Séquence 6 – La récursivité

Objectifs

1. Écrire un programme récursif
2. Analyser le fonctionnement d'un programme récursif
3. Savoir répondre aux effets de bord non désirés

Cette séquence s'appuie sur :

- https://pixees.fr/informatiquelycee/n_site/nsi_term_calcu.html
- https://pixees.fr/informatiquelycee/n_site/nsi_term_paraProg_fct.html
- https://pixees.fr/informatiquelycee/n_site/nsi_term_fctRec.html


1 Introduction

A faire vous même 1.

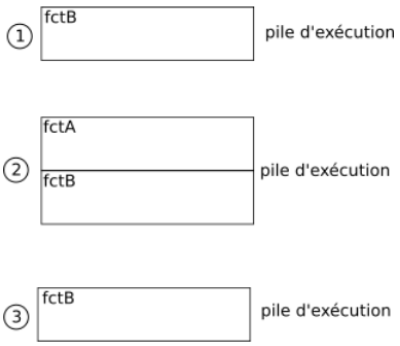
Analysez puis testez le programme suivant :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def fctA():
    print ("Début fonction fctA")
    i=0
    while i<5:
        print(f"fctA {i}")
        i = i + 1
    print ("Fin fonction fctA")

def fctB():
    print ("Début fonction fctB")
    i=0
    while i<5:
        if i==3:
            fctA()
            print("Retour à la fonction fctB")
        print(f"fctB {i}")
        i = i + 1
    print ("Fin fonction fctB")

fctB()
```

Vous devriez obtenir l'enchainement suivant :

```
Début fonction fctB
fctB 0
fctB 1
fctB 2
Début fonction fctA
fctA 0
fctA 1
fctA 2
fctA 3
fctA 4
Fin fonction fctA
Retour à la fonction fctB
fctB 3
fctB 4
Fin fonction fctB
```

Pour gérer ces fonctions qui appellent d'autres fonctions, le système utilise une "pile d'exécution". Une pile d'exécution permet d'enregistrer des informations sur les fonctions en cours d'exécution dans un programme. On parle de pile, car les exécutions successives "s'empilent" les unes sur les autres. Pour le programme étudié ci-dessus, nous obtenons le schéma suivant :

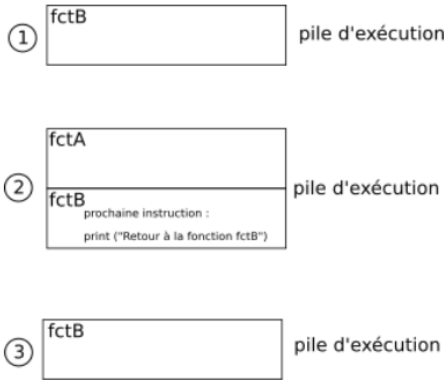


Nous pouvons "découper" l'exécution de ce programme en 3 parties :

1. la fonction `fctB` s'exécute jusqu'à l'appel de la fonction `fctA`
2. l'exécution de la `fctB` est mise en "pause" pendant l'exécution de la fonction `fctA`
3. une fois que l'exécution de `fctA` est terminée, on termine l'exécution de la fonction `fctB`

Il est important de bien comprendre que la fonction située au sommet de la pile d'exécution est en cours d'exécution. Toutes les fonctions situées "en dessous" sont mises en pause jusqu'au moment où elles se retrouveront au sommet de la pile. Quand une fonction termine son exécution, elle est automatiquement retirée du sommet de la pile (on dit que la fonction est dépilée).

La pile d'exécution permet de retenir la prochaine instruction à exécuter au moment où une fonction sera sortie de son "état de pause" (qu'elle se retrouvera au sommet de la pile d'exécution) :



En fait, c'est l'adresse mémoire de la prochaine instruction machine à exécuter qui est conservée dans la pile d'exécution

Dans l'exemple ci-dessus, on retrouve une variable i dans les deux fonctions : `fctA` et `fctB`. La variable `i` présente dans la fonction `fctA` n'a rien à voir avec la variable i présente dans la fonction `fctB` (elles portent le même nom, mais elles représentent 2 adresses mémoires différentes). Il est très important de bien comprendre que les variables créées dans une fonction ne "sortent" pas de la fonction : chaque fonction possède sa propre liste de variable, comme déjà dit ci-dessus la variable i de la fonction `fctB` est différente de la variable i de la fonction `fctA`.

La pile d'exécution conserve une "trace" des valeurs des variables lorsqu'une autre fonction est exécutée. Par exemple la valeur de `i fctB` est conservée au moment de l'exécution de `fctA`. Quand l'exécution de `fctA` se termine est que l'exécution de `fctB` "reprend", la valeur référencée par i `fctB` a été "conservée" (voilà pourquoi on reprend l'exécution de `fctB` avec un `fctB 3`.

2 Une fonction récursive, une fonction qui s'appelle elle-même

2.1 Premier exemple

A faire vous même 2.

Analysez puis testez le programme suivant :

```
def fctA():
    print ("Hello")
    fctA()
fctA()
```

Comme vous pouvez le constater, nous avons une erreur dans la console Python :

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Dans le cas où une fonction s'appelle elle-même (fonction récursive), on retrouve le même système de pile d'exécution. Dans l'exemple traité ci-dessus, les appels s'enchaînent sans rien pour mettre un terme à cet enchaînement, la taille de la pile d'exécution augmente sans cesse (aucune fonction ne termine son exécution, nous n'avons pas de "dépilement" juste des "empilements"). Le système interrompt le programme en générant une erreur quand la pile d'exécution dépasse une certaine taille.

.....

A faire vous même 3.

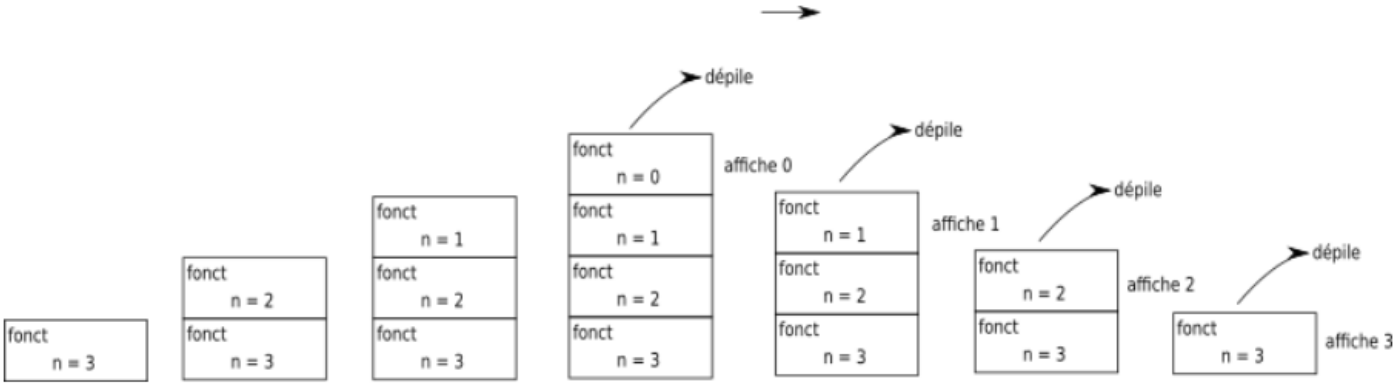
Essayez de prévoir le résultat de l'exécution du programme ci-dessus. Vérifiez votre hypothèse en exécutant le programme.

```
def fonct(n):
    if n>0:
        fonct(n-1)
    print(n)

fonct(3)
```

Essayons de comprendre en détail ce qui se passe dans le programme ci-dessus :

- 1er appel de `fonct` avec le paramètre n = 3 ; n > 0 donc appel de `fonct` avec le paramètre n = 2
- 2e appel de `fonct` avec le paramètre n = 2 ; n > 0 donc appel de `fonct` avec le paramètre n = 1
- 3e appel de `fonct` avec le paramètre n = 1 ; n > 0 donc appel de `fonct` avec le paramètre n = 0
- 4e appel de `fonct` avec le paramètre n = 0 ; n = 0 donc on exécute l'instruction `print(n)` => affichage : 0
- on "dépile" (3e appel, n = 1) : on exécute l'instruction `print(n)` => affichage : 1
- on "dépile" (2e appel, n = 2) : on exécute l'instruction `print(n)` => affichage : 2
- on "dépile" (1er appel, n = 3) : on exécute l'instruction `print(n)` => affichage : 3

Voici un schéma expliquant le processus en termes de pile d'exécution :



Il ne faut jamais perdre de vu qu'à chaque nouvel appel de la fonction fonct le paramètre n est différent.

2.2 Deuxième exemple : Fonction factorielle

Nous allons étudier le calcul de la factorielle grâce à une fonction récursive. D'après Wikipédia : "En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n".
Par exemple :

- la factorielle de 3 est : $3\times2\times1=6$
- la factorielle de 4 est $4\times3\times2\times1=4\times6=24$
- la factorielle de 5 est $5\times4\times3\times2\times1=5\times24=120$
- ...

Si on note la factorielle de n par n!, on a :

- $0!=1$ (par définition)
- $n!=n\times(n-1)\times(n-2)\times(n-3)\times...\times2\times1$

A faire vous même 4.

- Complétez la fonction `factorielClassique` en utilisant une bouche bornée :

```
def factorielClassique(n) :
    """
    >>>  factorielClassique(5)
    120
    >>>  factorielClassique(0)
    1
    """
    ...
    return ...
```

- Testez dans la console

Si on étudie de près les factoriels, on remarque que :

- Pour tout entier n > 0, $n!=n\times(n-1)!$

Nous allons utiliser cette définition de la factorielle pour définir notre fonction récursive (nous allons utiliser le fait que la factorielle de n dépend de la factorielle de n-1 et que 0! = 1)

A faire vous même 5.

Analysez puis testez la fonction `factorielRecursive` à l'aide de la console Python :

```
def factorielRecursive(n) :
    if n > 0 :
        return n*factorielRecursive(n-1)
    else :
        return 1
```

Comme vous pouvez le constater, la fonction `factorielRecursive` est structurée de la même manière que la définition mathématique vu ci-dessus :

- dans le cas où n = 0 la fonction renvoie 1 (0! = 1)
- dans le cas où n > 0 la fonction renvoie n*fact(n-1) (n! = n x (n – 1)!)

2.3 Troisième exemple : Suite de Fibonacci

L'utilisation des fonctions récursives est souvent liée à la notion de récurrence en mathématiques :
En mathématiques une suite définie par récurrence est une suite définie par son premier terme et par une relation de récurrence, qui définit chaque terme à partir du précédent ou des précédents lorsqu'ils existent.
Prenons l'exemple de la suite de Fibonacci qui est définie par :

- $U_0=0$ et $U_1=1$
- Et par la relation de récurrence suivante avec n entier et n > 1 :
  $U_n=U_{n-1}+U_{n-2}$

Ce qui nous donne pour les 6 premiers termes de la suite de Fibonacci :

- $U_0=0$
- $U_1=1$
- $U_2=U_1+U_0=1+0=1$
- $U_3=U_2+U_1=1+1=2$
- $U_4=U_3+U_2=2+1=3$
- $U_5=U_4+U_3=3+2=5$

A faire vous même 6.

- En vous aidant de ce qui a été fait pour la fonction `factorielRecursive`, écrivez une fonction récursive `fib` qui donnera le n^{ième} terme de la suite de Fibonacci. Cette fonction prendra en paramètre l'entier n.

Lire cours P. 38-39
L' ESSENTIEL :

.....

.....

.....

.....

.....

.....

.....

.....

P. 46 ex 1

P. 48 ex 4

P. 48 ex 5

3 Avantages/inconvénients de la récursivité

3.1 Avantage

Les algorithmes récursifs sont souvent beaucoup plus rapides que les algorithmes classiques.
Mesurons cela avec le module python `timeit` : https://docs.python.org/fr/3/library/timeit.html

A faire vous même 7. Travail préalable : Mesurer le temps d' exécution d' instructions python.

- Écrivez le script suivant dans un fichier `temp.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from math import sqrt
import timeit
def distanceDeuxPoints(xa, xb, ya=0, yb=0):
    return sqrt((xa-xb)**2+(ya-yb)**2)
if __name__=='__main__':
    tps = timeit.timeit("""\
distanceDeuxPoints(1,2,3,4)""", number = 1000000, globals=globals())
    print(f"Temps écoulé pour 1 000 000 exécution de la fonction : {tps} ")
```

- Lancez-le, ce script mesure le temps d' exécution de commandes python
- Observez bien le corps de programme

A faire vous même 8.

- Adaptez le script ci-dessus pour mesurer le temps de calcul pour les deux fonctions factoriel écrite avant.
  `factorielClassique` :
  `factorielRecursive` :
- Qu' en concluez-vous ?

.....

A faire vous même 9. Mettre les résultats sous forme de graphique.

Mieux encore, il est intéressant de voir l' évolution du temps de calcul en fonction du nombre à calculer

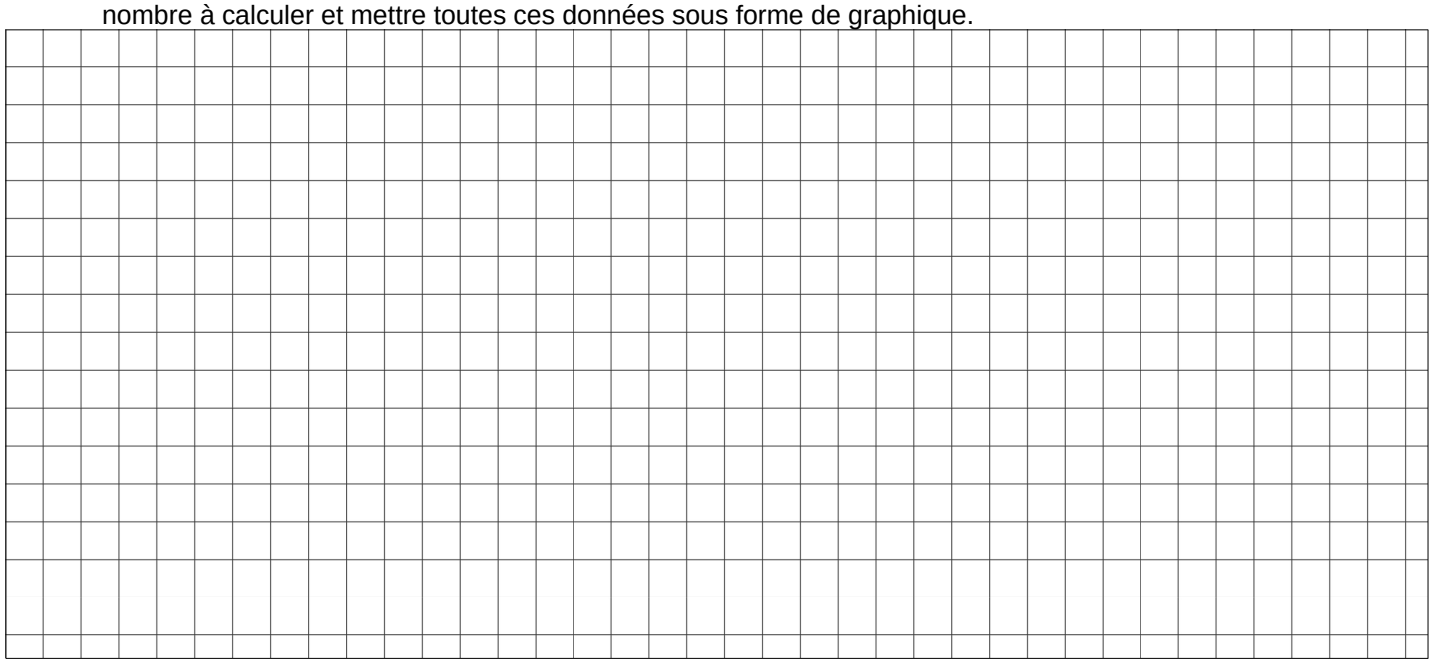
- Écrivez le script suivant dans un fichier `temp.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(2, 1, 1)
line = ax.plot([1,2,3,4], [10,20,40,80], color='blue')
line2 = ax.plot([1,2,3,4], [20,30,50,90], color='red')
fig.show()
```

- Lancez-le, ce script dessine un graphe avec deux courbes.
- Observez bien le corps de programme. Pour chaque courbe, on fournit la liste des x (abscisses) et la liste des y (ordonnées).

A faire vous même 10.

- Pour les deux fonctions factoriel écrites avant, adaptez le script ci-dessus pour mesurer le temps en fonction du nombre à calculer et mettre toutes ces données sous forme de graphique.



- Qu' en concluez-vous ?

.....

3.2 Inconvénient – Taille mémoire

Ce gain de temps significatif se paye en taille mémoire.
En effet, un algorithme récursif créé de nombreuses fonctions qui ne sont détruites que quand leur exécution est arrivé à son terme. Toutes ces fonctions coexistent entre-elles et prennent donc plus de place en mémoire vive.

4 Un algorithme récursif à complexité exponentielle

Rappel : La suite de Fibonacci est définie par :

- $U_0=0$ et $U_1=1$
- $U_{n+2}=U_{n+1}+U_n$ pour tout entier $n\geq 2$

A faire vous même 11.

- Reprenez cette fonction récursive en python
- Exécutez cette fonction avec une dizaine de valeurs
- Mesurez les temps d' exécution et dessinez les graphes

- Qu' en concluez-vous ?

.....

.....

Nous verrons plus tard dans l' année comment améliorer ceci grâce à la mémoïsation.

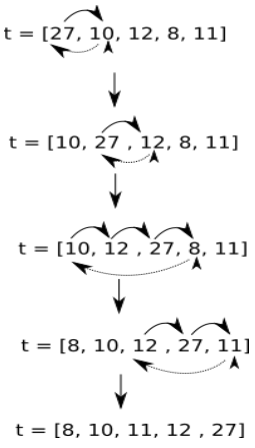
P. 48 ex 6

5 Un algorithme récursif de tri : Le tri fusion

5.1 Rappel sur les différents tris

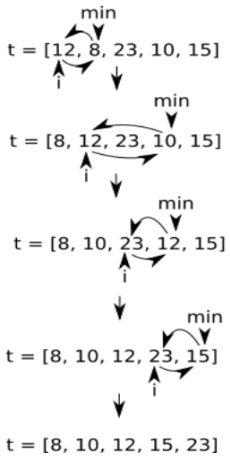
5.1.1 Tri par insertion

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def tri_insertion(liste):
    j = 1
    while j<len(liste):
        i = j-1
        k = liste[j]
        while i>=0 and liste[i]>k:
            liste[i+1] = liste[i]
            i = i-1
        liste[i+1] = k
        j = j+1
    return liste
```



5.1.2 Tri par sélection

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def tri_selection(liste):
    i = 0
    while i<len(liste):
        j = i+1
        min = i
        while j<len(liste):
            if liste[j]<liste[min]:
                min = j
            j = j+1
        if min != i :
            liste[i], liste[min] = liste[min],
liste[i]
        i = i+1
    return liste
```



5.1.3 Tri à bulles

Fonctionne sur le principe suivant :

- On regarde si les 2 premières valeurs sont rangées en ordre croissant. Si ce n' est pas le cas, on les échange
- On regarde si les 2 valeurs suivantes sont rangées en ordre croissant. Si ce n' est pas le cas, on les échange
- …

Une fois la 1ère passe finie, on recommence. Si la liste contient n valeurs, le tableau sera trié au bout de n-1 passes au plus

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

def tri_a_bulle(liste):
    for j in range(len(liste)-1):
        for i in range(len(liste)-1):
            if liste[i] > liste[i+1]:
                liste[i], liste[i+1] = liste[i+1], liste[i]
    return liste
```

5.2 Le tri fusion

5.2.1 Principe : Diviser pour régner

Le diviser pour régner est une méthode algorithmique basée sur le principe suivant :

.....

.....

.....

.....

Le paradigme "diviser pour régner" repose donc sur 3 étapes :

.....

.....

.....

.....

.....

.....

Les algorithmes basés sur le paradigme "diviser pour régner" sont très souvent des algorithmes récursifs.

5.2.2 Algorithme

Comme pour les algorithmes déjà étudiés, cet algorithme de tri fusion prend en entrée un tableau non trié et donne en sortie, le même tableau, mais trié.

A faire vous même 12.

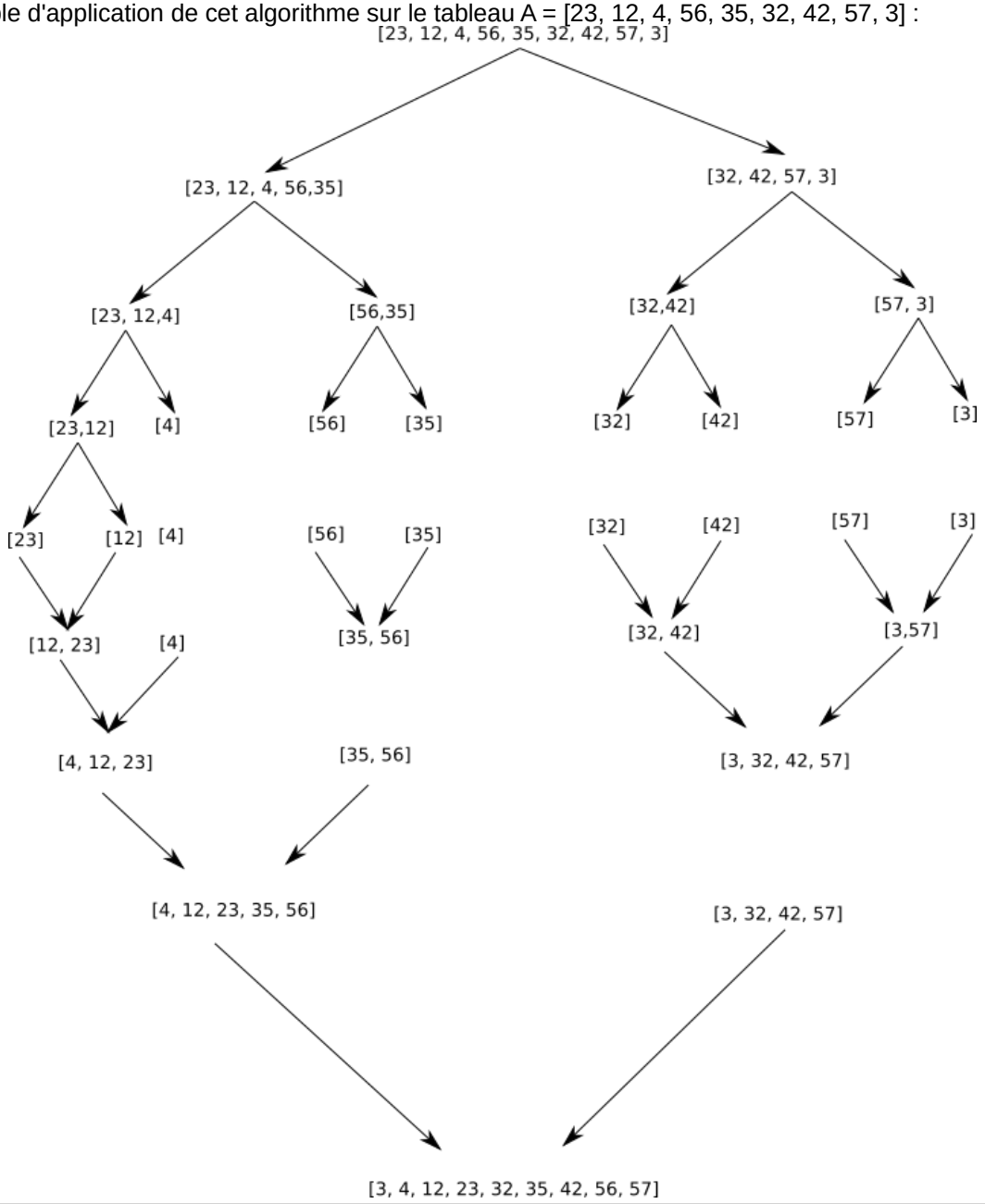
Étudiez cet algorithme :

```
VARIABLE
A : tableau d'entiers
L : tableau d'entiers
R : tableau d'entiers
p : entier
q : entier
r : entier
n1 : entier
n2 : entier

DEBUT

FUSION (A, p, q, r):
  n1 ← q - p + 1
  n2 ← r - q
  créer tableau L[1..n1+1] et R[1..n2+1]
  pour i ← 1 à n1:
    L[i] ← A[p+i-1]
  fin pour
  pour j ← 1 à n2:
    R[j] ← A[q+j]
  fin pour
  L[n1+1] ← ∞
  R[n2+1] ← ∞
  i ← 1
  j ← 1
  pour k ← p à r:
    si L[i] ⩽ R[j]:
      A[k] ← L[i]
      i ← i + 1
    sinon:
      A[k] ← R[j]
      j ← j + 1
    fin si
  fin pour
fin FUSION

TRI-FUSION(A, p, r):
  si p < r:
    q = (p + r) / 2
    TRI-FUSION(A, p, q)
    TRI-FUSION(A, q+1, r)
    FUSION(A, p, q, r)
  fin si
fin TRI-FUSION
FIN
```

Pour trier un tableau A, on fait l'appel initial `TRI-FUSION(A, 1, A.longueur)`
Rappel : Attention, en algorithmique, les indices des tableaux commencent à 1

Cet algorithme est un peu difficile à appréhender, on notera qu'il est composé de deux fonctions `FUSION` et `TRI-FUSION` (fonction récursive). La fonction `TRI-FUSION` assure la phase "DIVISER" et la fonction `FUSION` assure les phases "RÉGNER" et "COMBINER".

Voici un exemple d'application de cet algorithme sur le tableau A = [23, 12, 4, 56, 35, 32, 42, 57, 3] :



A faire vous même 13.

Étudiez attentivement le schéma ci-dessus afin de mieux comprendre le principe du tri-fusion (identifiez bien les phases "DIVISER" et "COMBINER").
On remarque que dans le cas du tri-fusion, la phase "RÉGNER" se réduit à sa plus simple expression, en effet, à la fin de la phase "DIVISER", nous avons à trier des tableaux qui comportent un seul élément, ce qui est évidemment trivial.

La fusion des 2 tableaux déjà triés est simple, prenons comme exemple la dernière fusion entre le tableau [4, 12, 23, 35, 56] et le tableau [3, 32, 42, 57] (le principe est identique pour toutes les fusions) :
Soit T le tableau issu de la fusion du tableau B = [4, 12, 23, 35, 56] et du tableau C = [3, 32, 42, 57] (on donne des noms aux tableaux uniquement pour essayer de rendre l'explication la plus claire possible).

- On considère le premier élément du tableau B (4) et le premier élément du tableau C (3) : 3 est inférieur à 4, on place 3 dans le tableau T et on le supprime du tableau C. Nous avons donc alors
  T = [3], B = [4, 12, 23, 35, 56] et C = [32, 42, 57].
- On recommence ensuite à comparer le premier élément du tableau B (4) et le premier élément du tableau C (32) : 4 est inférieur à 32, on place 4 dans le tableau T et on le supprime du tableau B. Nous avons donc alors
  T = [3, 4], B = [12, 23, 35, 56] et C = [32, 42, 57].
- On compare le premier élément du tableau B (12) et le premier élément du tableau C (32) : 12 est inférieur à 32, on place 12 dans le tableau T et on le supprime du tableau B. Nous avons donc alors
  T = [3, 4, 12], B = [23, 35, 56] et C = [32, 42, 57].
- On compare le premier élément du tableau B (23) et le premier élément du tableau C (32) : 23 est inférieur à 32, on place 23 dans le tableau T et on le supprime du tableau B. Nous avons donc alors
  T = [3, 4, 12, 23], B = [35, 56] et C = [32, 42, 57].
- On compare le premier élément du tableau B (35) et le premier élément du tableau C (32) : 32 est inférieur à 35, on place 32 dans le tableau T et on le supprime du tableau C. Nous avons donc alors
  T = [3, 4, 12, 23, 32], B = [35, 56] et C = [42, 57].

- On compare le premier élément du tableau B (35) et le premier élément du tableau C (42) : 35 est inférieur à 42, on place 35 dans le tableau T et on le supprime du tableau B. Nous avons donc alors
  T = [3, 4, 12, 23, 32, 35], B = [56] et C = [42, 57].
- On compare le premier élément du tableau B (56) et le premier élément du tableau C (42) : 42 est inférieur à 56, on place 42 dans le tableau T et on le supprime du tableau C. Nous avons donc alors
  T = [3, 4, 12, 23, 32, 35, 42], B = [56] et C = [57].
- On compare le premier élément du tableau B (56) et le premier élément du tableau C (57) : 56 est inférieur à 57, on place 56 dans le tableau T et on le supprime du tableau B. Nous avons donc alors
  T = [3, 4, 12, 23, 32, 35, 42, 56], B = [] et C = [57].
- Le tableau B est vide, il nous reste juste à placer le seul élément qui reste dans C (57) dans T :
  T = [3, 4, 12, 23, 32, 35, 42, 56, 57], B = [] et C = [].
  La fusion est terminée.

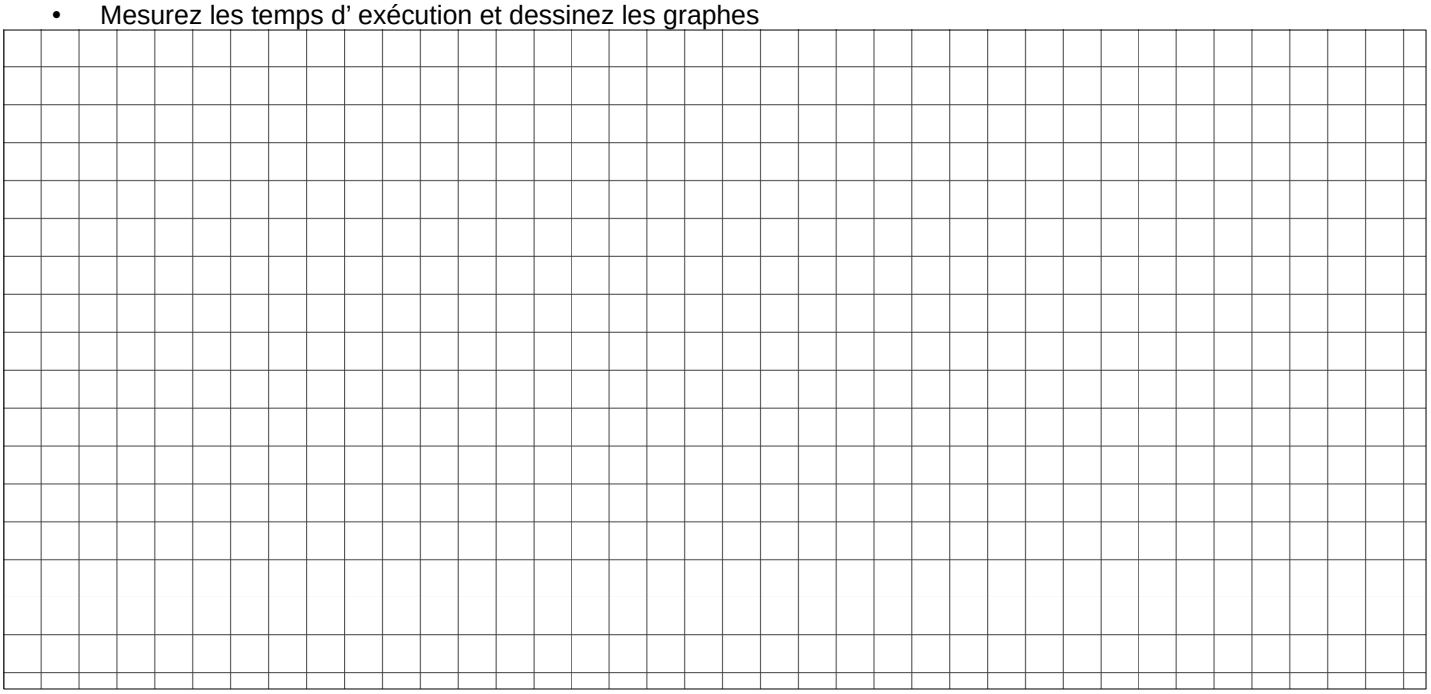
A faire vous même 14.

- Écrivez une fonction python `fusion` qui prend 2 listes ordonnées et qui les fusionne en une liste ordonnée suivant l' algorithme décrit ci-dessus.

- Écrivez une fonction récursive `triFusion` :
  - qui prend une liste non ordonnée,
  - qui divise cette liste en deux,
  - qui fait 2 appels récursifs pour chacune de ces listes
  - et qui fusionne ces listes en faisant appel à la fonction fusion.

A faire vous même 15.

- Téléchargez le fichier suivant : http://ninoo.fr/LC21_22/NSI_Term/seq5_recursivite/listes_a_trier.py
  Vous y trouverez des listes générées aléatoirement de longueurs de 10, 50, 100, 500 et 1000 que vous pourrez importer.
- Reprenez ces fonctions de tri
- Exécutez ces fonctions avec les listes importées ci-dessus
- Mesurez les temps d' exécution et dessinez les graphes



- Qu' en concluez-vous ?

…….

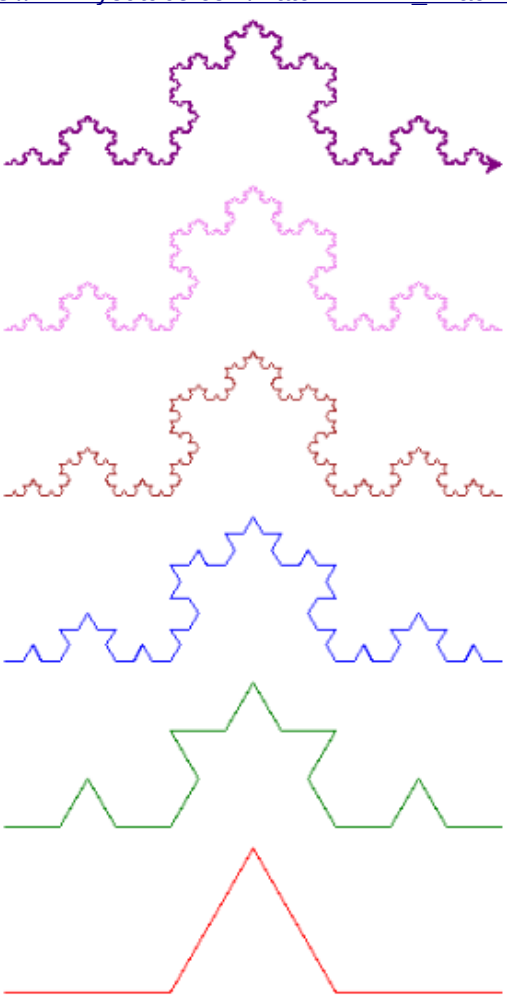
P. 48 ex 7

6 Application : Les fractales

Les fractales est une branche des mathématiques non enseigné en lycée et pourtant elles sont intéressantes à plusieurs points de vue.
Un documentaire-introduction : https://www.youtube.com/watch?v=Tpsu2uz9rCE
Pour ceux qui veulent aller plus loin : https://www.youtube.com/watch?v=iFA3g_4myFw

6.1 Le flocon de Koch

Voir la vidéo :
https://www.youtube.com/watch?v=PW_Pka9iBko



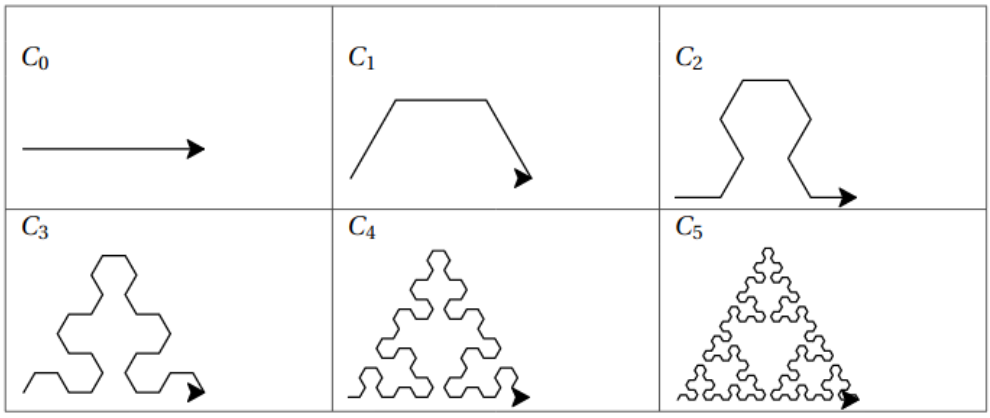
A faire vous même 16.

Complétez la fonction python permettant de dessiner le flocon de Koch par récursivité :

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Les imports
import turtle
from turtle import Screen, Terminator, Turtle

# Liste de couleurs pour que ce soit plus jolie
COLOR = ['red', 'green', 'blue', 'brown',
'violet', 'purple', 'yellow']

# Pour empêcher Turtle de se mettre en erreur
def spyder_bye():
    try:
        Screen().bye()
        turtle.TurtleScreen._RUNNING = True
    except Terminator:
        pass
    turtle.bye = spyder_bye

# La fonction récursive
def draw_branch(n, longueur=30):
    t.down()
    if n <= 1 :
        … # A compléter
    else :
        … # A compléter
    … # A compléter
    t.up()

# Le corps principale
t=Turtle()
t.speed(0)
distance = 100
profondeur = 3
t.up()
t.setpos(-2*distance,0)
for i in range(3):
    t.color(COLOR[i])
    draw_branch(profondeur, distance)
    # Appel à la fonction récursive
    t.right(120)
t.hideturtle()
t.screen.exitonclick() # A noter : Il faut
#cliquer sur la fenêtre graphique pour quitter
turtle.bye()
```

6.2 Courbe de Sierpinski (pour les plus rapides)

La courbe de Sierpinski est une courbe fractale dont l'initiateur est un segment et dont le générateur remplace un segment de longueur c par trois segments de longueurs c/2, le premier fait un angle de×60◦avec le segment remplacé pour le $n^{ième}$ téragone, le second lui est parallèle et le troisième forme un angle de−60◦avec le segment remplacé. A chaque remplacement d'un segment par un générateur, l'orientation de 60◦du générateur alterne selon la position 1, 2 ou 3 du segment remplacé : la rotation s'effectue dans le sens opposé pour les segments 1 et 3 et dans le sens opposé pour le segment 2



A faire vous même 17.

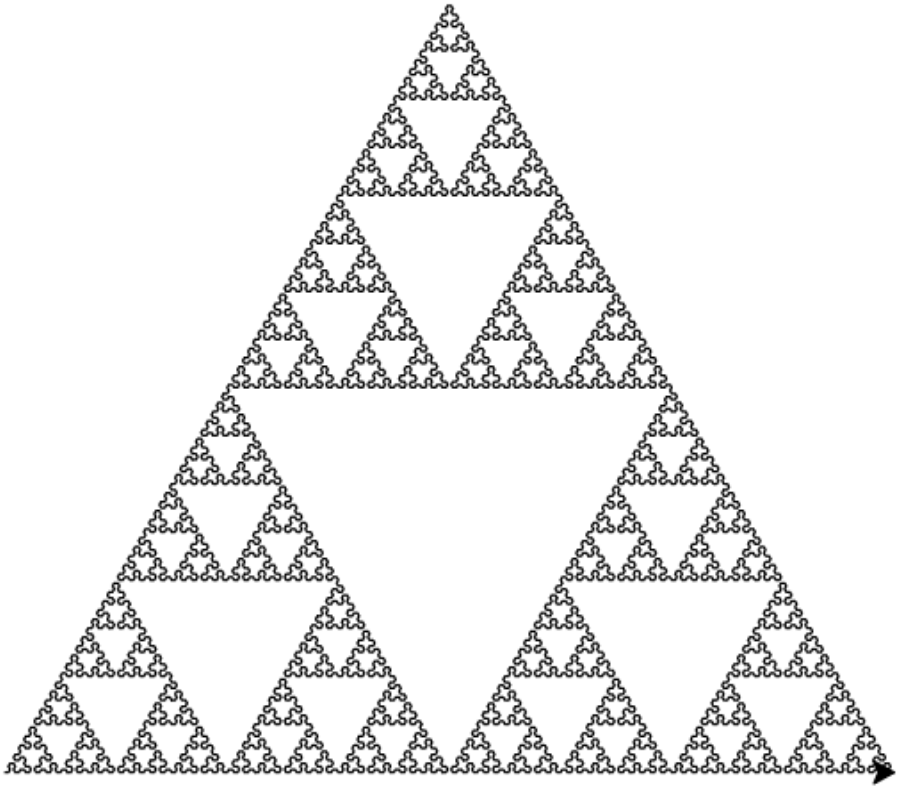
- Compléter la série d'instructions suivantes pour le tracé du téragone C1 de la courbe de Sierpinski si onpart d'un initiateur C0 de longueur 100 pixels.

```
k = 1 #coefficient pour choisir l'orientation du generateur.
C = 100
left(k*60)
forward(c/2)
right(k*60)
#a completer
```

- Compléter la fonction récursive `courbe_sierpinski(n, c, k)` ci-dessous pour qu'elle trace le nième téragone de la courbe de Sierpinski en partant d'un segment initiateur de longueur c pixels. Dans la description du générateur C1, on remplace les instructions `forward(c/2)` de tracé de segment par des appels récursifs `courbe_sierpinski(n - 1,c/2,k)` ou `courbe_sierpinski(n - 1,c/2,-k)` selon l'orientation du générateur.

```
def courbe_sierpinski(n, c, k):
    if n == 0:
        forward(c)
    else:
        left(k*60)
        courbe_sierpinski(n - 1, c/2, -k)
        #a completer
```

- Tracer plusieurs téragones Cn avec n de plus en plus grand, observer la forme obtenue et faire une conjecture sur la courbe de Sierpinski.



6.3 Autre courbe (pour les plus rapides)

A faire vous même 18.

Choisissez une des 3 courbes et programmez la fonction récursive correspondante.

- Courbe de Peano: https://mathcurve.com/fractals/peano/peano.shtml
- Courbe de Koch quadratique : https://mathcurve.com/fractals/kochquadratique/kochquadratique.shtml
- Courbe du dragon : https://mathcurve.com/fractals/dragon/dragon.shtml

6.4 La fractale de Mandelbrot – Codage en Python (pour les plus rapides)

Quel est le point commun entre un chou romanesco, les côtes terrestres, une feuille de fougère ou bien encore un flocon de neige ? Tous ces éléments sont de nature fractale, c'est-à-dire qu'il possède une propriété d'auto-similarité quelle que soit l'échelle à laquelle on les observe. Autrement dit, le tout est semblable à l'une de ses parties. C'est en reprenant les travaux de Gaston Julia et Pierre Fatou que Benoît Mandelbrot a introduit et définit le terme « fractale ». Grâce aux moyens informatiques dont il disposait, il a pu obtenir une représentation « visuelle » de ces objets mathématiques que ces prédécesseurs ne pouvaient seulement qu'imaginer.
Dans cet article, nous allons définir l'ensemble de Mandelbrot, écrire un algorithme permettant de déterminer si un point du plan fait partie ou non de cet ensemble et enfin écrire un programme Python permettant de visualiser la fractale de Mandelbrot.

6.4.1 Définition de la fractale de Mandelbrot

Soit un point donné $C(c_x;c_y)$ du plan muni d'un repère $(O;\vec{u},\vec{v})$. Pour tout entier naturel n , on construit, à partir des coordonnées de ce point C , une suite de points $M_n(x_n;y_n)$ défini par les relations de récurrence suivantes :

$$\begin{pmatrix} x_0=y_0=0 \\ x_{n+1}=x_n^2-y_n^2+c_x \\ y_{n+1}=2x_ny_n+c_y \end{pmatrix}$$

Pour déterminer si le point C appartient ou non à l'ensemble de Mandelbrot, on commence par calculer quelques termes des suites (x_n) et (y_n) . Prenons deux exemples concrets (en arrondissant les résultats) :

- Soit $C(1;1)$, alors la suite des points (M_n) construite à partir de ce point C est $M_0(0;0)$; $M_1(1;1)$; $M_2(1;3)$; $M_3(-7;7)$; $M_4(1;-97)$; $M_5(-9407;-193)$
- Soit $C(0{,}1;0{,}2)$, alors la suite des points (M_n) construite à partir de ce point C est $M_0(0;0)$; $M_1(0{,}1;0{,}2)$; $M_2(0{,}07;0{,}24)$; $M_3(0{,}0473;0{,}234)$; $M_4(0{,}0477;0{,}222)$; $M_5(0{,}0529;0{,}221)$

Nous allons nous intéresser à la distance $OM_n=\sqrt{x_n^2+y_n^2}$ c'est-à-dire à la distance qui sépare le point M_n de l'origine du repère et nous pouvons constater que deux cas de figures peuvent se présenter :

- Soit la distance OM_n augmente infiniment, autrement dit les suites (x_n) et (y_n) divergent vers l'infini.
- Soit la distance OM_n est bornée, autrement dit les suites (x_n) et (y_n) sont bornées.

Évidemment, tout ceci n'est que conjecture puisque nous n'avons calculer que les 6 premiers termes de chaque suite. Toutefois, un résultat que l'on admettra permet d'affirmer que si la distance OM_n devient supérieure à 2 à partir d'un certain rang, alors les suites divergent et la distance OM_n tend vers l'infini. En revanche, si pour une certaine valeur de n suffisamment grande, la distance OM_n reste inférieure à 2, alors on pourra considérer que les deux suites sont bornées et que cette distance OM_n est bornée (minorée par 0 et majorée par 2).

La règle de prise de décision quant à l'appartenance du point $C(c_x;c_y)$ à l'ensemble de Mandelbrot est alors la suivante :

- Soit le point M_n « s'éloigne » infiniment de l'origine auquel cas le point C n'appartient pas à l'ensemble de Mandelbrot, ce qui est le cas du point C de coordonnées (1;1)
- Soit le point M_n « reste » au voisinage de l'origine c'est-à-dire dans un cercle de centre O et de rayon 2, auquel cas le point C appartient à l'ensemble de Mandelbrot, c'est le cas du point C de coordonnées (0,1;0,2)

6.4.2 Algorithme associé

Maintenant que nous pouvons dire si un point du plan fait partie ou non de l'ensemble de Mandelbrot, nous allons pouvoir visualiser cet ensemble en utilisant l'outil informatique. L'algorithme est le suivant :

```
Affecter à MAX_ITERATION une valeur seuil
Pour chaque pixel C de coordonnées (x;y) de l'écran
    Convertir (x;y) dans le système de coordonnées du repère
    Tant Que la distance OMn < 2 et que n < MAX_ITERATION
          Calculer les coordonnées de Mn
          Affecter à n la valeur n+1
    Fin Tant Que
    Si n = MAX_ITERATION Alors
          Colorier le pixel en noir
    Sinon
          Colorier le pixel en blanc
    Fin Si
Fin Pour
```

L'algorithme est relativement simple. Il balaye l'écran pixel par pixel en convertissant ses coordonnées dans le système de coordonnées de notre repère pour savoir si celui-ci fait partie ou non de l'ensemble de Mandelbrot. A la sortie de la boucle « Tant Que », deux cas de figures se présentent :

- Soit on est sorti de la boucle parce que la distance OM_n est devenue plus grande que 2 auquel cas le pixel (et donc le point C) ne fait pas partie de l'ensemble de Mandelbrot, on lui attribue alors la couleur blanche.
- Soit on est sorti de la boucle parce que le seuil du nombre d'itérations maximales est atteint, c'est-à-dire qu'on a calculé suffisamment de termes de la suite pour considérer que la distance OMn restera toujours plus petite que 2 auquel cas le pixel (et donc le point C) fait partie de l'ensemble de Mandelbrot et on lui attribue la couleur noire.

6.4.3 Programme python associé

A faire vous même 19.

Voici un début de programme python à compléter :

```
from math import sqrt
from PIL import Image

MAX_ITER = 50

def reccurenceMandelbrot(m, c):
    # A COMPLETER
    return (x, y)

def appartientAEnsembleMandelbrot(c) :
    # A COMPLETER
    return True


#CORPS DE PROGRAMME
LARG, HAUT = 600, 600
MIN_X, MAX_X = -2, 2
MIN_Y, MAX_Y = -2, 2
coef_x = (MAX_X - MIN_X)/LARG
coef_y = (MAX_Y - MIN_Y)/HAUT
img = Image.new('RGB', [LARG,HAUT], (0,0,0))
data = img.load()
for i in range(img.size[0]):
    x = i*coef_x + MIN_X
    for j in range(img.size[1]):
        y = j*coef_y + MIN_Y
        c = (x,y)
        conclusion = appartientAEnsembleMandelbrot(c)
        if conclusion :
            data[i,j] = (255,255,255)
        else :
            data[i,j] = (0,0,0)
img.save('image.png')
img.show()
```

P. 49 ex 8

P. 49 ex 9