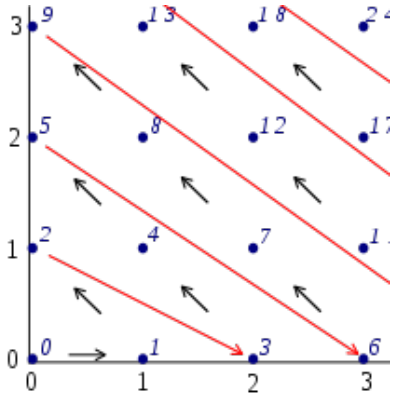
Récursivité: TP

Exercice 1. *Diagonale de Cantor*. On numérote les couples d'entiers naturels de la façon suivante.



Écrire une fonction récursive `enum(x,y)` associant à un couple d'entiers son numéro dans l'énumération ci-dessus. Réciproquement, écrire une fonction récursive `couple` renvoyant le couple associé à un entier.

```
>>> enum(1,2)
8
>>> couple(8)
(1, 2)
```

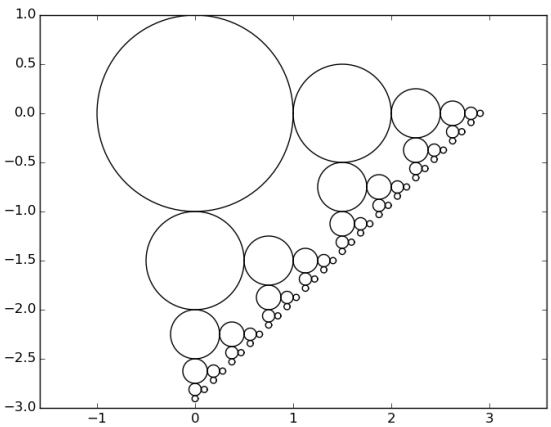
Remarque : ne cherchez pas une formule explicite : contentez vous de suivre les flèches !

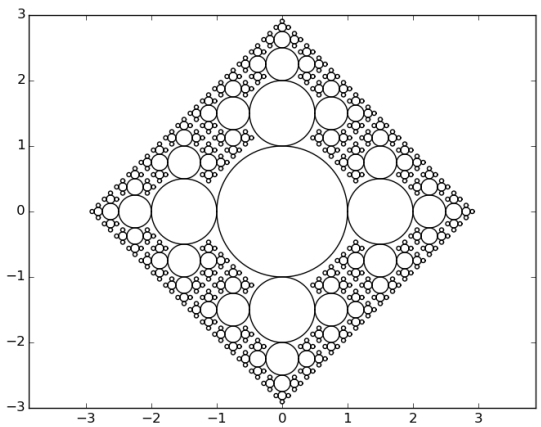
Exercice 2. *Des fractales*. Télécharger le fichier joint sur mon site. Vous y trouverez le code ci-dessous :

```
import matplotlib.pyplot as plt
import numpy as np

def cercle(x,y,r):
    theta = np.linspace(0, 2*np.pi, 100)
    X = r*np.cos(theta)+x
    Y = r*np.sin(theta)+y
    plt.plot(X,Y)
```

La fonction cercle trace le cercle de centre (x, y) et de rayon r. Remarquez que `np.cos` et `np.sin` sont vectorielles : lorsqu'on les applique à un tableau numpy, on calcule un tableau de même taille contenant les images par ces fonctions des éléments du tableau initial. Remarquez aussi qu'on peut facilement multiplier toutes les entrées d'un tableau numpy par une valeur (ici r) ou toutes les translater d'une constante (x ou y ici) pour obtenir de nouveaux tableaux. 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)`).

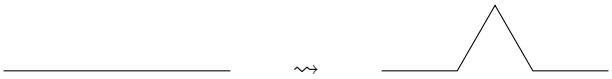




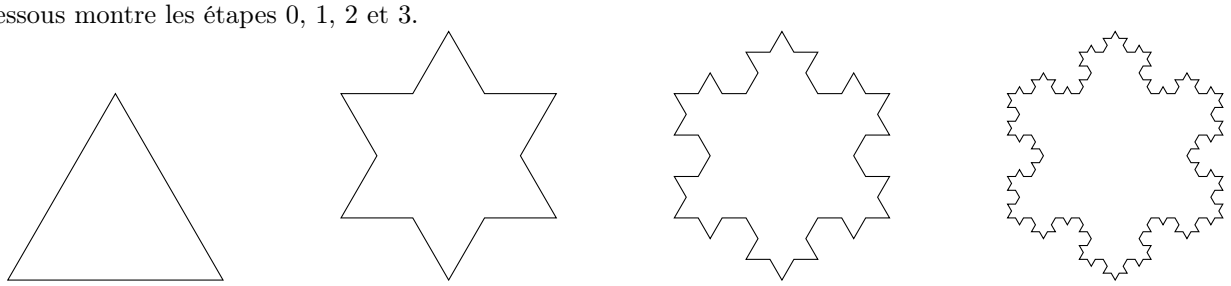
Le cas de base ($n = 0$) est celui du cercle unité. On utilisera à chaque fois des fonctions auxiliaires récursives. Indication : pour `bulles1`, utiliser une fonction auxiliaire `aux(x,y,r,n)`. Pour `bulles2`, rajouter un paramètre indiquant « la direction d'où on vient ». Voici le début de `bulles1` :

```
def bulles1(n):
    plt.figure()
    plt.axis("equal")
    def aux(x,y,n,r):
       [...]
    aux(0,0,n,1)
    plt.show()
```

Exercice 3. *Flocon de Von Koch*. Le but de cet exercice est de tracer une ligne brisée qui s'approche de l'objet fractal appelé le Flocon de Von Koch. La figure ci-dessous montre l'étape élémentaire de tracé de la ligne brisée : on remplace un segment de droite par le même segment, avec un triangle équilatéral qui a « poussé » du milieu vers l'extérieur.



Pour tracer le flocon, on part d'un triangle équilatéral, et on applique cette transformation sur ses trois côtés. On obtient ainsi une ligne brisée constituée de 12 segments. On réitère le procédé sur ces 12 segments, etc... La figure ci-dessous montre les étapes 0, 1, 2 et 3.



Étape 0 Étape 1 Étape 2 Étape 3

Le flocon de Von Koch n'est autre que la « figure limite » (dans un sens mathématique bien défini) obtenue après une infinité d'itérations. On se propose d'écrire une fonction récursive en Python, permettant de tracer la figure obtenue après n itérations, en partant d'un triangle initial de côté fixé. La fonction `trace(a,b)` (dans le fichier joint) suivante prend en entrée deux tableaux Numpy à deux éléments (symbolisant deux points du plan) et trace le segment les reliant :

```
def trace(a,b):
    x0,y0=a
    x1,y1=b
    plt.plot([x0,x1],[y0,y1])
```

La fonction `flocon(n)` suivante trace le flocon, à partir d'une fonction `koch(n,a,b)` qui travaille sur un segment. Écrire cette fonction `koch`.

```
def flocon(n):
    plt.axis("equal")
    plt.axis(draw=None)
    a=np.array([0,0])
    b=np.array([1,0])
    c=np.array([0.5,np.sqrt(3)/2])
    koch(n,a,c)
    koch(n,c,b)
    koch(n,b,a)
    plt.show()
```

Exercice 4. *Multiplication de polynômes*. On considère dans cet exercice qu'un polynôme $P = \sum_{k=0}^{n-1}$ est représenté sous la forme d'une liste $[p_0, p_1, \ldots, p_{n-1}]$.

1. Écrire des fonctions `add(P,Q)`, `sous(P,Q)`, `mult_scal(P,a)` et `mult_mon(P,k)` de soustraction et d'addition de polynômes, ainsi que de multiplication avec un scalaire et de multiplication par un monôme (la dernière fonction

renvoie la liste associée à $X^k \times P$. On fera attention pour les deux premières, dans le cas où les polynômes n'ont pas la même taille. La taille du polynôme obtenu est le maximum des deux tailles des polynômes initiaux, on ne fera pas attention aux cas où les degrés et les coefficients dominants sont les mêmes (notamment) ce qui pourrait permettre d'enlever des zéros.

2. En déduire une implémentation `mult(P,Q)` de multiplication de deux polynômes.

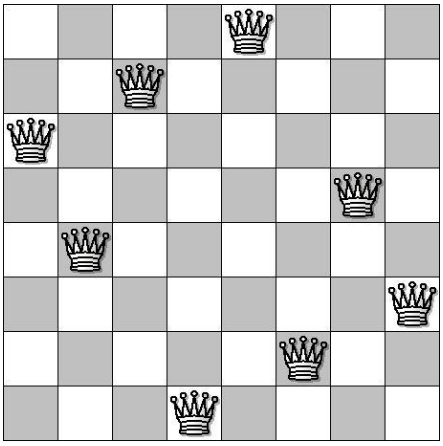
3. On rappelle l'algorithme de Karatsuba, fonctionnant sur des polynômes ayant même degré : l'implémenter.

Algorithme 1 : L'algorithme de Karatsuba

Entrées : Deux polynômes P et Q de même taille n.
Sortie : Le produit $P \times Q$.
si $n = 1$ **alors**
  retourner $([p_0 q_0])$
sinon
  $m = \lfloor n/2 \rfloor$;
  Décomposer $P = P_0 + X^m P_1, Q = Q_0 + X^m Q_1$;
  T_0 =Karatsuba(P_0, Q_0);
  T_1 =Karatsuba(P_1, Q_1);
  T_2 =Karatsuba$(P_0 + P_1, Q_0 + Q_1)$;
  retourner $T_0 + X^m(T_2 - T_1 - T_0) + X^{2m} T_1$

4. Effectuer une comparaison du temps d'exécution de ces deux algorithmes pour multiplier deux polynômes de taille 1000 ou 10000 générés aléatoirement. On utilisera la fonction `randint(a,b)` du module `random` pour générer aléatoirement des polynômes et `clock()` du module `time` pour mesurer des temps d'exécution.

Exercice 5. *Le problème des dames : résolution par backtracking.* On rappelle qu'au jeu d'échecs, on travaille sur un échiquier de taille 8×8. Deux dames posées sur l'échiquier sont « en prise » si elles se trouvent sur la même colonne, la même ligne, ou la même diagonale (au sens large). Le problème des 8 dames consiste à placer 8 dames sur un échiquier de sorte que deux d'entre elles ne soient jamais en prise. La figure qui suit est un exemple de solution.



Ce problème se généralise naturellement à un échiquier $n \times n$. Les questions qui suivent donnent une stratégie pour calculer (et afficher le nombre de solutions) au problème, pour des n raisonnables (la complexité est exponentielle, mais c'est un peu la faute de ce qu'on cherche à calculer !).

On représente une grille avec des dames posées dessus comme un tableau (liste de listes) `T` de booléens : lignes et colonnes sont numérotées de haut en bas et de gauche à droite, et un `True` en `T[i][j]` indique qu'une dame est posée dans la case (i, j). On va essayer de calculer des solutions en suivant le principe du « backtracking » : cela consiste à tester toutes les possibilités, mais intelligemment : on remplit la grille de haut en bas et de gauche à droite, quite à revenir en arrière. Vous trouverez sur mon site web trois fonctions :

— `case_suivante(n,i,j)` prenant en entrée les indices d'une case (i, j), et renvoyant la case suivante (celle située immédiatement à sa droite si elle existe, ou $(i + 1, 0)$ sinon ;

— `init(n)` permettant de générer une grille de $n \times n$ `False` ;

— `imprimer(T)` permettant d'imprimer une solution à l'écran, à partir du tableau de booléens.

1. Écrire une fonction `en_prise(i,j,T)` prenant en entrée un tableau de booléens représentant un échiquier $n \times n$ (n est accessible par `len(T)`) et testant si la case (i, j) est en prise : ceci signifie qu'il existe un `True` sur une case située à gauche, au dessus, au dessus à gauche en diagonal, ou au dessus à droite en diagonal (on ne considérera pas les cases situées à droite ou en dessous). Attention à ne pas faire de dépassement d'indice sur les cases !

2. La fonction suivante, à compléter, donne le nombre de solutions (et affiche à l'écran celles-ci) pour le placement de n dames sur un échiquier $n \times n$. Elle fait usage d'une fonction `aux(i,j,m)` récursive.

```
def calcule_sol(n):
    T=init(n)
    def aux(i,j,m):
        """ (i,j) une case, m le nombre de reines placées dans les cases avant (i,j) """
        if m<i:
            return 0 #dans ce cas on a placé moins d'une reine par ligne au dessus: ce n'est pas bon.
        if i==n and m==n: #on  est arrivé au bout ! on a donc une solution.
            imprimer(T)
            return 1
        i2,j2=suivante(n,i,j)
        [...]
    return aux(0,0,0)
```

Indication : si la case (i, j) n'est pas en prise, on fera deux appels récursifs à `aux(i2,j2,...)` : l'un pour lequel on a mis `True` en case (i, j), l'autre ou on a mis `False`. Si elle est en prise, on ne fait qu'un seul appel.

```
>>> calcule_sol(4)

oxoo
ooox
xooo
ooxo

ooxo
xooo
ooox
oxoo
2
```

3. Vérifiez que vous trouvez 92 solutions pour $n = 8$.