Corrigé du TP n°5 - Récursivité - Corrigé

1. Premiers pas avec la récursivité

Faisons nos premiers pas avec la récursivité :

Q1 : En s'inspirant de la version récursive de la fonction factorielle présentée dans le cours, écrire une fonction récursive `Deuxpuissance(n)` qui renvoie la valeur de 2^n pour $n \in \mathbb{N}$. On se basera sur la relation de récurrence :

$$\begin{cases} 2^0 & = 1 \\ 2^n & = 2 \times 2^{n-1} \text{ pour } n \in \mathbb{N}^* \end{cases}$$

Q2 : **a)** On considère la fonction récursive suivante :

```
1 def triangle(n):
2     if n==0:
3         return None
4     else:
5         print(n*'*')
6         triangle(n-1)
```

Tester l'instruction `triangle(n)` avec différentes valeurs de `n` (par exemple 5,6,10) puis écrire une fonction `triangle2(n)` qui affiche le même triangle mais dans l'autre sens :

Exemple de comportement

```
>>> triangle2(5)
*
**
***
****
*****
```

b) Écrire une fonction `triangle3(n)` qui affiche les 2 triangles des instructions `triangle(n)` et `triangle2(n)` l'un au dessus de l'autre (sans utilser les fonctions `triangle(n)` et `triangle2(n)` bien-sûr).

Exemple de comportement

```
>>> triangle3(4)
****
***
**
*
*
**
```

```
***
****
```

Correction.

Q1 :

```
1 def Deuxpuissance(n):
2     if n==0:
3         return 1
4     else:
5         return 2*Deuxpuissance(n-1)
```

Q2 : a)

```
1 def triangle2(n):
2     if n==0:
3         return None
4     else:
5         triangle2(n-1)
6         print(n*'*')
```

b)

```
1 def triangle3(n):
2     if n==0:
3         return None
4     else:
5         print(n*'*')
6         triangle3(n-1)
7         print(n*'*')
```

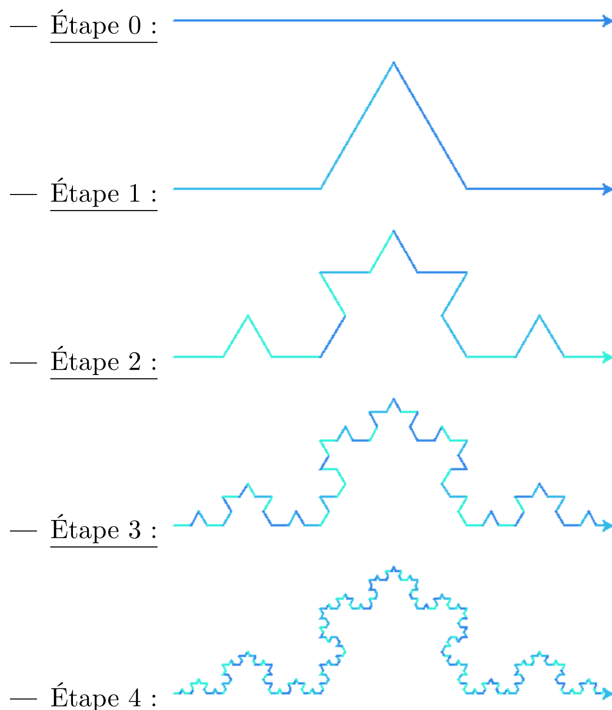
2. La tortue récursive et fractale

Dans cette partie, nous allons utiliser le module `turtle` pour réaliser de jolis dessins qui se prêtent très bien à l'utilisation de la récursivité. Commençons par importer le module :

```
1 from turtle import *
```

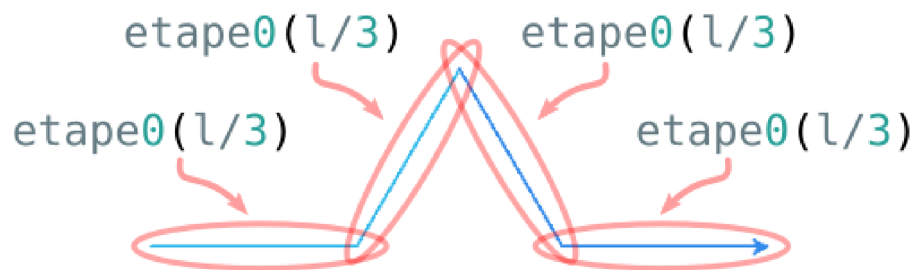
Q1 : La courbe de Von Koch : la courbe de Von Koch est une courbe fractale - une courbe dont le motif se répète "indéfiniment" lorsqu'on zoome dessus. Nous allons écrire une fonction qui réalise

les étapes de constructions de cette fractale de Von Koch.
Voici ce que nous allons obtenir (en monochrome pour commencer) :



Dans ces images, les longueurs de chaque segment sont égaux et chaque "triangle" est équilatéral.

a)
- Écrire une fonction `etape0(l)` qui trace l'étape 0 de la courbe de Von Koch de taille `l` pixels.
- Écrire une fonction `etape1(l)` qui trace l'étape 1 de la courbe de Von Koch de taille `l` pixels (il s'agit de la longueur entre le point de départ et le point d'arrivée, pas la longueur totale de la courbe) et qui utilise la fonction `etape0` précédente : on remarquera que la fonction `etape1(l)` appellera quatre fois `etape0(l/3)`

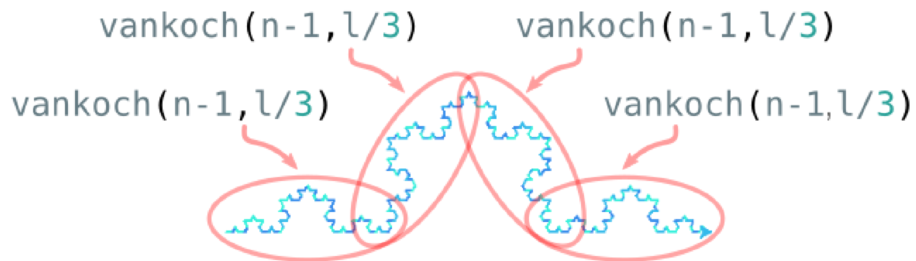


Étape 1 en fonction de l'étape 0.

b) On décrit le principe récursif - multiple - de réalisation des étapes de la courbe de Von Koch :

> - **Initialisation :** étape 0 : on dessine un trait de la longueur donné en argument
> - **Récursion :** étape $n \geq 1$: on suppose que la tortue sait tracer l'étape $n-1$ sur toute longueur. On veut tracer l'étape n sur une longueur ℓ, on réalise alors l'étape 1 mais, au lieu de réaliser l'étape 0 sur une longueur de $\ell/3$, on réalise

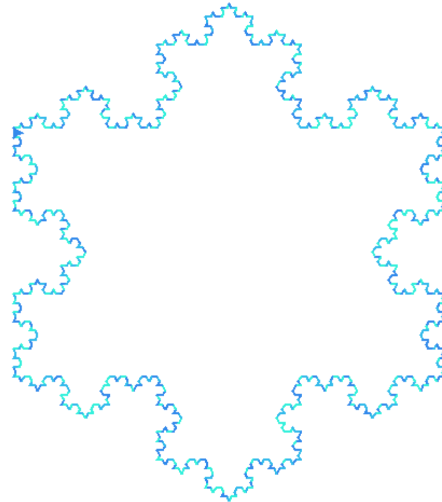
chaque fois (donc 4 fois) l'étape $n-1$ sur une longueur $\ell/3$.



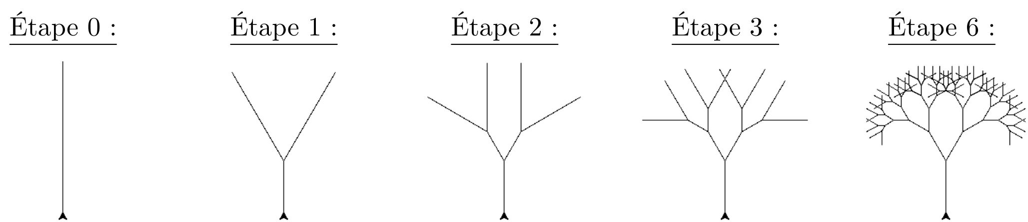
`vonkoch(n,l)` en fonction de `vonkoch(n-1,l)` pour n≥ 1.

Écrire une fonction récursive `vonkoch(n,l)` qui trace l'étape n de la courbe de Von Koch. On exécutera l'instruction `speed(0)` avant tout test pour s'assurer un tracé le plus rapide possible.

c) Utiliser la fonction `vonkoch(n,l)` plusieurs fois pour tracer le *flocon de Von Koch* à l'étape 4.



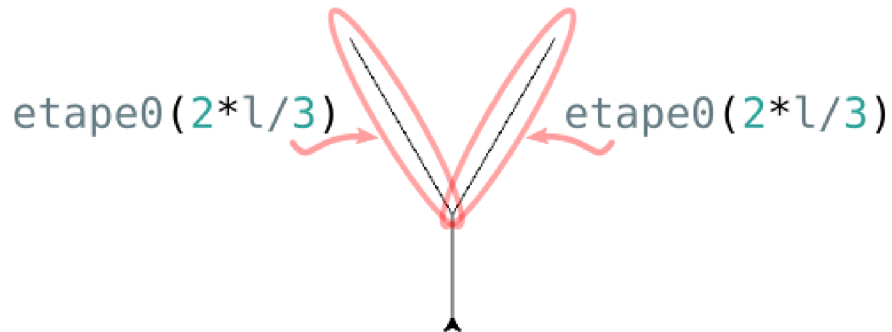
Q2 : **Un arbre !** En s'inpirant de ce que nous avons fait précédemment avec la courbe de Von Koch, nous allons faire tracer à la tortue les étapes d'un arbre fractale : Voici ce que nous allons obtenir (en monochrome pour commencer) :



Remarque importante : on tracera l'arbre à **l'horizontale** dans les questions suivantes. Pour le mettre à la verticale, on fera simplement `left(90);arbre(n,l,a)`.

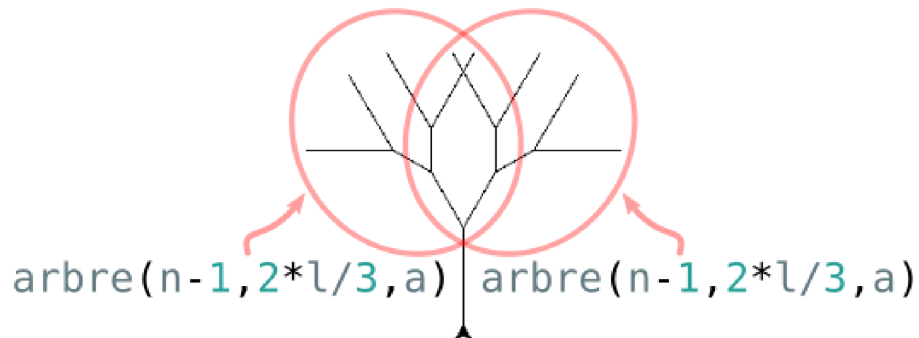
a) — Écrire une fonction `etape0_arbre(l)` qui trace l'étape 0 de l'arbre taille `l` pixels. **Attention : comme on peut le voir, la tortue revient à son point de départ à la fin !**

— Écrire une fonction `etape1_arbre(l,a)` qui trace l'étape 1 de l'arbre dont le tronc est de taille `l/3` ; les branches sont de taille `2*l/3` pixels et l'angle entre les deux branches est égal à `2*a` degrés et qui utilise la fonction `etape0_arbre` précedente : on remarquera que la fonction `etape1_arbre(l,a)` appellera deux fois `etape0_arbre(l/3)`



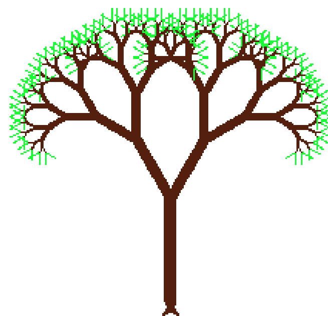
Étape 1 en fonction de l'étape 0.

b) Écrire une fonction récursive `arbre(n,l,a)` qui trace l'étape n de l'arbre fractal.

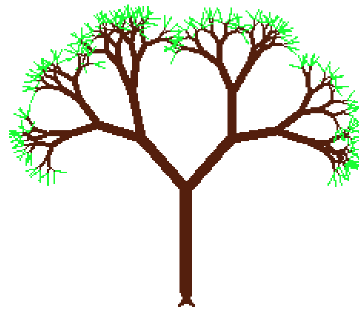


`arbre(n,l,a)` en fonction de `arbre(n-1,l,a)` pour `n`≥ 1.

c) Mettre de jolies couleurs avec la fonction `color` pour que l'arbre ressemble à un arbre tel que celui-ci :



d) Et en utilisant le module `random`, modifier la fonction `arbre` pour obtenir un rendu plus "naturel" !



Correction.

Q1 : **a)**

—

```
1 def etape0(l):
2     forward(l)
```

—

```
1 def etape1(l):
2     etape0(l/3)
3     left(60)
4     etape0(l/3)
5     right(120)
6     etape0(l/3)
7     left(60)
8     etape0(l/3)
```

b)

```
1  def vonkoch(n,l):
2      if n==0:
3          etape0(l)
4      else:
5          vonkoch(n-1,l/3)
6          left(60)
7          vonkoch(n-1,l/3)
8          right(120)
9          vonkoch(n-1,l/3)
10         left(60)
11         vonkoch(n-1,l/3)
```

c)

```
>>>speed(0)
>>>vonkoch(4,200)
>>>right(120)
>>>vonkoch(4,200)
>>>right(60)
>>>vonkoch(4,200)
```

Q2 : a) —

```
def etape0_arbre(l):
    forward(l)
    forward(-l)
```

—

```
def etape1_arbre(l,a):
    forward(l/3)
    left(a)
    etape0_arbre(2*l/3)
    right(2*a)
    etape0_arbre(2*l/3)
    left(a)
    forward(-l/3)
```

b)

```
def arbre(n,l,a):
    if n==0:
        etape0_arbre(l)
    else:
        forward(l/3)
        left(a)
        arbre(n,2*l/3,a)
        right(2*a)
        arbre(n,2*l/3,a)
        left(a)
        forward(-l/3)
```

c)

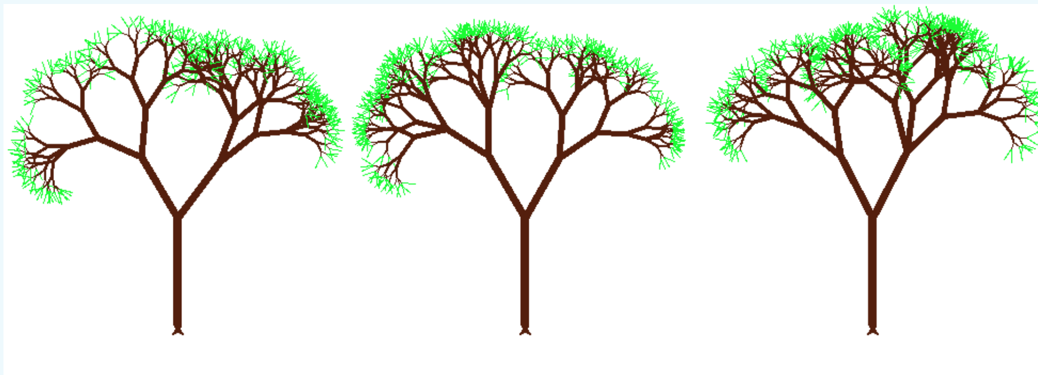
```python
def arbre_couleur(n,l,a):
    if n==0:
        color(0.1,0.99,0.2) #vert
        etape0_arbre(l)
        color(0.33,0.12,0.05) #marron
    else:
        color(0.33,0.12,0.05)
        width(n) #épaisseur du tracé
        forward(l/3)
        left(a)
        arbre_couleur(n,2*l/3,a)
        right(2*a)
        arbre_couleur(n,2*l/3,a)
        left(a)
        forward(-l/3)
```

d)

```python
import random as rd

def arbre_random(n,l,a):
    if n==0:
        color(0.1,0.99,0.2)
        etape0_arbre(l)
        color(0.33,0.12,0.05)
    else:
        color(0.33,0.12,0.05)
        width(n)
        forward(l/3)
        angle1=rd.random()*35+10 # angle entre 10 et 45 degrés
        angle2=rd.random()*35+10
        left(-angle1)
        arbre_random(n-1,l*2/3,angle1)
        left(angle1+angle2)
        arbre_random(n-1,l*2/3,angle2)
        left(-angle2)
        forward(-l/3)
```

Et pour la route, un peu plus complexe :

```python
import random as rd

def sommetot(k,tot):
    L=[0.7+rd.random()*0.3 for i in range(k)]
    S=sum(L)
    L=[tot*l/S for l in L]
    return L

def proba123(a,b):
    r=rd.random()
    if r<a:
        return 1
    if r>b:
        return 3
    else:
        return 2

def prodhasard(a):
    return (2*a*rd.random()+(1-a))*2/3

def arbre_joli(n,longueur):
    if n==0:
        color(0.1,0.99,0.2)
        etape0(l)
        color(0.33,0.12,0.05)
    else:
        color(0.33,0.12,0.05)
        width(n)
        forward(l/3)
        k=proba123(0,0.9)
        angletot = rd.random()*50+30
        semiangletot=angletot/2
        angles = sommetot(k,angletot)
        left(-angles[0])
        arbre_joli(n-1,l*prodhasard(0.05))
        left(angles[0])
        for i in range(1,k):
            left(angles[i])
            arbre_joli(n-1,l*prodhasard(0.05))
        left(-sum(angles[1:]))
        forward(-l/3)
```



Différents résultats pour les intructions `speed(0);left(90);arbre_joli(8,200)`.

3. Récursivité et suites récurrentes

Les fonctions récursives sont parfaitement adaptées pour calculer les termes d'une suite définie par récurrence.

Q1 : Considérons la suite suivante :

$$\begin{cases} u_0 &= 0 \\ u_n &= u_{n-1}^2 + 0.25 \text{ pour tout } n \in \mathbb{N}^* \end{cases}$$

a) Écrire une fonction récursive `u(n)` qui renvoie la valeur de u_n où `n` est un entier naturel.

b) Vers quelle valeur la suite semble-t-elle converger ? On pourra augmenter la taille de la pile d'appels pour en avoir le coeur net.

c) Écrire une fonction récursive `m(n,c)` qui renvoie la valeur de u_n où la suite (u_n) a été modifiée : on remplace le "0.25" par `c`.

d) Tester la suite reste bornée ou non pour différentes valeurs de c : $c = 0.26$, $c = -1$, $c = -0.1 + 0.8i$

e) Taper "ensemble de Mandelbrot" dans un moteur de recherche et essayer, à la maison, de coder un programme qui affiche (une approximation) de cet ensemble.

Q2 : La suite de Syracuse d'un nombre entier p est definie de la manière suivante :

$$s_0 = p \qquad s_{n+1} = \begin{cases} \frac{s_n}{2} & \text{si } s_n \text{ est pair} \\ 3s_n + 1 & \text{si } s_n \text{ est impair} \end{cases} \text{ pour } n \in \mathbb{N}$$

Écrire une fonction `syracuse(n,p)` qui renvoie la valeur de s_n de la suite de Syracuse du nombre entier p.

Correction.

Q1 : a)

```
1 def u(n):
2     if n==0:
3         return 0
4     else:
5         return u(n-1)**2+0.25
```

b) Vers $0,5$ semble-t-il...

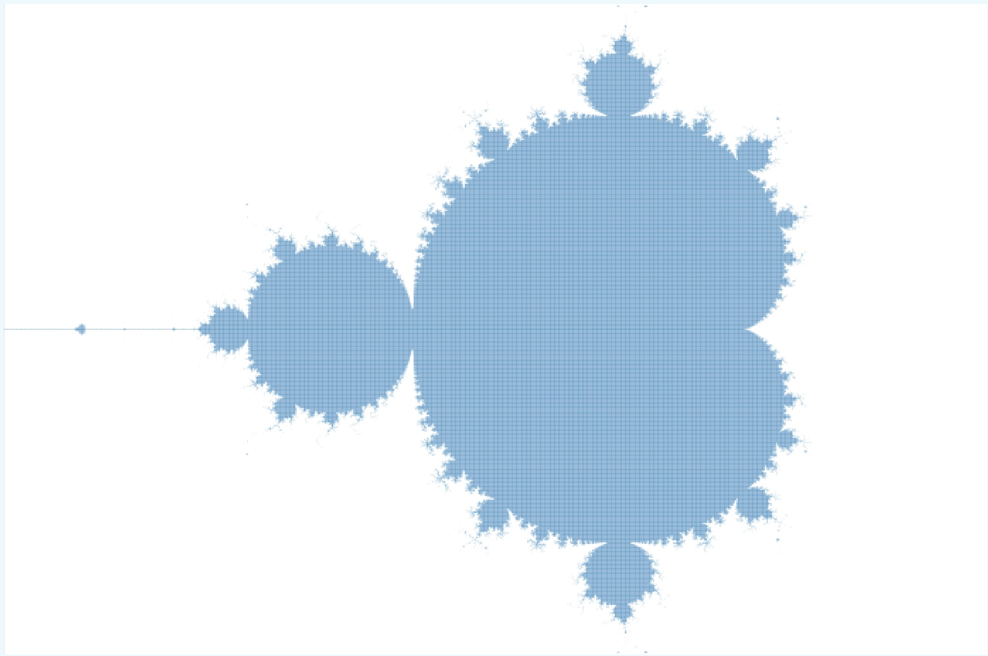
c)

```
1 def m(n,c):
2     if n==0:
3         return 0
4     else:
5         return m(n-1,c)**2+c
```

d) Pour $c = 0.26$ diverge vers $+\infty$, $c = -1$ diverge mais reste bornée, $c = -0.1 + 0.8i$ diverge mais reste bornée.

e) Une première façon simple en plaçant les point avec la fonction scatter de matplotlib mais pas très efficace :

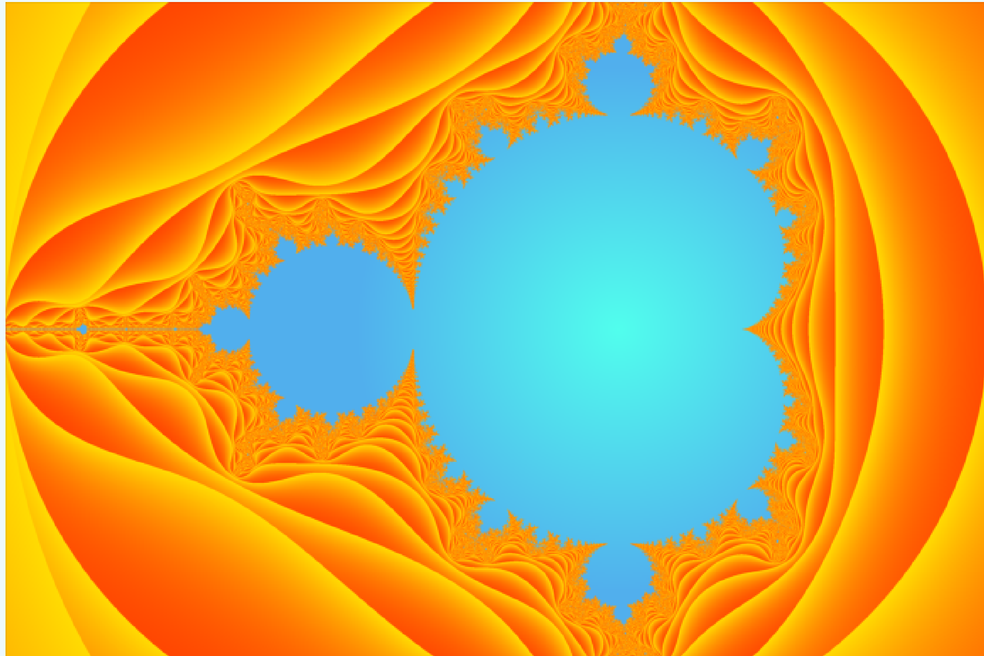
```python
import numpy as np
import matplotlib.pyplot as plt

def converge(c,nmax):
    z=0
    n=0
    while abs(z)<=2 and n<nmax:
        z=z**2+c
        n+=1
    return n==nmax

def affichage_mandelbrot(nmax,precision):
    """ Affiche les points appartenant à M pris sur un quadrillage de taille "precision fois precision" de [-2,1]x[-1,1]  """
    nb_points_unite = int(1/precision) #nombres de points à prendre dans l'intervalle [0,1]
    X=np.linspace(-2,1,3*nb_points_unite) #liste des abscisses de points à tester
    Y=np.linspace(0,1,nb_points_unite) #liste des ordonnées de points à tester
    XM=[] #future liste des abscisses des points dans M
    YM=[] #future liste des ordonnées des points dans M
    for x in X:
        for y in Y:
            c = x+y*1j #affixe du point C(c) à tester
            if converge(c,nmax): # si la suite converge le point C(c) est dans M
                XM.append(x)
                YM.append(y) #on ajoute abs et ord de C(c)
                if x!=0: # si C(c) est dans M, D(cbarre) aussi (Q3b)
                    XM.append(x)
                    YM.append(-y)
    plt.figure(figsize=(9,6)) #on initialise le graphique
    plt.xlim(-2,1)
    plt.ylim(-1,1) #on donne les coordonnées extrémales
    plt.scatter(XM,YM,s=10000*precision**2,linewidths=0) #on place les points sur le graphique
    plt.show() #on ouvre la fenêtre du graphique
```



Résultat pour `affichage_mandelbrot(50,10**-3)`.

Version un peu plus efficace utilisant la création d'une image :

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def modulecarre(z):
5     return z.real**2+z.imag**2
6
7 def convergence(c,iteration_max):
8     z=c
9     n=0
10    while modulecarre(z)<=4 and n<iteration_max: #tant que |z|<=2 et
          qu'on a pas fait trop d'itération
11        z=z**2+c
12        n+=1
13    return modulecarre(z)
14
15 def mandelbrot(iteration_max,precision):
16    decoupage_unite = int(1/precision)
17    y=2*decoupage_unite
18    x=3*decoupage_unite
19    M=np.zeros((y,x,3),dtype='uint8')
20    place_zero=(2*(x//3)-1,y//2-1)
21    def absc(i):
22        return precision*(i-place_zero[0])
23    def ordo(j):
24        return precision*(place_zero[1]-j)
25    for i in range(x):
26        for j in range(y//2):
27            xi=absc(i)
28            yj=ordo(j)
29            c=xi+yj*1j
30            p=(xi-0.25)**2+yj**2
31            est_dans_cardio = xi<p**0.5-2*p+0.25
32            est_dans_disque = (xi+1)**2+yj**2<1/16
33            if est_dans_cardio or est_dans_disque:
34                mzc = 0
35            else:
36                mzc=convergence(c,iteration_max)
37            if mzc<=4:
38                rouge = 81
39                vert = 57+int(198*(1-min(abs(c+0.125),0.8)/2)) #juste
                     pour faire un joli dégradé radial dans M (
                    mathématiquement inutile !)
40                bleu = 237
41            else: # ici |z|>2 et on met jaune si |z| proche de 2 et
                  plutôt rouge/orange si |z| très grand
42                rouge = 255
43                vert = 50+int(170*4/mzc)
44                bleu = 2
45            M[j,i]=[rouge,vert,bleu]
46    for i in range(x):
47        for j in range(y//2,y):
48            M[j,i]=M[y-1-j,i]
49    return M
50
51 def afficher(M):
52    plt.figure(figsize=(3*3,2*3))
53    plt.imshow(M)
54    plt.show()
```



Résultat pour M=mandelbrot(50,10**-3);afficher(M).

Q2 :

```
def syracuse(n,p):
    if n==0:
        return p
    else:
        s=syracuse(n-1,p)
        if s%2==0:
            return s//2
        else:
            return 3*s+1
```