MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE LA JEUNESSE
ET DES SPORTS

Liberté
Égalité
Fraternité

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et sciences informatiques

ENSEIGNEMENT SPÉCIALITÉ

RÉCURSIVITÉ

Cette ressource a pour objectif d'accompagner le professeur dans sa préparation de cours. Elle est conçue pour être utilisable en l'état. Toutefois les exemples proposés sont exhaustifs (et donc parfois redondants). Il appartient au professeur de sélectionner ceux qu'il juge les plus pertinents selon le profil de sa classe pour en faire une utilisation en classe entière ou pour une utilisation en séances d'exercices.

SOMMAIRE

1. Le principe 3

1. Définition 3
2. Premier exemple 3
3. Deuxième exemple 4
   - Version itérative 4
   - Version récursive 5
4. Autre exemple 5
   - Version itérative 5
   - Version récursive 6

2. Les exemples classiques 6

1. La suite de Fibonacci 6
2. La somme des n premiers entiers 7
3. L'opération de puissance n-ième d'un nombre x 8

Retrouvez éduscol sur

4. La fonction factorielle d'un entier naturel n 10
5. Le maximum dans une liste de nombres 11
6. Quelques exemples de programmes récursifs sur les chaînes de caractères 11
a) Les palindromes 11
b) Les anagrammes 12
Proposition de correction 13
7. Figures récursives 13
a) Bubble 13
b) Le Flocon de Von Koch 14
c) Le triangle de Sierpinski 15
8. Un exemple de projet sur la récursivité réinvestissant les aspects graphiques 17
3. Les limites de la récursivité 19
4. Bilan 21
Prolongements 22
Sources 22
Des exercices d'entraînement 22

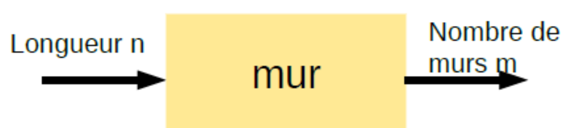
1. Le principe

1. Définition

Une **fonction récursive** est une fonction qui s'appelle elle-même.

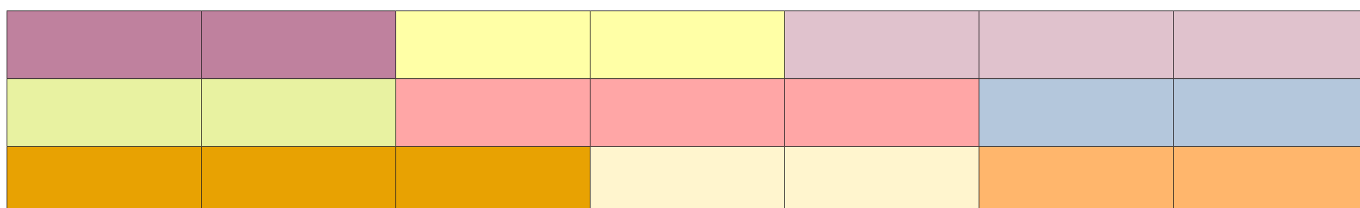
2. Premier exemple

Ce premier exemple peut être traité en algorithmique débranchée à l'aide de briques de Lego.



On dispose de briques de longueur 2 ou 3. Combien de murs de longueur n distincts peut-on construire avec ces briques ? (Extrait du projet Euler)

Voici par exemple les trois solutions possibles pour des rangées de longueur 7 : (chaque brique est symbolisée par une couleur différente)



Ainsi il y a 3 façons de construire un mur de longueur 7 : 223 232 322

Méthodologie :

Plusieurs approches sont possibles. Voici une approche dite récursive (mais on sait d'après une propriété démontrée que pour toute approche récursive, il existe aussi une approche itérative).

Technique pour construire un mur de longueur n :

- on met une brique de 2 et on construit un mur de longueur n-2 ;
- on met une brique de 3 et on construit un mur de longueur n-3 ;
- cas limites :
  si on a 0 place, le mur est fini, on n'en rajoutera pas on renvoi 0 ;
  si on a 1 place, le mur n'est pas constructible on renvoi 0 ;
  si on a 2 ou 3 places, on met une brique et c'est fini : on renvoie.

On obtient ainsi une relation récursive (en mathématique, on appelle cela une relation de récurrence) :

si n<=1 : mur(n)=0

sinon : mur(n)=mur(n-2)+mur(n-3), c'est ici que l'on voit que cette fonction est récursive.

Si on cherche une solution itérative à ce problème, elle n'est pas du tout évidente. Dans la partie *IV Bilan*, nous présentons des critères sur le choix de la méthode : itérative ou récursive.

Les résultats attendus de notre fonction mur : (la fonction écrite renvoi en plus la liste des murs possibles)

```
il y a  3  façons de construire un mur de longueur  7
223 232 322
il y a  4  façons de construire un mur de longueur  8
2222 233 323 332
il y a  5  façons de construire un mur de longueur  9
2223 2232 2322 3222 333
```

3. Deuxième exemple

Vous avez probablement vu en mathématiques la récursivité lorsque vous avez étudié les **suites définies par récurrence**. On la retrouve aussi comme un puissant moyen de démonstration avec la **démonstration par récurrence** (programme de terminale de la spécialité mathématiques).

Dans cet exemple, on s'intéresse à la suite (u_n) définie par son premier terme $u_0=2$ et la formule de passage d'un terme au suivant : $u_n=3\ u_{n-1}$.

On souhaite calculer le terme d'indice n de cette suite.

Note : pour les élèves ayant suivi l'enseignement de spécialité mathématiques en première, on occulte volontairement que cette suite est géométrique et qu'on peut par propriété exprimer directement la valeur de u_n en fonction de n.

Version itérative vs version récursive

Version itérative

Il s'agit de la méthode que vous avez l'habitude d'utiliser pour répondre à un problème de programmation : pour calculer u_3 par exemple, on calcule :

- $u_1=3\times u_0=3\times 2=6$
- $u_2=3\times u_1=3\times 6=18$
- $u_3=3\times u_2=3\times 18=54$

À faire par l'élève : écrire une fonction qui calcule le terme d'indice n de la suite (un) de manière itérative.

```
def u(n):
    #A compléter
    return #A compléter

assert u(3) == 54
assert u(10) == 118098
```

Retrouvez éduscol sur

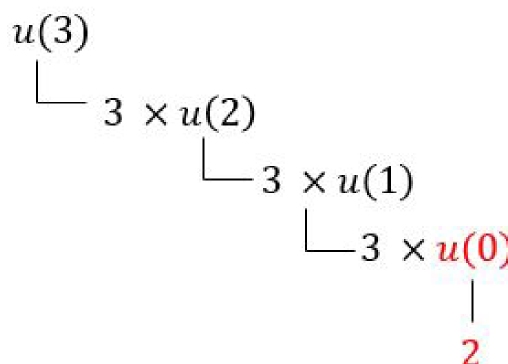
Version récursive

On peut également programmer le calcul des termes de cette suite en calquant sa définition par récurrence :

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

assert u(3) == 54
assert u(10) == 118098
```

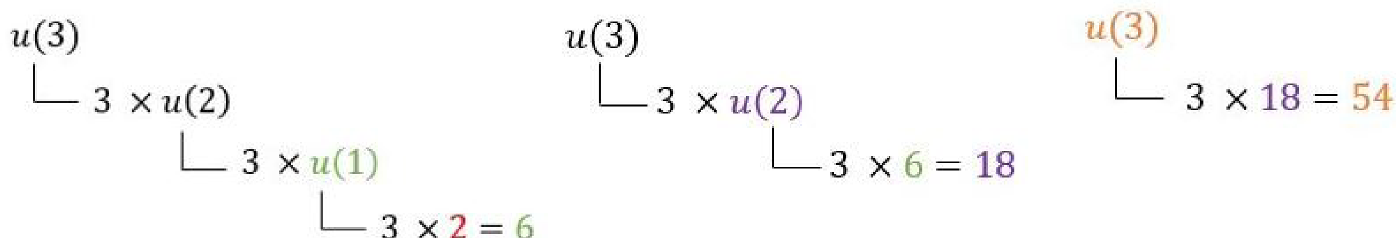
L'évaluation de l'appel à u(3) peut se représenter de la manière suivante.



Cette manière de représenter l'exécution d'un programme en indiquant les différents appels effectués est appelée **un arbre d'appels**.

Ainsi, pour calculer la valeur renvoyée par u(3), il faut tout d'abord appeler u(2). Cet appel va lui-même déclencher un appel à u(1), qui à son tour nécessite un appel à u(0). Ce dernier appel se termine directement en renvoyant la valeur 2.

Le calcul de u(3) se fait ensuite « à rebours » :



4. Autre exemple

À faire par les élèves : programmer des deux façons précédentes (version itérative et version récursive) le calcul du terme u_n d'une suite définie par ses deux premiers termes $u_0=5$ et $u_1=7$ et la formule de récurrence suivante : $u_n=2\times u_{n-2}+3\times u_{n-1}$.

Version itérative

```
def u(n):
#A compléter
    return #A compléter
```

Version récursive

On choisit d'appeler la suite v afin de comparer avec les résultats du programme écrit en version itérative.

```
def v(n):
    #A compléter
    return #A compléter

assert u(5) == v(5)
assert u(10) == v(10)
```

2. Les exemples classiques

1. La suite de Fibonacci

La suite de Fibonacci est la suite (u_n) définie par :

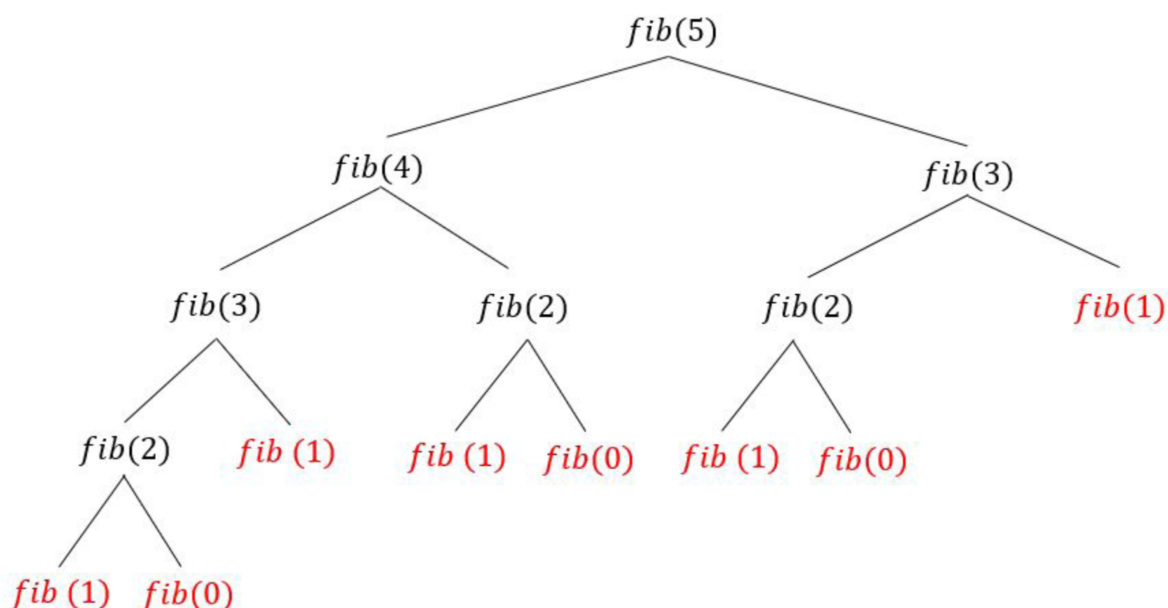
$u0=1$

$u1=1$

$u_n=u_{n-1}+u_{n-2}, \forall n \geqslant 2$

On souhaite écrire une fonction fib(n), définie de façon récursive, qui retourne le terme de rang n de la suite (u_n).

À présent, dessinons l'**arbre d'appels** de cette fonction :



Retrouvez éduscol sur

Les instructions fib(1) et fib(0) sont appelées les **cas de base** de ce programme récursif : il s'agit des cas pour lesquels on peut obtenir le résultat sans avoir recours à l'appel de la fonction.

Les cas de base sont indispensables à toute fonction récursive : ils permettent au programme de se terminer. En effet, comme une fonction récursive s'appelle elle-même, alors, sans ces cas de base, la fonction s'appellerait à l'infini. C'est pour cette raison que les cas de base sont également appelés **conditions d'arrêt**.

Pour concevoir un programme récursif, il faut toujours se poser la question : **« quel(s) est(sont) le(s) cas de base ? »**

À faire par les élèves : programmer la fonction **fib** de façon récursive.

```
def fib(n):
    #A compléter
    return  #A compléter

assert fib(3) == 3
assert fib(7) == 21
```

2. La somme des n premiers entiers

Pour définir la somme des n premiers entiers, on a l'habitude d'écrire la formule suivante :

$$0+1+2+3+4+...+n$$

Une solution pour calculer cette somme consiste à utiliser une boucle **for** pour parcourir tous les entiers *i* entre *0* et *n*, en s'aidant d'une variable intermédiaire *S* pour accumuler la somme des entiers de *0* à *i*. On obtient par exemple le programme itératif suivant écrit en Python :

```
def somme(n):
    S = 0
    for i in range(n+1):
        S = S + i
    return S

somme(100)
```

S'il n'est pas difficile de se convaincre que la fonction somme(n) ci-dessus calcule bien la somme des n premiers entiers, on peut néanmoins remarquer que ce code Python n'est pas directement lié à la formule de départ.

En effet, il n'y a rien dans cette formule qui puisse laisser deviner qu'une variable intermédiaire S est nécessaire pour calculer cette somme…

Il existe une autre manière d'aborder ce problème : il s'agit de définir la **fonction récursive** somme(n) qui, pour tout entier naturel n, donne la somme des n premiers entiers de la manière suivante :

si n=0 : somme (n) = 0

si n>0 : somme (n) = n+somme(n-1)

À faire par les élèves :

1. Écrire la fonction récursive somme(n) qui calcule la somme des n premiers entiers.
2. Dessiner l'arbre d'appels de cette fonction pour l'appel somme(5).
3. Parcourir cet arbre d'appels «à rebours» pour illustrer le fonctionnement de la fonction somme comme cela a été fait dans la version récursive du premier exemple du paragraphe I.

```
def somme(n):
    #A compléter
    return #A compléter

assert somme(4) == 10
assert somme(100) == 5050
```

3. L'opération de puissance n-ième d'un nombre x

Il s'agit de la multiplication répétée n fois de x avec lui-même, que l'on écrit habituellement ainsi :

$x^n = x. x. x. x\dots x. x$ Le facteur x est contenu n fois

avec, par convention, $x^0=1$.

Pour écrire une version récursive du calcul de x^n, on va définir une fonction `puissance(x,n)` ainsi :

- on commence par déterminer le cas de base à cette opération : il s'agit du cas pour n=0 ;
- pour définir ensuite la valeur de puissance(x,n) pour un entier n strictement positif, on utilise le résultat suivant :

$$x^n=x\times x^{n-1}$$

ainsi, x^n s'obtient en multipliant x par x^{n-1}. Une définition récursive de l'opération de puissance n-ième d'un nombre x est alors la suivante :

si n=0 $x^n = 1$

si n>0 $x^n = x.x^{n-1}$

À faire par les élèves :

1. Écrire la fonction récursive puissance(x,n) qui calcule le nombre x^n pour tout entier naturel n.
2. Dessiner l'arbre d'appels de cette fonction lorsque x = 3 et n = 5.

3. Parcourir cet arbre d'appels « à rebours » pour illustrer le fonctionnement de la fonction puissance comme cela a été fait dans la version récursive du premier exemple du paragraphe I.

```
def puissance(x,n):
    #A compléter
    return #A compléter

assert puissance(2,7) == 128
assert puissance(9,5) == 59049
```

À noter : la définition d'une fonction récursive n'est pas toujours unique.

Par exemple, la définition récursive de l'opération de puissance n-ième d'un nombre peut également être :

si n=0	$x^n = 1$
si n=1	$x^n = x$
si n>1	$x^n = x.x^{n-1}$

L'introduction d'un deuxième cas de base dans cette nouvelle définition a l'avantage d'éviter de faire la multiplication inutile de x par 1 qui, dans la première définition est faite à chaque appel de la fonction puissance(x,n) à partir du moment où n   0.

On pourrait continuer à ajouter des cas de base pour n=2, n=3, etc., mais cela n'apporterait rien à la définition. En particulier, cela ne réduirait pas le nombre de multiplication.

Voici une troisième définition récursive de cette même fonction qui distingue le calcul de x^n selon la parité de n :

si n=0	$x^n =1$
si n est pair	$x^n = (x^{n/2})^2$
si n est impair	$x^n = x.(x^{(n-1)/2})^2$

L'algorithme qui découle de cette définition porte également le nom d'**exponentiation rapide**. Comme son nom l'indique, il s'agit d'un algorithme particulièrement efficace pour calculer rapidement de grandes puissances entières.

À faire par les élèves :

1. Écrire la fonction récursive puissancev2(x,n) qui calcule le nombre x^n pour tout entier naturel n selon la deuxième méthode.
   On propose ci-dessous la fonction récursive puissancev3(x,n) qui calcule le nombre x^n pour tout entier naturel n selon la troisième méthode.
2. Dessiner l'arbre d'appels de cette fonction pour l'appel puissancev3(3,5).

3. Parcourir l'arbre d'appels « à rebours » pour illustrer le fonctionnement de la fonction puissancev3(x,n) lorsque n = 5 et x = 3.

```
def puissancev2(x,n):
    #A compléter
    return #A compléter

assert puissancev2(3,5) == 243

def carre(x):
    return x * x


def puissancev3(x,n):
    if n == 0:
        return 1
    else:
        if n % 2 == 0:
            return carre(puissancev3(x,n // 2))
        else:
            return x * carre(puissancev3(x,(n-1) // 2))

assert puissancev3(3,5) == 243
assert puissancev3(7,4) == 2401
```

4. La fonction factorielle d'un entier naturel n

La factorielle d'un entier naturel n, noté n! (se lit **factorielle** n), est le produit des nombres entiers strictement positifs inférieurs ou égaux à n :

$$n! = n \times (n-1) \times \ldots \times 1$$

avec, par convention, $0! = 1$.

On a par exemple :

- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

La fonction factorielle est définie sur l'ensemble des entiers naturels et admet une définition récursive :

si $n = 0$ $\quad n! = 1$

si $n \geqslant 1$ $\quad n! = n \times (n-1)!$

À faire par les élèves :

1. Écrire la fonction récursive factorielle(n) qui calcule le nombre n! pour tout entier naturel n.
2. Dessiner l'arbre d'appels de cette fonction pour l'appel factorielle(7).
3. Parcourir cet arbre d'appels « à rebours » pour illustrer le fonctionnement de la fonction factorielle lorsque n = 4.
4. Vérifier les résultats avec 5!=120 et 7!=5040.

5. Le maximum dans une liste de nombres

On donne le programme suivant :

```
def maximum(a,b):
    if a > b:
        return a
    else:
        return b

def maximum_liste(L):
if len(L) == 1:
    return L[0]
else:
    return maximum(L[0],maximum_liste(L[1:]))

import random as r
L = []
for i in range(20):
    L.append(r.randint(-100,100))
print(L)
print(maximum_liste(L))
```

À faire par les élèves :

1. Décrire, en langage usuel, le principe de fonctionnement de la fonction maximum_liste.
2. Dessiner l'arbre d'appels de cette fonction pour l'appel maximum_lis te([-4,55,-1,-35,-52,31]).
3. Parcourir cet arbre d'appels « à rebours » pour illustrer le fonctionnement de la fonction maximum_liste lorsque L = [-4, 55, -1, -35, -52, 31].

6. Quelques exemples de programmes récursifs sur les chaînes de caractères

a) Les palindromes

On appelle palindrome un mot qui se lit dans les deux sens comme « été » ou « radar ».

Écrire une fonction récursive palindrome qui teste si un mot est un palindrome.

- Entrée : Un mot (type str).
- Sortie : Un booléen égal à True si le mot est un palindrome, False sinon.

Principe

Cas de base :

- si le mot est la chaîne vide, c'est un palindrome ;
- si le mot ne contient qu'une seule lettre, c'est un palindrome.

Dans les autres cas :

- le mot est un palindrome si et seulement si la première et la dernière lettre sont égales et le mot tronqué de ses première et dernière lettres est un palindrome.

```
def palindrome(mot):
#A compléter
return #A compléter
# assert palindrome('antilope') == False
# assert palindrome('radar') == True
```

b) Les anagrammes

L'anagramme d'un mot est un mot s'écrivant avec les mêmes lettres que le mot initial.

Par exemple :

- ironique et onirique ;
- baignade et badinage ;
- estival et vitales.

Ici, nous ne demandons pas que les mots envisagés soient des mots du dictionnaire. Ainsi 'abc' et 'bca' sont deux anagrammes.

Écrire une fonction python récursive anagramme qui renvoie la liste des anagrammes d'un mot.

- Entrée : Un mot (type str).
- Sortie : La liste des anagrammes du mot.

Principe

Cas de base :

- si le mot est une chaîne vide, la liste des anagrammes est alors constituée de l'unique chaîne vide ;
- si le mot n'a qu'une seule lettre, la liste de ses anagrammes ne contient qu'un seul élément : le mot lui-même.

Dans les autres cas :

- on peut définir la liste des anagrammes d'un mot à partir de la liste des anagrammes du mot obtenu en enlevant la première lettre, en plaçant cette première lettre successivement dans toutes les positions dans chaque élément de la liste.

Un exemple : prenons le cas d'un mot de trois lettres : 'abc'.

- La première lettre est 'a'.
- Le mot obtenu en enlevant la première lettre est 'bc' et la liste des anagrammes de 'bc' est ['bc','cb'].
- Pour constituer la liste des anagrammes de 'abc', il suffit de placer la lettre 'a' dans toutes les positions possibles des anagrammes de 'bc'. On obtient la liste ['abc', 'bac', 'bca', 'acb', 'cab', 'cba'].
- De la même façon, on obtiendra la liste de toutes les anagrammes du mot 'dabc' en glissant la lettre 'd' dans toutes les positions possibles de toutes les anagrammes de 'abc'.

Proposition de correction

```
def listeAnagramme(mot) :
    if mot == '' :
        return ['']
    elif len(mot) == 1 :
        return [mot]
    else:
        liste = []
        for anagr in listeAnagramme(mot[1:]): # pour chaque anagramme du mot privé de sa première lettre,
            for k in range(len(mot)) : # on obtient un nouvel anagramme du mot de départ en insérant sa première lettre à toutes les positions possibles.
                liste.append(anagr[:k] + mot[0] + anagr[k:])
        return liste

print(listeAnagramme('abc'))
print(listeAnagramme('chat'))
```

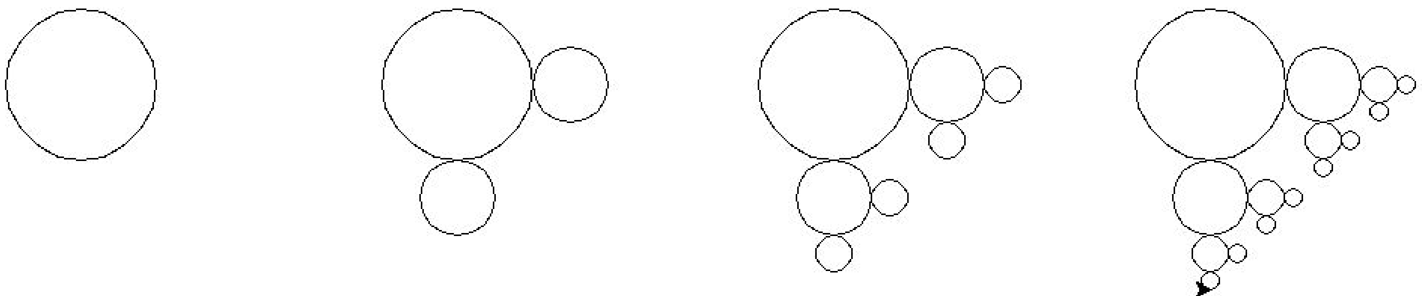
7. Figures récursives

Ces activités fonctionnent très bien avec les élèves par leur aspect ludique.

a) Bubble

Un « bubble » est une figure 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 telles que les lignes des centres soient parallèles aux axes du repère. Ces deux petits cercles deviennent à leur tour « cercle initial » pour poursuivre le dessin.

La figure ci-dessous illustre de gauche à droite des « bubbles » de profondeur 1 à 4.



Un programme permettant d'obtenir la figure à l'étape n où n est un entier naturel strictement positif est donné ci-dessous. La fonction récursive bubble(n,x,y,r) est celle où la tortue dessine un « bubble » de profondeur n à partir d'un cercle de centre (x,y) et de rayon r. À chaque étape, le rayon est divisé par 2.

```
from turtle import *

def dessine_cercle(x,y,r):
    up()
    goto(x,y-r)
    down()
    circle(r)
    return None

def bubble(n,x,y,r):
    if n == 0:
        return None
    else:
        dessine_cercle(x,y,r)
        bubble(n-1,x+3/2*r,y,r/2)
        bubble(n-1,x,y-3/2*r,r/2)
    return None

bubble(3,-100,100,50)
#exitonclick() #en cliquant dans la fenêtre graphique, on provoque sa fermeture, selon IDE utilisé.
```

Avant d'exécuter un nouveau programme produisant une sortie graphique, refermez la fenêtre Turtle.

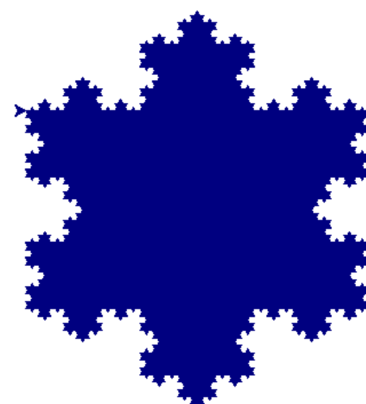
À faire par les élèves :

1. Recopier et exécuter le code.
2. Qu'est-ce qui garantit que cette fonction ne s'appellera qu'un nombre fini de fois ?
3. Dans l'appel initial, si l'on change bubble(3,-100,-100,50) par bubble(5,-200,100,100), qu'obtient-on ?

b) Le Flocon de Von Koch

Une image qui a une apparence similaire quelle que soit l'échelle à laquelle on l'observe est appelée une fractale (il y a d'autres types de fractales).

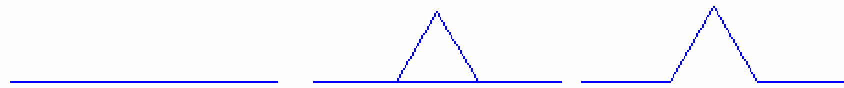
Un exemple simple de fractale est le flocon de Von Koch, dont voici une représentation (pour un degré 4).



On peut la créer à partir d'un segment de droite, en modifiant récursivement chaque segment de droite de la façon suivante :

- on divise le segment de droite en trois segments de longueurs égales ;
- on construit un triangle équilatéral ayant pour base le segment médian de la première étape ;
- on supprime le segment de droite qui était la base du triangle de la deuxième étape.

Voici le résultat obtenu à chacune des étapes.



Pour continuer, il suffit de considérer chaque segment de cette dernière figure comme segment de départ.

```
from turtle import *

def Koch(n,l):
    if n==0:
        forward(l)
    else:
        Koch(n-1,l/3)
        left(60)
        Koch(n-1,l/3)
        right(120)
        Koch(n-1,l/3)
        left(60)
        Koch(n-1,l/3)
    return None

def flocon(n,l):
    for k in range(3):
        Koch(n,l)
        right(120)
    return None

flocon(4,300)
#exitonclick() #en cliquant dans la fenêtre graphique, on provoque sa fermeture, selon IDE utilisé.
```

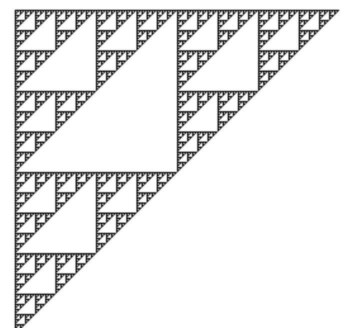
À faire par les élèves :

1. Identifier le cas de base de la fonction récursive Koch(n,l). Que fait-il ?
2. Modifier les paramètres n et l lors de l'appel à la fonction flocon et observer l'impact de ces modifications sur le dessin.
3. Expliquer le principe de fonctionnement de ce programme.
4. Selon le niveau de la classe, faire programmer sans donner le code source ci-dessus.

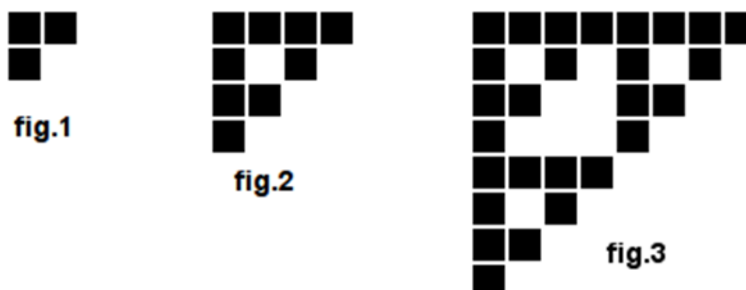
c) Le triangle de Sierpinski

Voici un dernier exemple de figure récursive : le triangle de Sierpsinki. On peut exposer le problème aux élèves et demander une résolution en projet.

On commence par former un triangle de trois pixels (fig.1). On en fait ensuite trois copies que l'on utilise pour former un plus grand triangle (fig2), et on recommence l'opération avec ce nouveau triangle (fig.3), et ainsi de suite, jusqu'à obtenir une figure de la taille souhaitée.



Voici une vue zoomée des trois premières étapes de l'opération :



Les dimensions de la figure étant multipliées par 2 à chaque nouvelle étape, le côté de la figure finale est une puissance de 2.

Tout d'abord une version graphique.

```
import turtle as t
from math import sqrt

t.up()
t.goto(-100,-100)
t.down()

def motif(a):
    #dessine un motif élémentaire sous forme de triangle
    t.down()
    t.fillcolor("black")
    t.begin_fill()
    t.forward(a)
    t.left(90)
    t.forward(a)
    t.left(135)
    t.forward(a*sqrt(2))
    t.left(135)
    t.end_fill()
    t.up()

def Sierpinski(n,a):
    if n == 0:
        motif(a)
    else:
        Sierpinski(n-1,a/2)
        t.forward(a/2)
        Sierpinski(n-1,a/2)
        t.left(90)
        t.forward(a/2)
        t.right(90)
        Sierpinski(n-1,a/2)
        t.right(135)
        t.forward(a*sqrt(2)/2)
        t.left(135)

Sierpinski(3,200)
```

Puis une version en console sans utiliser de tableau, pas si simple qu'il y paraît.

```
# -*- coding: utf-8 -*-
#on n'utilise pas de tableau, et donc il faut aussi remplir
#les "carrés blancs" par des espaces,ce qui complique la chose
from math import *
n=int(input("combien de lignes ?"))
k=int(log(n)/log(2))
if 2**k!=n :
    print("le nombre doit être une puissance de 2. Je retiens n=",2**k)
n=2**k
les_li=['']*int(n) #fabrication d'une liste de n lignes vides
carre=chr(9608)


    def carre_blanc(k,l):
        '''dessine un carré blanc de 2^k caractères à la ligne i'''
        for i in range (2**k):
            les_li[l+i]+=" "*2**k

    def motif(k,l):
        '''fonction récursive faisant le motif'''
        if k<=1 :
            les_li[l]+=carre+carre
            les_li[l+1]+=carre+' '
        else :
            motif(k-1,l)
            motif(k-1,l)
            motif(k-1,l+2**(k-1))
            carre_blanc(k-1,l+2**(k-1))

    motif(k,0) #cet appel rempli les lignes
    for i in range (n): #pour afficher les lignes
        print(les_li[i])
```

L'utilité de la **récursivité** est bien visible dans le cas des figures récursives : on obtient des figures assez complexes en seulement quelques lignes de code. Imaginez le nombre de lignes de codes qu'il serait nécessaire d'écrire pour obtenir ces figures en utilisant une programmation itérative…

Cependant, la récursivité a ses limites, comme nous allons le voir dans la partie suivante (après le projet).

8. Un exemple de projet sur la récursivité réinvestissant les aspects graphiques

Une légende qui remonte à la nuit des temps

« Dans le fameux temple de Bénarès, sous le dôme marquant le centre du Monde, trône un socle de bronze sur lequel sont fixées trois aiguilles de diamant, chacune d'elles haute d'une coudée et fine comme la taille d'une-guêpe. Sur une de ces aiguilles, à la Création, Dieu empila soixante-quatre disques d'or pur, du plus grand au plus petit, le plus large reposant sur le socle de bronze.

Il s'agit de la tour de BRAHMA. Jour et nuit, inlassablement, les moines déplacent les

disques d'une aiguille vers l'autre tout en respectant les immuables lois de BRAHMA qui obligent les moines à ne déplacer qu'un disque à la fois et à ne jamais le déposer sur un disque plus petit. Quand les soixante-quatre disques auront été déplacés de l'aiguille sur laquelle Dieu les déposa à la Création vers une des autres aiguilles, la tour, le temple et les brahmanes seront réduits en poussière, et le Monde disparaîtra dans un grondement de tonnerre. »

Chercher une solution récursive au problème, et faire une interface **en console** qui montre les différentes étapes auxquelles procéder.

Pour réussir, réfléchissez au caractère récursif du problème (quand résoudre le problème pour n revient à résoudre le problème pour n-1 ?), à la structure de données que vous allez utiliser (comment sont modélisées les 3 aiguilles ?), à la structure du programme (quelles sont les différentes fonctions à écrire pour alléger le code ?).

Si vous avez fini avant les autres, vous pouvez réaliser une représentation graphique de la solution avec le module « turtle » ou bien le module « tkinter ».

Une solution récursive du problème en vidéo est présentée par un professeur : « Les tours de Hanoi - Automaths #06 ».

```
debut=[]
transit=[]
fin=[]
def affichage():
    #affichage sommaire
    print(debut,'\t',transit,'\t',fin)
def hanoi(n,debut,fin,transit):
    """déplace n palets de debut à fin en passant par transit"""
    if n>0 :
        hanoi(n-1,debut,transit,fin)
        fin.append(debut.pop())
        affichage()
        #sleep(0.1)
        hanoi(n-1,transit,fin,debut)
N=int(input("entrer le nombre de palets : "))
for i in range (N,0,-1):#remplissage de la pile de début
    debut.append(i)
hanoi(N,debut,fin,transit)
```

3. Les limites de la récursivité

Considérons à nouveau la fonction récursive somme(n) calculant la somme des n premiers entiers naturels.

```
def somme(n):
    if n == 0:
        return 0
    return n + somme(n - 1)
```

À expérimenter sur le poste maître :

1) Que produit l'appel à somme(1000) ?

2) Que produit l'appel à somme(3000) ?

Explications : une partie de l'espace mémoire d'un programme est organisée sous forme d'une pile où sont stockés les **contextes d'exécution** de chaque appel de fonction (voir le cours sur la gestion des processus). Lorsqu'un appel récursif se termine, cela permet à la fonction appelante de retrouver son contexte propre avec ses variables dans l'état où elles étaient avant appel.

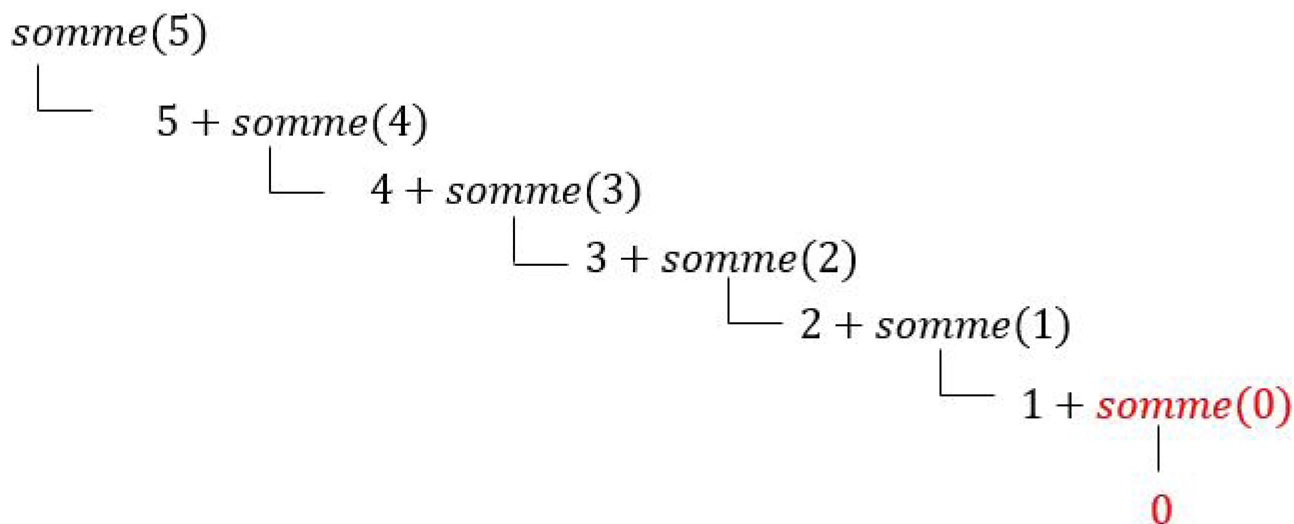
Chaque appel récursif consomme donc de la mémoire dans la pile d'exécution.

Prenons l'exemple de la fonction somme.

L'organisation de la mémoire au début de l'appel à somme(5) peut être représentée par la pile ci-dessous contenant un contexte d'exécution avec, entre autres, un emplacement pour l'argument n initialisé à 5, ainsi que d'autres valeurs (comme l'emplacement pour la valeur renvoyée par la fonction) qui sont simplement représentées par des dans le schéma ci-après.

Contexte somme(5) : n = 5 …
Contexte somme(4) : n = 4 …
Contexte somme(3) : n = 3 …
Contexte somme(2) : n = 2 …
Contexte somme(1) : n = 1 …
Contexte somme(0) : n = 0 …

Le calcul récursif de somme(5) va engendrer une suite d'appels « en cascade » à la fonction somme, que l'on peut représenter par l'arbre d'appels suivant :



En ce qui concerne l'organisation de la pile mémoire, un nouvel environnement d'exécution va être alloué dans la pile pour chacun de ces appels.

Ainsi, juste après le dernier appel à somme(0) la pile contient donc les contextes d'exécution pour les six appels à la fonction somme.

Lors de l'exécution d'une fonction récursive, chaque appel récursif conduit à un empilement du contexte dans la pile d'exécution. Lorsque la condition d'arrêt de la récursivité se produit, les différents contextes sont progressivement dépilés pour poursuivre l'exécution de la fonction.

Toutefois, puisque les appels récursifs sont effectués sur une zone de mémoire plutôt limitée, la pile d'exécution ne doit pas dépasser une certaine limite. Par défaut, elle est de 1000 dans l'implémentation courante de Python. Sinon, on obtient l'erreur constatée précédemment.

Il est cependant possible de modifier le plafond du nombre d'appels :

```
import sys
sys.setrecursionlimit(3500)

def somme(n) :
if n == 0:
return 0
return n + somme(n - 1)

somme(3000)
```

Toutefois, comme indiqué dans la documentation, pour des raisons de portabilité, on modifie avec prudence le plafond des appels en l'élevant de façon modérée.

Enfin, un programme récursif peut être plus lent que son équivalent itératif sans pour autant dépasser le nombre d'appels récursifs autorisés.

Pour illustrer cela, on s'intéresse à nouveau à la suite de Fibonacci.

Chacun des deux programmes suivants permet de calculer le terme de rang n de cette suite.

La compréhension de la « lenteur » de la version récursive est traitée dans le chapitre consacré à la programmation dynamique.

Version itérative

```
def fib_v1(n) :
    u, u1, u2 = 1, 1, 1
    for i in range(2,n+1) :
        u = u1 + u2
        u1, u2 = u, u1
    return u
```

Version récursive

```
def fib_v2(n) :
    if n == 0 or n == 1 :
        u = 1
    else :
        u = fib_v2(n-1) + fib_v2(n-2)
    return u
```

Comparaison des deux versions

```
from time import time
debut = time()
print('u(35) = ',fib_v1(35))
print("Temps d'exécution avec la version itérative : ", time()-
debut)
debut = time()
print('u(35) = ',fib_v2(35))
print("Temps d'exécution avec la version récursive : ", time()-
debut)
```

4. Bilan

- **Synthèse :** téléchargez et regardez la vidéo de Pierre Marquestaut.
- Méthode de programmation : il convient de bien avoir en tête la façon de concevoir une fonction (ou un programme) récursive :

```
def f(x) :
if #cas de base :
   #traitement direct du/des cas de base
   #ici pas d'appel récursif
else :
    #traitement du cas général par appel(s) récursif(s)
```

N'oubliez surtout pas les cas de base qui constituent la ou les condition(s) d'arrêt, sans quoi votre algorithme tournera à l'infini et votre programme s'arrêtera faute de mémoire disponible sans jamais rendre de réponse.

- Choix entre récursif et itératif : la programmation récursive est à la fois un style de programmation mais également une technique pour définir des concepts et résoudre certains problèmes qu'il n'est parfois pas facile de traiter en programmant uniquement avec des boucles. Elle s'avère extrêmement pratique lorsque la résolution d'un problème se ramène à celle d'un problème plus petit. À force de réduire notre problème, on arrive à un problème trivial que l'on sait résoudre : c'est ce qu'on utilise dans notre condition d'arrêt.
Un programmeur doit écrire une fonction récursive quand c'est la solution la plus adaptée à son problème.

Prolongements

D'autres parties du programme de terminale NSI s'appuient sur la récursivité et permettent de réinvestir ce thème. Par exemple :

- diviser pour régner : le tri fusion, le tri pivot et la rotation d'un quart de tour d'une image ;
- parcours d'arbres et de graphes en profondeur.

Sources

- Le cours d'Olivier Lecluse.
- Le site de l'I.R.E.M. de Lyon.
- Le cours sur la récursivité proposé par l'ENIB.
- Le livre *Numérique et Sciences Informatiques Terminale*, aux éditions Ellipses, de T. Balabonski, S. Conchon, J-C Fillâtre et K. NGuyen.

Des exercices d'entraînement

Consulter :

- le site de l'I.R.E.M. de Lyon ;
- le site de l'ENIB ;
- le tutoriel de Pascal Ortiz sur la récursivité propose un grand nombre d'exemples commentés et d'exercices (non corrigés).