Chap 02 : La récursivité

Compétences évaluables :

- Ecrire un programme récursif
- analyser le fonctionnement d'un programme récursif

Table des matières

1. Algorithmes récursifs

1.1. Introduction

Activité n°1.: Etudions ces deux algorithmes d'implémentation d'une fonction `decompte(n)`

Version itérative :

```
def decompte_i(n):
    while n > 0:
        print(n)
        n -= 1
    print("fin")



print(decompte_i(5))
```

Version récursive :

```
def decompte_r(n):
    if n == 0 :
        print("fin")
    else:
        print(n)
        decompte_r(n-1)


print(decompte_r(5))
```

Activité n°2.: Tester les deux fonctions sur http://pythontutor.com/visualize.html#mode=edit

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

La récursivité est une méthode de résolution de problèmes qui consiste à **décomposer le problème en sous-problèmes identiques** de plus en plus petits jusqu'à obtenir un problème suffisamment petit pour qu'il puisse être résolu de **manière triviale.**

1.2. Comment écrire une fonction récursive ?

Pour écrire une fonction récursive :

- Un (ou plusieurs) cas de base : Les valeurs d'entrées pour lesquelles on ne fait aucun appel récursif sont appelées **les cas de base**
- **Appels récursifs (cas récursif)**
  - Appels de la méthode courante
  - Chaque suite d'appels récursifs doit essentiellement se terminer sur un cas de base

```
def fonction(arguments) :
        if condition d'arrêt :
                return cas de base
        else :
                return fonction(nouveaux arguments)
```

1.3. Application à la fonction puissance

Le but est d'écrire une fonction `puissance(x,n)` sans utiliser ** de Python : On cherche à calculer l'opération de puissance *n-ième* d'un nombre *x*, c'est-à-dire la multiplication répétée *n* fois de *x* avec lui-même : $x^n = \underbrace{x \times \ldots \times x}_{n\ fois}$ sans utiliser **.

1. Cherchons **le(s) cas de base** :
   On sait que la puissance de *x* pour $n = 0$ vaut 1. (On pourrait aussi ajouter $x^1 = x$ non utile)
2. Cherchons **le *cas récursif*** c'est-à-dire le passage des valeurs renvoyées par l'appel précédent à l'appel suivant.
   On sait : $x^n = \underbrace{x \times \ldots \times x}_{n\ fois} = x \times \underbrace{x^{n-1}}_{appel\ précédent}$

Remarque : il est très important de supposer que les appels récursifs donnent les bons résultats => Faire confiance à la récursion.

Activité n°3.: implémentation de la fonction en python

```
def puissance(x, n):
    if n == 0 :
        return 1
    else:
        return x*puissance(x, n-1)

print(puissance(2,5))
```

Remarque : souvent on « oublie » de noter le « else » car si on ne fait pas le cas de base le cas récursif doit se faire par défaut. Cela donne :

```
def puissance(x, n):
    if n == 0 :
        return 1
    return x*puissance(x, n-1)

print(puissance(2,4))
```

Correction :

- **Preuve de terminaison** : A chaque appel, la valeur du paramètre (*n*) diminue strictement. L'appel récursif s'arrête dès que *n* sera égal à 0. L'algorithme s'arrête alors.
- **Correction partielle :** A chaque boucle on peut écrire :

  $$x \times puissance(x, n-1)$$

  Puis

  $$x \times x \times\ puissance(x, n-2)$$

  Puis

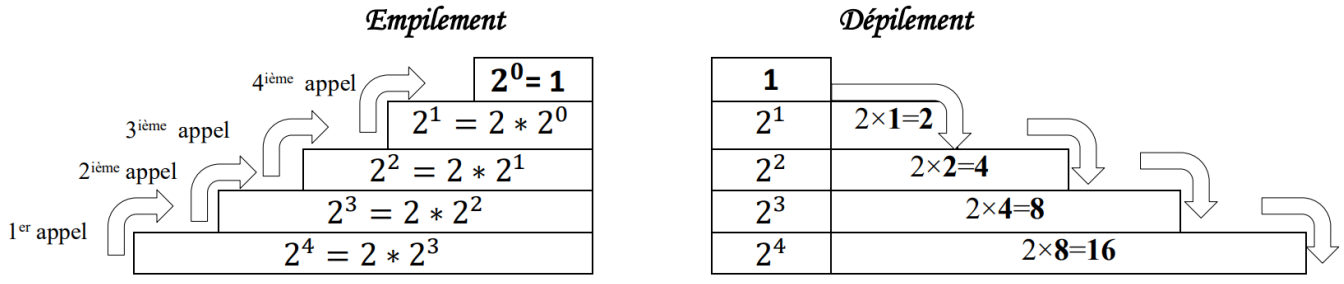
  $$x \times x \times x \times\ puissance(x, n-3)$$

  Et ainsi de suite jusqu'à n = 0

  $$\underbrace{x \times \ldots \times x}_{n\ fois} \times 1$$

  On obtient bien la fonction souhaitée

Remarque : Le principe de programmation par récursivité est basé sur le fonctionnement de « l'empilement – dépilement » à l'aide d'une pile d'exécution stockant l'adresse mémoire de la prochaine instruction machine à exécuter et conservant une "trace" des valeurs des variables. Dans le cas de `puissance(2,4)` on obtiendra :



La mise en œuvre des algorithmes récursifs nécessite le plus souvent une **pile d'exécution.**

1.4. Fonction récursive sans cas de base….

Activité n°4.: Fonction récursive sans condition d'arrêt :

```
>>> def f(n):
...     return 1+f(n+1)
...
>>> f(0)

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    f(0)
  File "<pyshell#3>", line 2, in f
    return 1 + f(n+1)
  File "<pyshell#3>", line 2, in f
    return 1 + f(n+1)
  File "<pyshell#3>", line 2, in f
    return 1 + f(n+1)
  [Previous line repeated 1022 more times]
RecursionError: maximum recursion depth exceeded
```

Ici la fonction ne s'arrêtera jamais !!

L'interpréteur Python **limite** arbitrairement le nombre d'appels récursifs (la valeur par défaut est égale à 1000).

Remarque : Pour augmenter ce nombre :

```
import sys
sys.setrecursionlimit(1500)
```

1.5. Application à la multiplication du paysan russe

La méthode du paysan russe est un très vieil algorithme de **multiplication de deux nombres entiers**. Il s'agissait de la principale méthode de calcul en Europe avant l'introduction des chiffres arabes, et les premiers ordinateurs l'ont utilisé avant que la multiplication ne soit directement intégrée dans le processeur sous forme de circuit électronique.

```
fonction multiply(x,y)
  p := 0
  tant que x > 0
       si x est impair faire
              p := p + y
       x := x // 2
       y := y * 2
  fin tant que
  return p
```

Calcul de 105 x 253 par la méthode du paysan russe, p = 0 (au départ)

x impair	V	F	F	V	F	V	V
p = p + y	253			2277		10373	26565
x = x // 2	52	26	13	6	3	1	0
y = y * 2	506	1012	2024	4048	8096	16192	32384

105 x 253 = 26565

On ramène ainsi le problème du calcul du produit de *x* par *y* à un **sous-problème**.

Activité n°5.: On peut implémenter la version itérative et la version récursive en Python ainsi :

version itérative

```
def multiply_i(x,y):
    p = 0
    while x > 0:
        if x % 2 != 0:
            p += y
        x = x // 2
        y = y * 2
    return p
```

version récursive

```
def multiply_r(x,y):
    if x <= 0: # cas de base
        return 0
    elif x % 2 == 0:
        return multiply_r(x//2 , y*2)
    else :
        return multiply_r(x//2, y*2) + y
```

```
>>> multiply_i(105,253)
26565
>>> multiply_r(105,253)
26565
```

1.6. Application au calcul de factorielle

La factorielle : **qu'est-ce que c'est ?**
$n! = n \times (n-1) \times (n \cdot 2) \times \ldots \cdot 2 \times 1$ pour n entier > 0. Cas particulier : 0 !=1.

Activité n°6.: Tester les deux implémentations suivantes :

version itérative

```
def factorielle_i(n) :
    result = 1
    while n > 0:
        result *= n
        n -=1
    return result
```

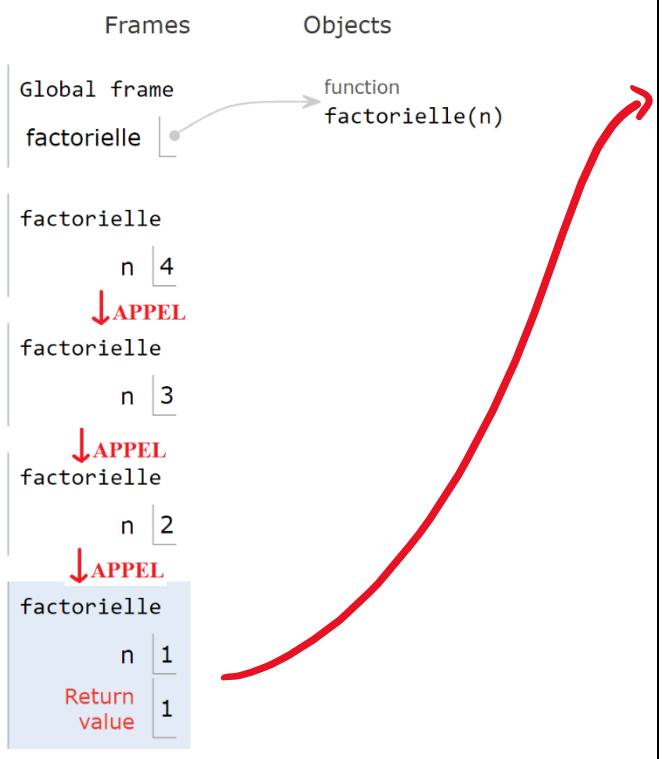
version récursive

```
def factorielle_r(n) :
    if n == 1 or n == 0:
        return 1
    return n * factorielle_r(n-1)
```

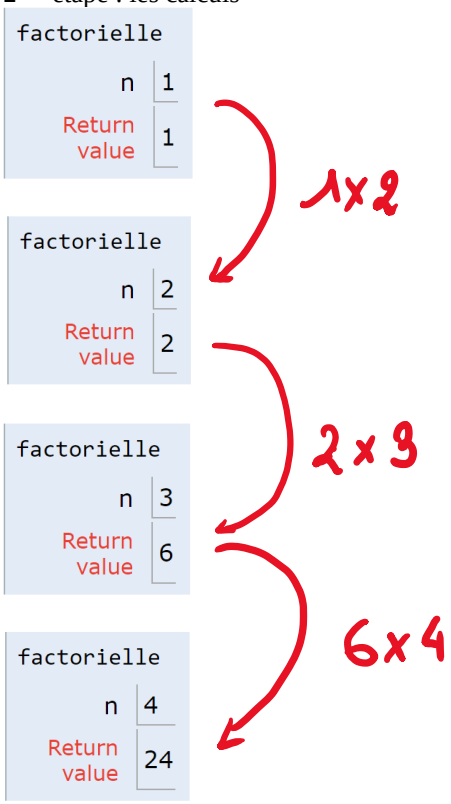
```
>>> factorielle_i(10)
3628800
>>> factorielle_r(10)
3628800
```

Par exemple : Cherchons pour la version récursive : factorielle(4) =

1ère étape : les appels successifs à la fonction



2ème étape : les calculs



> Un appel de fonction est une opération **plus couteuse** en soi qu'une opération arithmétique ou un test. C'est pourquoi on préfère souvent la fonction itérative à la fonction récursive.

Dans le cas de la fonction factorielle, on prendra plutôt la **version itérative**, mais il y a des cas où la fonction récursive est clairement préférable, par exemple pour parcourir des arbres (vu un peu plus tard) ou pour faire des tris (chapitre suivant).

1.7. Application aux tours de Hanoï

Le casse-tête des tours de Hanoï est un jeu de réflexion consistant à déplacer des disques de diamètres différents d'un tour de « départ » à une tour d' « arrivée » en passant par une tour « intermédiaire » et ceci en un minimum de coups, tout en respectant les règles suivantes :

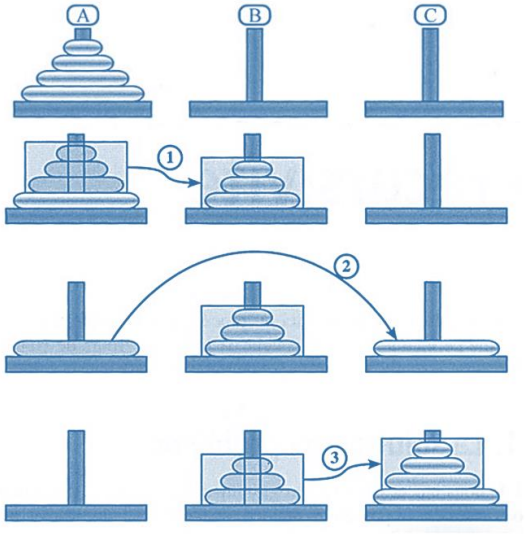
- **on ne peut pas déplacer plus d'un disque à la fois**,
- **on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide**.

On suppose que cette dernière règle est également respectée dans la configuration de départ.

Vidéo : les tours de Hanoi : https://www.youtube.com/watch?v=U3nGNJTxYc4

Pour résoudre le problème des tours de Hanoï, il faut raisonner récursivement.

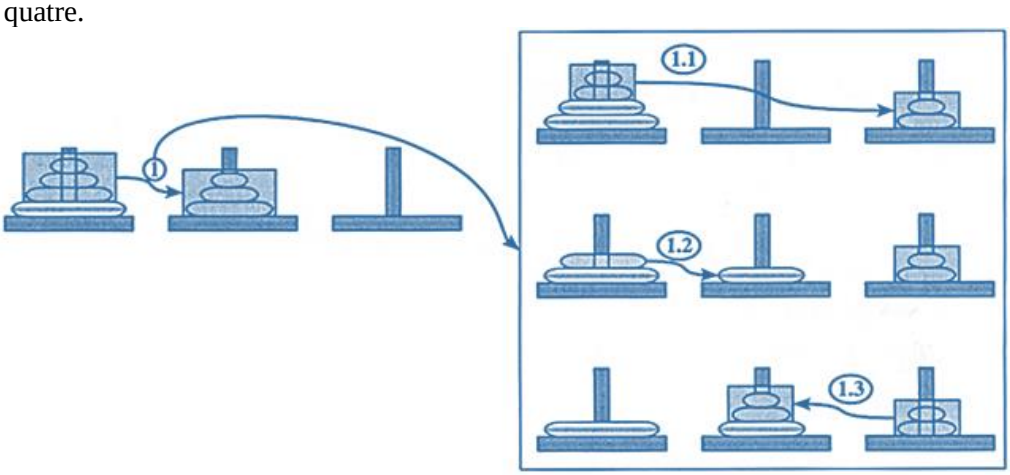
- **La situation de départ** :



- Avec quatre disques, **le plus grand** ne peut être posé sur aucun autre : il est donc le plus contraignant à déplacer. Il faudra **le déplacer de la pique A vers la pique C**, et cela ne pourra se faire que si aucun disque n'est présent sur C. Donc il faudra que le grand disque soit seul sur la pique A et les trois autres disques sur la pique B.

  On vient ainsi de réduire le problème : **pour pouvoir déplacer 4 disques de A vers C, il faut d'abord déplacer 3 disques de A vers B, puis déplacer le grand sur C et enfin déplacer les trois disques de B vers C.**

- Autre remarque, le grand disque est le seul sur lequel on peut poser n'importe lequel des autres disques. Donc, dans **le cadre d'un déplacement des trois disques supérieurs, tout se passe comme s'il n'était pas là**. On peut traiter le problème du déplacement des trois disques exactement de la même manière que nous avons traité celui des quatre.



- L'étape 3 pourrait être décomposée de la même façon.

En résumé, déplacer *n* disques de A vers C en passant par B consiste à :

1. déplacer *(n-1)* disques de A vers B (en passant par C);
2. déplacer le plus grand disque de A vers C ;
3. déplacer *(n-1)* disques de B vers C (en passant par A).

Les étapes 1 et 3 peuvent elles-mêmes se décomposer selon le même principe, sauf que les rôles des paquets sont intervertis. Par exemple, dans la première, on va de A vers B, donc forcément par l'intermédiaire de C. Voici la marche à suivre, donnée sur deux niveaux :

1. déplacer *(n-1)* disques de A vers B (en passant par C) ;
   1.1. déplacer *(n-2)* disques de A vers C (en passant par B) ;
   1.2. déplacer un disque de A vers B ;
   1.3. déplacer *(n-2)* disques de C vers B (en passant par A).
2. déplacer le plus grand disque de A vers C ;
3. déplacer *(n-1)* disques de B vers C (en passant par A).
   3.1. déplacer *(n-2)* disques de B vers A (en passant par C) ;
   3.2. déplacer un disque de B vers C ;
   3.3. déplacer *(n-2)* disques de A vers C (en passant par B).

Et ainsi de suite...

Remarque : Un jeu à 64 disques requiert un minimum de 2^{64} -1 déplacements. En admettant qu'il faille 1 seconde pour déplacer un disque, ce qui fait 86 400 déplacements par jour, la fin du jeu aurait lieu au bout d'environ 213 000 milliards de jours, ce qui équivaut à peu près à 584,5 milliards d'années, soit 43 fois l'âge estimé de l'univers (13,7 milliards d'années selon certaines sources)

Activité n°7.: Ce qui donne en Python :

```
def hanoi(n, a="A", b="B", c="C"):
    if n == 0: # cas de base
        return None
    hanoi(n-1, a, c, b) # de A vers B en passant par C
    print("Déplacer le disque {} de la pique {} vers la pique {}.".format(n, a, c))
    hanoi(n-1, b, a, c) # de B vers C en passant par A
```

```
>>> hanoi(4)
Déplacer le disque 1 de la pique A vers la pique B.
Déplacer le disque 2 de la pique A vers la pique C.
Déplacer le disque 1 de la pique B vers la pique C.
Déplacer le disque 3 de la pique A vers la pique B.
Déplacer le disque 1 de la pique C vers la pique A.
Déplacer le disque 2 de la pique C vers la pique B.
Déplacer le disque 1 de la pique A vers la pique B.
Déplacer le disque 4 de la pique A vers la pique C.
Déplacer le disque 1 de la pique B vers la pique C.
Déplacer le disque 2 de la pique B vers la pique A.
Déplacer le disque 1 de la pique C vers la pique A.
Déplacer le disque 3 de la pique B vers la pique C.
Déplacer le disque 1 de la pique A vers la pique B.
Déplacer le disque 2 de la pique A vers la pique C.
Déplacer le disque 1 de la pique B vers la pique C.
```

Pour mieux comprendre : http://accromath.uqam.ca/2016/02/les-tours-de-hanoi-et-la-base-trois/

2. Les dangers de la récursivité

Utiliser une fonction récursive n'est pas toujours une bonne idée.
Soit la suite de Fibonacci : 1, 1, 2, 3, 5, 8, 13, 21, 34,…
Par définition :

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-2} + F_{n-1} \; pour \; n > 1$$

On écrit deux fonctions (une récursive et une itérative) qui calculent le k-ième terme de cette suite, puis on comparera les temps de calcul.

Activité n°8.: implémenter les deux algorithmes suivants :

Version itératif

```
def fibo_it(n):
    fn_moins_2 = 0
    fn_moins_1 = 1
    i = 2
    if n == 0:
        return 0
    elif n == 1 or n == 2 :
        return 1
    else:
        while i <= n :
            fn = fn_moins_2 + fn_moins_1
            fn_moins_2 = fn_moins_1
            fn_moins_1 = fn
            i += 1
        return fn
```

Version récursif

```
def fibo_recur(n):
    if n == 0 : # cas de base
        return 0
    elif n == 1 or n == 2 : # 2 autres cas
        return 1
    return fibo_recur(n-1) + fibo_recur(n-2)
```

```
>>> fibo_it(10)
55
>>> fibo_recur(10)
55
```

Activité n°9.: On va comparer les temps de calcul pour chaque algorithme. Ajouter à la suite des deux algorithmes précédents :

```
import time

for k in range (5):
    print((k+1)*10)
    a = time. perf_counter_ns()
    fibo_it((k+1)*10)
    b = time. perf_counter_ns()
    print("itératif :", b-a, "s")
    a = time.perf_counter_ns()
    fibo_recur((k+1)*10)
    b = time.perf_counter_ns()
    print("récursif :", b-a, "ns")
```

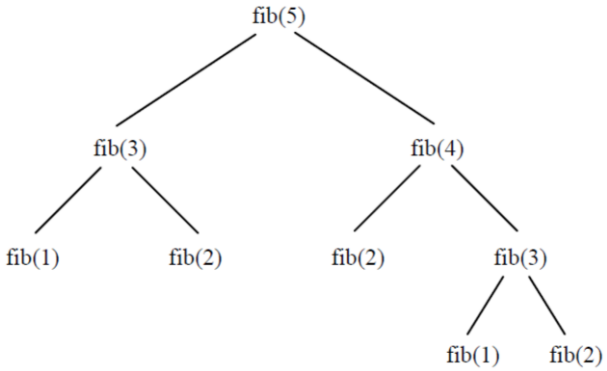
Voici une table des résultats (ces temps sont approximatifs et dépendent évidemment du processeur, mais ils mettent en évidence les problèmes de la récursivité) :

	Fonction récursive (*fib1*)	Fonction itérative (*fib2*)
k	**Temps en ns**	**Temps en ns**
10	26 100	2 400
20	2 359 600	3 100
30	397 709 900	5 300
40	42 835 094 000 = 42,835s	6 900
50	7 264 000 000 = 7 264 s	8 200

Pourquoi une telle différence ?
Observons comment se passe le calcul récursif. Calculons fib(5) avec la méthode récursive :

```
fib(5) ->     fib(4) + fib(3)
              ->     (fib(3) + fib(2)) + fib(3)
              ->     (fib(2) + fib(1)) + fib(2)) + fib(3)
              ->     (( 1 + fib(1)) + fib(2)) + fib(3)
              ->     (( 1 + 1 ) + fib(2)) + fib(3)
              ->     ( 2 + fib(2)) + fib(3)
              ->     ( 2 + 1 ) + fib(3)
              ->     3 + fib(3)
              ->     3 + (fib(2) + fib(1))
              ->     3 + (1 + fib(1))
              ->     3 + (1 + 1 )
              ->     3 + 2
              ->     5
```

On peut aussi représenter les appels de fonction dans un arbre (arbre d'appels).



On voit que ce n'est pas efficace : par exemple, *fib(3)* est appelé deux fois, ce qui est une perte de temps. De plus on imagine bien que l'arbre va devenir très vite gigantesque, avec un très grand nombre d'appels inutiles.

« **Marcher, en itératif**, c'est mettre un pied devant l'autre et recommencer.
Marcher en récursif, c'est mettre un pied devant l'autre et marcher. »

3. Exercices

Exercice n°1 : La fonction somme

Pour définir la somme des n premiers entiers, on a l'habitude d'écrire la formule suivante 1 : 0+1+2+ ⋯ + n
Ecrire une fonction `somme(n)` en récursif
Aide :

1. Déterminer le(s) cas de base
2. Déterminer le cas récursif

Exercice n°2 : Le palindrome

On appelle palindrome un mot qui se lit dans les deux sens comme "selles" ou "radar"
Le fonction ci-contre renvoie vrai si le mot passé en paramètre est un palindrome. Pour le mot "Selles" composé de 6 lettres, on fait 3 comparaisons. Pour le mot "Radar" composé de 5 lettres on ne fait qu e 2 comparaisons (une unique lettre est forcément un palindrome)

Version itérative

```
def est_palindrome(mot):
    mot=mot.lower()
    for i in range(len(mot)//2):
        if mot[i]!=mot[-i-1]:
            return False
    return True
```

En version récursive, l'idée est : "selles" est un palindrome si "s"= "s" et "elle" est un palindrome => cas récursif

Ecrire une version récursive de la fonction `est_palindrome(mot)` .
Aide :

1. Quel est les cas de base (cas d'arrêt)
   a. Pour renvoyer True
   b. Pour renvoyer False
2. Déterminer le cas récursif

Exercice n°3 : Nombre d'adhérents

Une association a remarqué que d'une année à l'autre :

- Elle perd 5% de ses adhérents
- Elle gagne 200 adhérents

1) En admettant que le nombre d'adhérents de cette association était égal à 2000 au 1 er janvier 2020, écrire en python une fonction récursive nommée `nombre(n)` affichant le nombre théorique d'adhérents après n année.
   a. Déterminer le cas de base
   b. Déterminer le nombre qu'elle a l'année suivante par rapport à l'année précédente
2) Dans ce même programme, afficher le nombre théorique d'adhérents au cours des 20 prochaines années.

Exercice n°4 : La suite de Syracuse

Elle est définie par :

$$x_1 = a \in \mathbb{N}^*$$
$$x_{n+1} = \begin{cases} \frac{x_n}{2} \; si \; x_n \; est \; pair \\ 3x_n + 1 \; si \; x_n \; est \; impair \end{cases}$$

1. Vérifier par le calcul que pour a = 14 et n = la suite est des nombres : 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2
2. Ecrire en Python une fonction itérative `syracuse_iter(a, n)` : donnant la suite de Syracuse lorsqu'on entre en paramètre la valeur de a est le rang n. Vérifier les tests suivants :

```
assert syracuse_iter(14,1) == 14
assert syracuse_iter(14,3) == 22
assert syracuse_iter(14,20) == 2
```

3. Ecrire une version récursive `syracuse_recur(a, n)`. Vérifier les tests suivants :

```
assert syracuse_recur(14,1) == 14
assert syracuse_recur(14,3) == 22
assert syracuse_recur(14,20) == 2
```

Exercice n°5 : Le PGCD

Ecrire une fonction récursive `pgcd(a, b)` qui renvoie le PGCD de deux entiers a et b. Intitulé le fichier `pgcd.py`

On utilisera l'algorithme d'Euclide :

- pour tout entiers a et b, on a pgcd(a;b) = pgcd(a −b;b).
- pour tout entier a, on a pgcd(a; 0) = a.

Exemple `pgcd(96,36)=12` et `pgcd(60,32)=4`

```
pgcd(96, 36) :
   96 = 36 x 2 + 24
   36 = 24 x 1 + 12
   24 = 12 x 2 + 0
   12 = 0 x 0 + 12
```

⇨ 12

```
pgcd(60, 32) :
   60 = 32 x 1 + 28
   32 = 28 x 1 + 4
   28 = 4 x 7 + 0
   4 = 0 x 0 + 4
```

⇨ 4

Exercice n°6 : Nombre de chiffres

Écrire une fonction récursive `nombre_de_chiffres(n)` qui prend un entier positif ou nul n en argument et renvoie son nombre de chiffres. Intitulé le fichier `nombre_chiffres.py`

Par exemple, `nombre_de_chiffres(34126)` doit renvoyer 5.

Exercice n°7 : Appartient

Écrire une fonction récursive `appartient(v, t, i)` prenant en paramètres une valeur v, un tableau t et un entier i et renvoyant `True` si v apparaît dans t entre l'indice i (inclus) et len (t) (exclu), et `False` sinon.

On supposera que i est toujours compris entre 0 et len(t). Intitulé le fichier `appartient.py`

Par exemple :

```
t = [1,3,5,6,7,9,10]
appartient(7,t,5) = False
appartient(7,t,3) = True
```

Exercice n°8 : Le triangle de Pascal

Le triangle arithmétique de Pascal est le triangle dont la ligne d'indice n (n = 0, 1, 2...) donne les coefficients binomiaux C_n^p pour p = 0, 1, 2..., n.

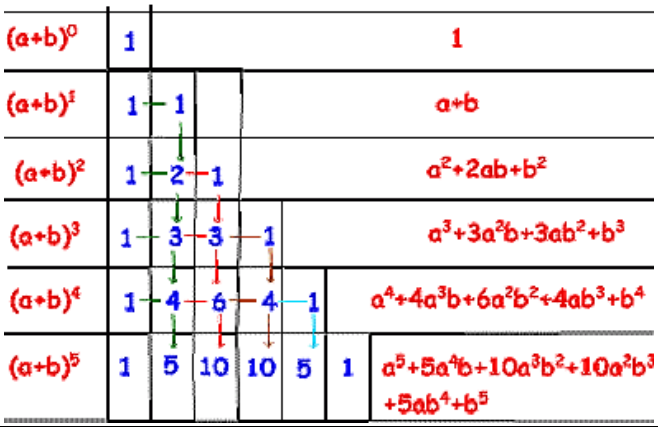
	p=0	1	2	3	4	5	6	7	8	9	10	11	12
n=0	1												
1	1	1											
2	1	2	1										
3	1	3	3	1									
4	1	4	6	4	1								
5	1	5	10	10	5	1							
6	1	6	15	20	15	6	1						
7	1	7	21	35	35	21	7	1					
8	1	8	28	56	70	56	28	8	1				
9	1	9	36	84	126	126	84	36	9	1			
10	1	10	45	120	210	252	210	120	45	10	1		
11	1	11	55	165	330	462	462	330	165	55	11	1	
12	1	12	66	220	495	792	924	792	495	220	66	12	1

Coefficients du développement de $(a + b)^n$.

Les coefficients du triangle de Pascal sont les coefficients du développement de $(a + b)^n$.

Par exemple :

- La ligne 0 est : 1 soit le coefficient de :$(a + b)^0 = 1$.
- La ligne 1 est : 1 - 1 soit les coefficients de : $(a + b)^1 = 1\times a + 1\times b$.
- La ligne 2 est : 1 - 2 - 1, soit les coefficients de : $(a + b)^2 = 1\times a^2 + 2\times ab + 1\times b^2$.
- La ligne 3 est : 1 - 3 - 3 - 1, soit les coefficients de : $(a + b)^3 = 1\times a^3 + 3\times a^2b + 3\times ab^2 + 1\times b^3$.
- La ligne 4 est : 1 - 4 - 6 - 4 - 1, soit les coefficients de : $(a + b)^4 = 1\times a^4 + 4\times a^3b + 6\times a^2b^2 + 4\times ab^3 + 1\times b^4$.

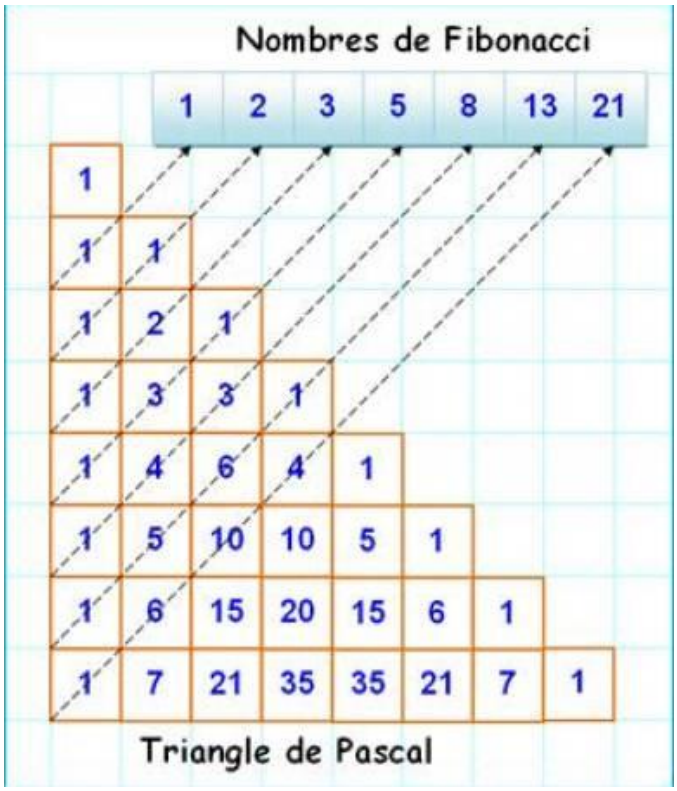


En analyse combinatoire.
Les nombres C_n^p correspondent, en analyse combinatoire, au nombre de façons de tirer p objets parmi n.
Par exemple :

- La ligne 5 est : **1 - 5 - 10 - 10 - 5 - 1** donc
- $C_5^0 = 1$: Il y a 1 seule façon de tirer 0 objet parmi 5.
- $C_5^1 = 5$: Il y a 5 façons de tirer 1 objet parmi 5.
- $C_5^2 = 10$: Il y a 10 façons de tirer 2 objets parmi 5.
- $C_5^3 = 10$: Il y a 10 façons de tirer 3 objets parmi 5.
- $C_5^4 = 5$: Il y a 5 façons de tirer 4 objets parmi 5.
- $C_5^5 = 1$: Il y a 1 seule façon de tirer 5 objets parmi 5.

Le triangle de Pascal Suite de Fibonacci

Remarquer que ...



Le triangle de Pascal (nommé ainsi en l'honneur au mathématicien Blaise Pascal) est une présentation des coefficients binomiaux sous la forme d'un triangle défini ainsi de manière récursive :

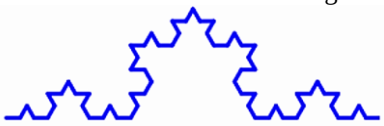
$$C(n,p) = \begin{cases} 1 \text{ si } p = 0 \text{ ou } n = p, \\ C(n-1,p-1) + C(n-1,p) \text{ sinon.} \end{cases}$$

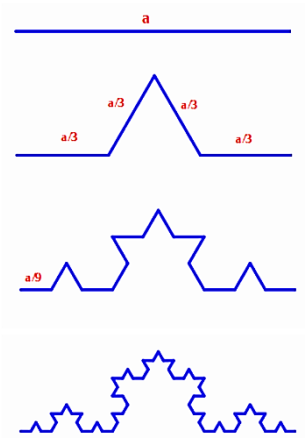
Écrire une fonction récursive `C(n,p)` qui renvoie la valeur de C(n,p). Intitulé le fichier `coef_binomial.py`
Exemple : `C(10,5) = 252`

4. Projet (démarche d'investigation)

Projet 1 : Le flocon de Koch

Le flocon de Koch est l'une des premières courbes fractales à avoir été décrite (bien avant l'invention du terme « fractal(e)»). Elle a été inventée en 1904 par le mathématicien suédois Helge von Koch.

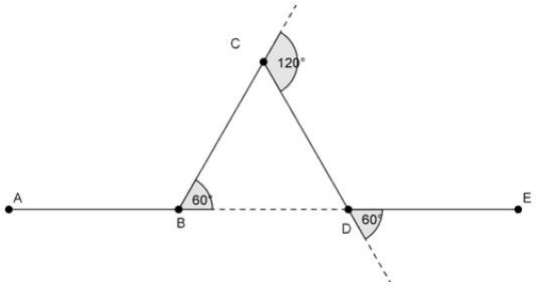




Méthode de construction :

- On commence par un segment de longueur a;
- On coupe ce segment en 3 parties égales;
- le segment central est remplacé par un triangle équilatéral de coté a/3;
- Chaque segment de longueur a/3 est lui même découpé en trois parties égales (donc de longueur a/9)
- On remplace la partie centrale par un triangle équilatéral de côté a/9;
- etc...

On décide à l'avance quand on doit s'arrêter



1. Analyser ce script et le faire fonctionner

```
import turtle
turtle.forward(100)
turtle.left(120)
turtle.forward(100)
turtle.left(120)
turtle.forward(100)
```

2. Une autre partie de script à analyser et à faire fonctionner

```
import turtle as t
# déplace la tortue aux coordonnées
t.penup()
t.goto(-100, 0)
t.pendown()
# orientation initiale de la tête :
#vers la droite de l'écran
t.setheading(0)
t.hideturtle() # on cache la tortue
t.speed(0) # on accélère la tortue
t.color('blue')
t.pensize(3)
t.forward(100)
t.left(60)
t.forward(100)
t.right(120)
t.forward(100)
t.left(60)
t.forward(100)
```

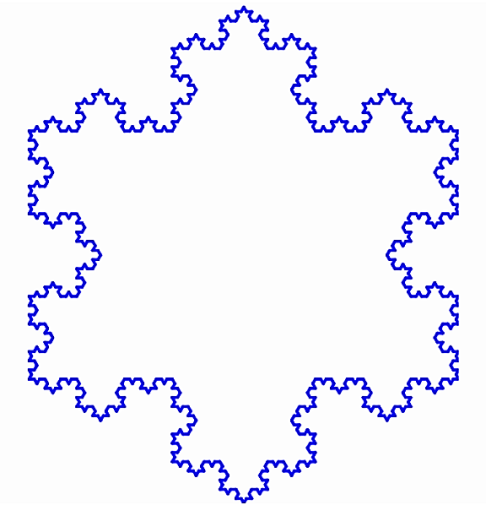
3. Compléter l'algorithme suivant :

```
fonction koch(longueur,n):
      Si n = 0 alors
            On trace le segment de longueur cote
      Sinon
            On appelle la fonction flocon avec les paramètres ???
            On tourne de ???
            ???
            ???
            ???
            ???
            ???
```

4. Implémenter l'algorithme

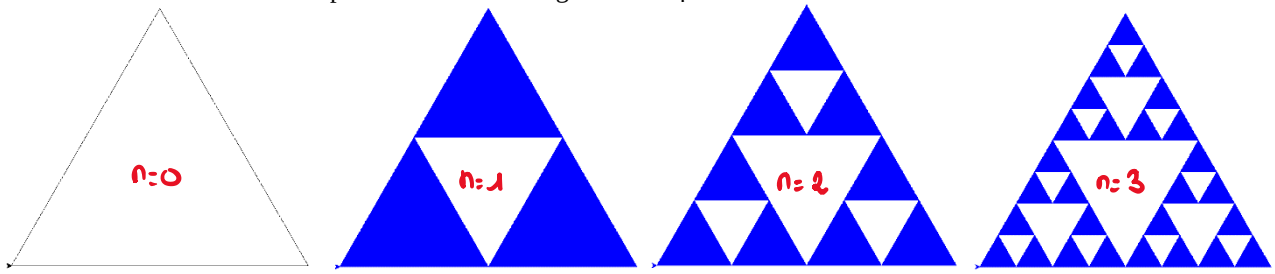
```python
import turtle

def koch(longueur, n):
    if n == 0:
        # à completer
    else:
        # à completer

koch(200,3)
```

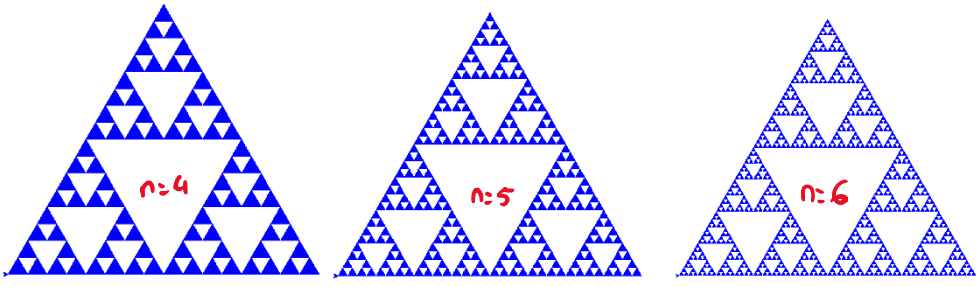
5. Cerise sur le gateau : Sauriez vous faire afficher cette figure :



Projet 2 Le triangle de Sierpinski

Utiliser une fonction récursive pour réaliser les triangles de Sierpinski ci-dessous.





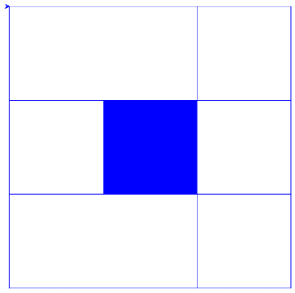
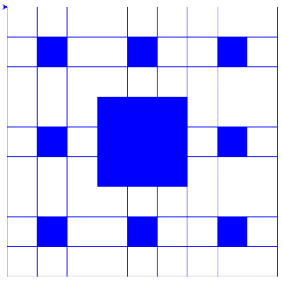
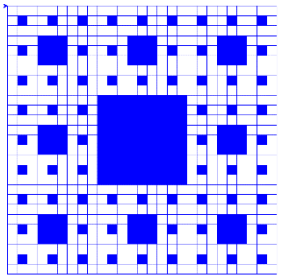
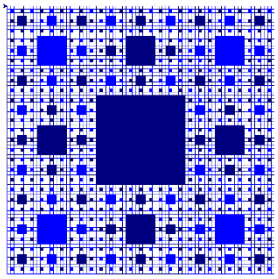
Selon la valeur du nombre du niveau on aura une représentation plus ou moins « dentelée »

Aides :

- seuls les petits triangles sont coloriés => mauvaise idée de tout coloriée puis de rajouter des triangles blancs
- pour passer d'un triangle de niveau supérieur vers un triangle inférieur il faut diviser par 2
- La fonction à créer a pour prototypage : `sierpinski(n : int, L : int)` où n est le niveau souhaité (de 0 à l'infini) et L la longueur d'un des côtés du grand triangle
- On **pourra** prendre L = 600 et se déplacer au départ en (-300, -300) pour center le dessin si on choisit d'aller vers la gauche…
- Pensez à `turtle.speed(0)` au début et `turtle.done()` à la fin
- Au-dessus du 6ème niveau il faut augmenter L pour pouvoir le voir

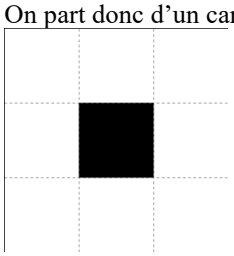
Projet 3 Le tapis de Sierpinski

Utiliser une fonction récursive pour réaliser les triangles de Sierpinski ci-dessous. L'idée est d'arrivée à un beau tapis de ce style (le dernier)

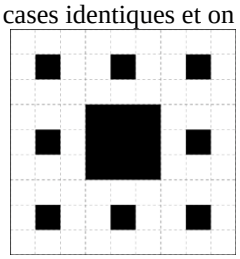
Le principe mathématique

On part donc d'un carré vide, que l'on s'empresse de découper en 9 cases identiques, puis on remplit le carré central:



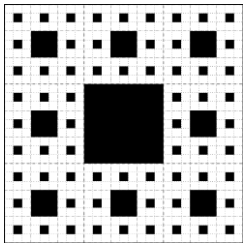
Étape 1

Ensuite, dans chaque carré vide, on fait la même chose que dans le carré initial : on le découpe en 9 cases identiques et on remplit le carré central:



Étape 2

On recommence avec les nouveaux carrés vides:

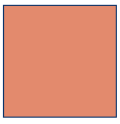


Et on fait cela à l'infini. Le résultat final donne le *tapis de Sierpinsky*.

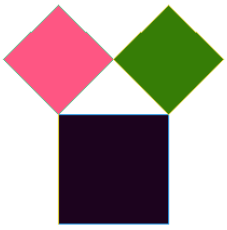
Projet 4 Arbre de Pythagore

Proposer une tortue python récursive réalisant :

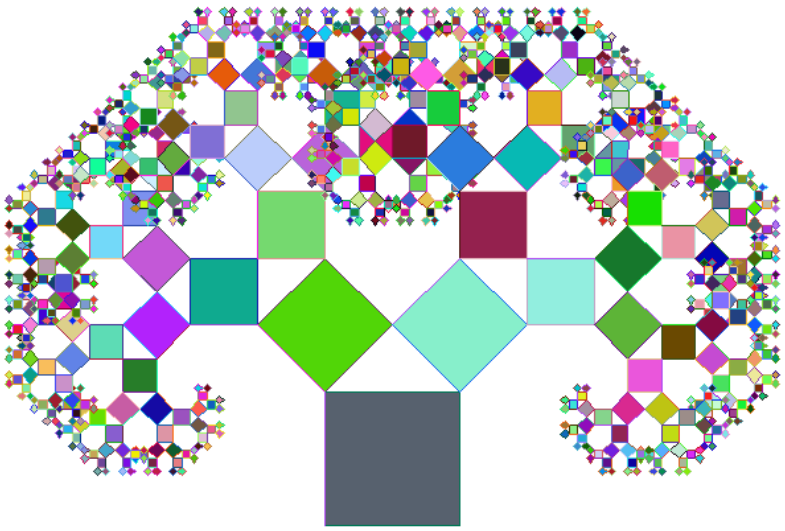
- Avec p = 1 : pythagore 1



- Avec p = 2 : pythagore 2



- Avec p = 3 : pythagore 3



- Avec p = 10 : pythagore 10

Aide :

- On pourra utiliser `turtle.colormode(255)` => pour coder en rgb puis un `random.randint(0,255)` sur les trois couleur rgb

Projet 5 Arbre de la forêt

Long en exécution…