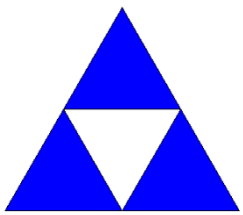
3) Programmer une fonction récursive `tri_fusion(t)` qui renvoie une version triée du tableau d'entier `t`.

4) Démontrer simplement que la complexité de cet algorithme est en $O(n \cdot \log(n))$.

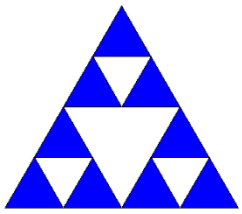
VI – Triangle de Sierpiński

Le triangle de Sierpiński est une figure fractale célèbre.

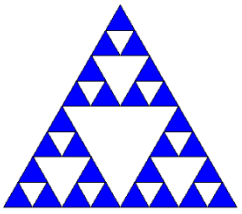
1) Consulter la page wikipédia du triangle de Sierpiński (le début de la page, jusqu'à Construction/Algorithme 1 inclus peut suffire).



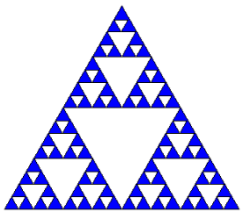
Ordre n = 0



Ordre n = 1



Ordre n = 2



Ordre n = 3

2) A l'aide de `turtle` ou d'une autre bibliothèque graphique, écrire une fonction `sierpinski(n, l)` qui trace le triangle de Sierpinsky jusqu'à l'ordre n en prenant un triangle de base de longueur l.

Aide : La fonction `sierpinsky` doit juste tracer le triangle plein de base puis appeler une fonction récursive `sierpinsky_rec(n, l)` qui se charge de tracer récursivement le triangle blanc au centre et de s'appeler elle-même pour les 3 sous-triangles ainsi formés.