TP 6 : Introduction à la récursivité

Exercice 1. *Itératif et récursif*. On considère dans cet exercice la suite définie par $u_0 = a$ et $u_{n+1} = 3u_n - 2n + 4$

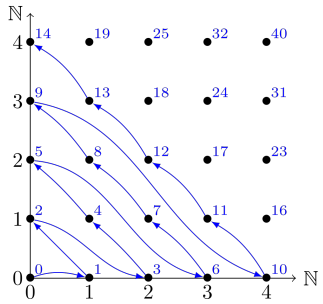
1. Écrire une fonction *itérative* `terme(a,n)` (c'est-à-dire avec une boucle) calculant u_n.

```
>>> terme(4,5)
1340
```

2. Écrire une fonction *récursive* `terme2(a,n)` (c'est-à-dire qui s'appelle elle-même) calculant u_n.

```
>>> terme2(4,5)
1340
```

Exercice 2. *Fonction de Cantor*. On numérote les couples d'entiers naturels de la façon suivante (ce qui explicite une bijection entre $\mathbb{N}$ et $\mathbb{N}^2$). On peut donner une formule explicite pour cette bijection, mais ce n'est pas le but de l'exercice !



1. Écrire une fonction récursive `couple(n)` renvoyant le couple associé à un entier. Par exemple, `couple(8)` renvoie $(1, 2)$. *Indication :* que renvoyer pour $n = 0$? Pour $n > 0$, calculer `couple(n-1)` récursivement, et voir comment en déduire le résultat (il y a deux cas, suivre les flèches !).

2. Inversement, écrire une fonction `enum(x,y)` renvoyant le numéro donné à un couple (x, y). *Indication :* de même, quel est le cas de base ? Sinon, trouver le point précédent (en suivant les flèches à l'envers), calculer son numéro et renvoyer celui-ci, plus 1.

Exercice 3. *Toutes les listes binaires*. Écrire une fonction récursive `listes_bin(n)` produisant une liste contenant les 2^n listes de taille n constituées de zéros et de uns (dans l'ordre que vous voulez). Par exemple :

```
>>> listes_bin(3)
[[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
```

Indication : Quel est le cas de base $n = 0$ (il y a une petite subtilité). Sinon, faire un appel récursif, et procéder par concaténation : `[0]+x` ou `[1]+x` avec `x` une liste, permet de créer une nouvelle liste contenant un élément en plus au début.
Remarque : ce qu'on cherche à calculer prend une place mémoire $O(n2^n)$ (ce sera également la complexité temporelle de la fonction). Ne pas faire un appel avec un trop gros n !

Exercice 4. *Tri fusion*. On va écrire pour la première fois un tri efficace, en $O(n \log n)$ où n est la taille de la liste à trier. Cette complexité sera démontrée en cours, plus tard.

1. Écrire une fonction `fusion(L1, L2)` prenant en entrée deux listes supposées triées dans l'ordre croissant, et renvoyant une liste contenant les mêmes éléments que `L1+L2`, mais triée. On suivra le schéma suivant :

```
def fusion(L1, L2):
    i1, i2 = 0, 0
    n1, n2 = len(L1), len(L2)
    L = []
    for i in range(n1+n2):
        if ...
            L.append(L1[i1])
            i1+=1
        else:
            ...
    return L
```

2. Le tri fusion consiste à suivre le schéma suivant pour renvoyer une copie triée de la liste `L` passée en paramètre (on ne modifie pas la liste initiale mais on renvoie une nouvelle liste).
   — si la liste est de taille 0 ou 1, renvoyer une copie de `L` (c'est-à-dire `L[:]`) ;
   — sinon, séparer `L` en deux parties de même taille à 1 élément près, calculer des copies triées via 2 appels récursifs, et fusionner les listes obtenues.

   Écrire `tri_fusion(L)` qui suit ce principe.

Exercice 5. *Des fractales.* Télécharger le fichier joint sur mon site. Vous y trouverez le code ci-dessous :

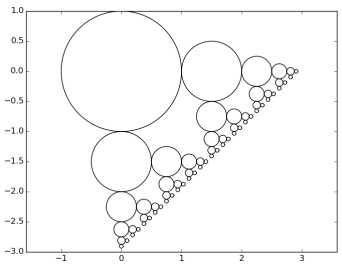
```
import matplotlib.pyplot as plt
import numpy as np
def cercle(x,y,r):
    theta = np.linspace(0, 2*np.pi, 100) #des points régulièrement espacés dans l'intervalle [0,2pi]
    X = [x+r*np.cos(t) for t in theta] #abscisses de points du cercle C((x,y),r)
    Y = [y+r*np.sin(t) for t in theta] #ordonnées de points du cercle C((x,y),r)
    plt.plot(X,Y) #tracé (pas d'affichage dans cette fonction)
```

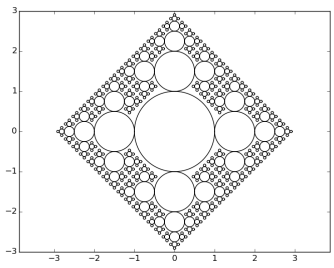
La fonction cercle trace le cercle de centre (x, y) et de rayon r. Courte explication du code :

- `numpy` est un module très pratique pour faire des mathématiques, en particulier de l'algèbre linéaire. La fonction `np.linspace(a,b,n)` permet de construire un tableau de n réels régulièrement espacés dans l'intervalle $[a, b]$. Les fonctions `np.cos` et `np.sin` sont les fonctions trigonométriques habituelles.
- `matplotlib` est un module de tracé. `plt.plot(X,Y)` prend en entrée deux listes de même taille `X` et `Y`, et relie les points (x_i, y_i) par des segments. Pour tracer une courbe (ici un cercle), il suffit de prendre des points de la courbe suffisamment rapprochés.
- Pour tracer plusieurs graphiques sur une même image (ce qu'on va faire ici), il suffit de n'afficher l'image (via `plt.show()`), qu'après avoir tracé toutes les courbes. Essayez par exemple le code suivant :

```
for i in range(1,10):
    cercle(0,0,i)
plt.show() #une cible !
```

Écrire deux fonctions `bulles1(n)` et `bulles2(n)` permettant d'obtenir des figures similaires à celles ci-dessous (qui résultent des appels `bulles1(5)` et `bulles2(5)`).





Le plus gros cercle est celui de centre 0 et de rayon 1. Les autres ont des rayons 1/2, 1/4, etc... et son tangents à ceux précédemment construits. On utilisera à chaque fois des fonctions auxiliaires récursives. *Indications* : pour `bulles1`, utiliser une fonction auxiliaire `aux(x,y,r,n)`. Pour `bulles2`, rajouter un paramètre indiquant « la direction d'où on vient ». Voici le début de `bulles1` :

```
def bulles1(n):
    plt.figure() #on réinitialise une figure
    plt.axis("equal") #on impose que les graduations sur les axes soient égales, c'est plus joli
    def aux(x,y,n,r):
       cercle(x,y,r)
       if n>0:
           [effectuer deux appels récursifs !]
    aux(0,0,n,1)
    plt.show()
```

Exercice 6. *Indices de sous-séquences.* Une chaîne $t = t_0, \dots, t_{k-1}$ est une *sous-séquence* d'une chaîne $s = s_0, \dots, s_{n-1}$ si les caractères de t apparaissent dans s dans le même ordre, mais pas nécessairement de manière contigüe. Par exemple, « abcd » et « aaaa » sont des sous-séquences de « abracadabra », alors que « dcb » n'en est pas une. On veut ici calculer *toutes* les possibilités d'apparition d'une chaîne comme sous-séquence d'une autre. Écrire une fonction récursive `indices_seq(s,t)` prenant en entrée deux telles chaînes, et renvoyant une liste de listes d'indices. Une liste d'indices $i_0, \dots, i_{k-1}$ sera telle que $t_j = s_{i_j}$ pour tout $j \in [\![0, k-1]\!]$.

```
>>> indices_seq("abracadabra", "aaaa")
[[0, 3, 5, 7], [0, 3, 5, 10], [0, 3, 7, 10], [0, 5, 7, 10], [3, 5, 7, 10]]
>>> indices_seq("abracadabra", "abcd")
[[0, 1, 4, 6]]
>>> indices_seq("abracadabra", "dcb")
[]
>>> indices_seq("abracadabra", "ab")
[[0, 1], [0, 8], [3, 8], [5, 8], [7, 8]]
```

Indications : prendre le cas où `t` est de taille 1 comme cas de base. Sinon, voir quels peuvent être les indices dans `s` où placer le dernier caractère de `t`, et voir (via un appel récursif) pour chaque indice possible i, où placer les autres caractères de `t` dans `s[:i]` (portion de s jusqu'à l'indice i exclu). Procéder par concaténation de listes.
Remarque : attention à ne pas prendre de trop grosses chaînes, puisque la complexité est potentiellement élevée !