1 Quelques généralités sur la notion de récursivité

Voici quelques éléments à avoir à l'esprit pour aborder cette notion :

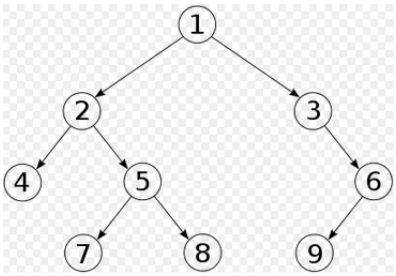
- La **récursivité** est une démarche qui fait référence à l'objet même de la démarche à un moment du processus.
- C'est une démarche dont la description mène à la répétition d'une même règle.
- C'est un processus dépendant de données et faisant appel à ce processus sur d'autres données « plus simples ».
- La récursivité peut consister, comme les fractales, à montrer une image qui contient des images similaires à une autre échelle.
- La récursivité est un concept qui est définit en invoquant le même concept.
- **La récursivité, c'est définir une structure à partir de l'une au moins de ses sous-structures**.
- On pourra retenir que la notion de récursivité est synonyme d'**auto-référence**.

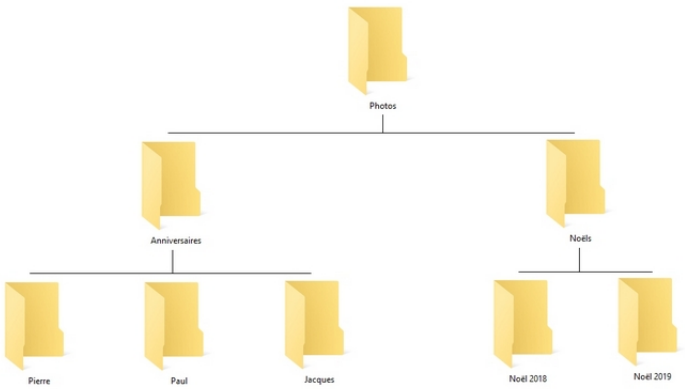
Beaucoup d'objets en informatique, de part leur construction, se prêtent à des traitements récursifs.

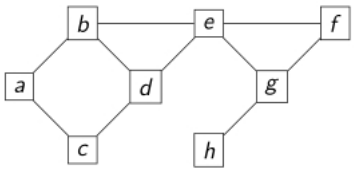
On peut citer par exemple :

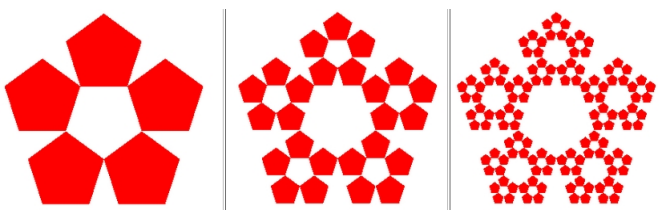
- les **listes**, les **piles** , les **files** et les **listes chainées** et d'une manière générale toutes les structures **linéaires**
- les **arbres binaires**, et d'une manière toutes les structures **arborescentes**
- les **graphes**

Voici quelques exemples de structures qui permettent des traitement récursifs en informatique.









2 Un exemple : la somme des n premiers entiers

On note S_n la somme de n premiers entiers. Par définition, on a donc : $S_n = 1+2+3+...+n$.

La **méthode classique** du point de vue algorithmique pour calculer cette somme utilise une boucle et la notion d'accumulateur. Voici le code :

```
def somme_boucle(n) :
        s = 0
        for i in range(n+1):
                s = s + i
        return s
```

Même si cette fonction répond parfaitement au problème, on peut remarquer que son écriture ne fait clairement référence à la formule de S_n écrite au-dessus. Si on ne le savait pas, il faudrait lire attentivement le code pour analyser ce qu'il fait. Programmer et tester cette fonction.

La **méthode récursive** est ici possible, et même plus en lien avec la formule de S_n. Elle consiste à définir S_n avec une **fonction mathématique** de la manière suivante :

$$somme(n) = \begin{cases} 0 & \text{si } n = 0 \\ n + somme(n-1) & \text{si } n > 0 \end{cases}$$

En utilisant cette fonction pour les premiers entiers, on voit que :

- $somme(0) = 0$; c'est le cas simple, il n'y a pas d'appel récursif.
- $somme(1) = 1 + somme(0) = 1 + 0 = 0$; l'appel de $somme(1)$ provoque un appel récursif.
- $somme(2) = 2 + somme(1) = 2 + 1 = 3$; l'appel de $somme(2)$ provoque deux appels récursifs.
- $somme(3) = 3 + somme(2) = 3 + 3 = 6$; l'appel de $somme(3)$ provoque trois appels récursifs...

La définition de $somme(n)$ dépend de $somme(n-1)$. L'intérêt d'une telle définition est qu'elle est directement calculable, c'est à dire exécutable par un ordinateur. Voici l'écriture récursive de S_n :

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

On remarque que la fonction *somme* s'appelle avec un « degré » inférieur.
Exemple : voici les appels récursifs pour somme(3) = return 3 + (return 2 + (return 1 + (return 0)))

> On dit de toute fonction qui contient au moins appel récursif que c'est une **fonction récursive**.

Remarque : il peut y avoir plusieurs appels récursifs dans une même fonction.
On retiendra que la **bonne pratique** pour écrire une formulation récursive consiste à :

1. envisager d'abord le (ou les) cas simple(s), qui ne génèrent pas d'appel récursif ;
2. écrire le (ou les) cas récursif(s), qui font appel à un « degré inférieur ».

Important : une formulation récursive demande une grande rigueur pour certaines choses qui peuvent paraître « évidentes ». Il faut aussi s'assurer qu'un moment donné, un appel récursif tombera sur un « cas simple ». De plus on rappel qu'en Python, les variables des fonctions sont **locales**, et donc à chaque appel ce sont de nouvelles instances.

3 D'autres exemples de formulations récursives

3.1 La puissance $n^{ième}$ de x

Par définition , la puissance $n^{ième}$ de x est :

$$x^n = \underbrace{x \times x \times ... \times x}_{n \text{ facteurs égaux à } x}$$

Le cas simple est $n = 0$, et pour $n > 0$, on a $x^n = x \times x^{n-1}$.
Voici donc la formulation mathématique récursive de la puissance x^n :

$$puissance(n,x) = \begin{cases} 1 & \text{si } n = 0 \\ x \times puissance(n-1,x) & \text{si } n > 0 \end{cases}$$

Cet exemple sera complété en exercice, où on abordera notamment la notion de complexité.

► Programmer la fonction `puissance(n,x)` en Python et vérifier que `puissance(3,5)` donne 125.

3.2 Un exemple avec une suite récurrente

Il y a beaucoup d'analogie (on peut parler de dualité) entre les suites numériques définies par une relation de récurrence et la récursivité. Pour illustrer ce fait, nous allons utiliser la suite $(u_n)_{n\in\mathbb{N}}$ définie par :

$$u_0 = 7 \quad \text{et} \quad u_{n+1} = 3u_n + 4.$$

Une formulation récursive de cette suite est :

$$u_n = \begin{cases} 7 & \text{si } \quad n = 0 \\ 3u_{n-1} + 4 & \text{si } \quad n > 0 \end{cases}$$

► Programmer la fonction `u(n)` en Python et vérifier que `u(5)` donne 2185.

3.3 Un exemple de récursivité multiple

On appelle **récursivité multiple** une fonction qui fait appel dans sa définition à plusieurs appels de cette même fonction.

Pour donner un exemple de récursivité multiple, nous allons utiliser la célèbre suite de Fibonacci. Pour rappel, cette suite a ses deux premiers termes égaux à 1, et les suivants sont calculés en additionnant les deux termes précédents.
Voici les premiers termes de cette suite : (1;1;2;3;5;8;13;...)
Voici la formulation récursive de la suite de Fibonacci, que nous noterons simplement `fibo(n)` :

$$fibo(n) = \begin{cases} 1 & \text{si } \quad n = 0 \\ 1 & \text{si } \quad n = 1 \\ fibo(n-1) + fibo(n-2) & \text{si } \quad n > 1 \end{cases}$$

On voit clairement que cette fonction fait appel à deux appels de cette même fonction, avec des rangs inférieurs.

► Programmer la fonction `fibo(n)` en Python et vérifier que `fibo(0)` donne 1, `fibo(1)` donne 1 et `fibo(10)` donne 89.

D'autres exemples de récursivité...

Pour conclure cette partie, on notera qu'il existe d'autres exemples de récursivité, comme les **récursions imbriquées** ou les **récursions mutuelles**. Elles seront abordées en exercice.

4 Veiller à la cohérence et la bonne formulation récursive

Exemple 1

On s'intéresse à la fonction f définie par :

$$f(n) = \begin{cases} 1 & \text{si} \quad n = 0 \\ n + f(n+1) & \text{si} \quad n > 0 \end{cases}$$

On voit ici que cette formulation récursive pose problème, car on ne peut pas atteindre le cas simple. En effet, on a :

$$f(1) = 1 + f(2) = 1 + 2 + f(3) = \ldots$$

Exemple 2

On s'intéresse à la fonction g définie par :

$$g(n) = \begin{cases} 1 & \text{si} \quad n = 0 \\ n + g(n-2) & \text{si} \quad n > 0 \end{cases}$$

On voit ici que cette formulation récursive pose problème, car dans l'appel récursif, on peut tomber sur une valeur non définie. En effet, on a :

$$g(1) = 1 + g(-1)$$

et la valeur $g(-1)$ n'est pas calculable avec la définition de g.

Exemple 3

On s'intéresse à la fonction h définie par :

$$h(n) = \begin{cases} 1 & \text{si} \quad n = 0 \\ n + h(n-1) & \text{si} \quad n > 1 \end{cases}$$

En regardant attentivement, on voit qu'il y a une valeur qui a été oubliée ! C'est le cas où $n = 1$...

A faire : programmer et tester les fonctions proposées sur cette page.

5 Programmer avec des fonctions récursives

La programmation d'une fonction récursive est **relativement simple** en Python, mais elle a tendance à être **gourmande sur la pile d'appels**. On se rappelle que les fonctions en Python définissent des **variables locales**, qui sont en quelques sortes des « bulles d'exécutions », et que l'appel récursif d'une fonction crée un nouvelle bulle, et par un **retour** dans la fonction.

Certains langages ne gèrent pas la récursivité. D'autres, comme les langages **fonctionnels**, pour lesquels le paradigme est « tout est fonction, les fonctions sont des valeurs » sont bien plus rapides pour gérer les appels récursifs.

Python est limité dans le nombre d'appels récursifs (1000 en général). Il est possible de modifier cette valeur maximale d'appels récursifs, avec la bibliothèque sys et la valeur setrecursionlimit, mais dans la pratique c'est sans intérêt car le temps d'exécution devient ingérable...

6 Exercices

Exercice 1 - Calcul de factorielle(n) .

1. Donner une **formulation récursive** de $n! = 1 \times 2 \times ... \times n$
2. Écrire et programmer une fonction récursive `factor(n)` qui calcule $n!$, puis tester que `factor(5)` donne 120.

Exercice 2 - La suite de Syracuse .

Pour tout entier n, on définit sa suite de Syracuse de la manière suivante : le premier terme est $u_0 = n$, et on passe d'un terme au suivant en respectant le principe :

- si u_n est pair, alors $u_{n+1} = u_n \div 2$,
- sinon $u_{n+1} = u_n \times 3 + 1$.

L'objectif de cet exercice est d'écrire une fonction récursive qui affiche un à un les termes de cette suite. On appellera cette suite `syracuse(n)`.

1. A la main, écrire `syracuse(10)`

   Même si personne n'a réussi à le prouver, on admettra le conjecture suivante : « pour tout nombre n, sa suite de Syracuse atteint la valeur 1, et à partir de là se succèdent 4, 2, 1...

   En conséquence, on admettra que l'affichage des termes de la suite pourra s'arrêter dès que 1 apparait.
2. Donner une formulation récursive de la suite de Syracuse.
3. Écrire et programmer une fonction récursive `syracuse(n)` qui permet d'afficher les valeurs successives de la suite de Syracuse de n.

   Tester que `syracuse(10)` affiche 10;5;16;8;4;2;1, et enfin tester d'autres valeurs.

Exercice 3 - Une suite récurrente d'ordre 2 .

Pour tout entier n, on définit la suite $(u_n)_{n\in\mathbb{N}}$ avec les nombres réels quelconques a et b par la relation de récurrence :

$$u_n = \begin{cases} a & \text{si } n = 0 \\ b & \text{si } n = 1 \\ 3u_{n-1} + 2u_{n-2} + 5 & \text{si } n \geq 2 \end{cases}$$

Écrire une fonction récursive `suite(n,a,b)` qui renvoie le n-ième terme de cette suite pour des valeurs a et b données en paramètres.
Tester que `suite(0,4,5)` donne 4, `suite(1,4,5)` donne 5 et enfin que `suite(3,4,5)` donne 99.
Vous pouvez tester d'autres valeurs...

Exercice 4 - Entiers consécutifs de i à k .

Écrire une fonction récursive `boucle(i,k)` qui affiche les entiers entre i et k inclus.
Par exemple, `boucle(0,3)` doit afficher 0 1 2 3.
Tester avec d'autres valeurs.

On rappelle que si a et b sont deux nombres entiers naturels non nuls,
alors $PGCD(a;b)$ est le **Plus Grand Commun Diviseur** de a et b.

Exercice 5 - PGCD de deux entiers .

On rappelle la propriété suivante :

Calcul du PGCD avec la méthode des différences

Si a et b sont deux nombres entiers tels que $a > b$, alors PGCD(a, b) = PGCD$(a - b, b)$.

1. Utiliser cette propriété pour proposer une méthode récursive du calcul du PGCD de deux nombres. Veiller à bien identifier les cas simples, et les appels récursifs.
2. Écrire une fonction récursive `pgcd_diff(a,b)` qui renvoie le PGCD de deux entiers en utilisant la propriété.

   Tester que `pgcd_diff(5,5)` donne 5, que `pgcd_diff(10,5)` et `pgcd_diff(5,10)` donne 5, et enfin que `pgcd_diff(18,99)` donne 9.

Exercice 6 - PGCD de deux entiers, la suite .

On rappelle la propriété suivante :

Calcul du PGCD avec la méthode de la division euclidienne

Si a et b sont deux nombres entiers tels que $a > b$, et r est le reste de la division euclidienne de a par b, alors PGCD(a, b) = PGCD(b, r).

1. Utiliser cette propriété pour proposer une méthode récursive du calcul du PGCD de deux nombres. Veiller à bien identifier les cas simples, et les appels récursifs.
2. Écrire une fonction récursive `pgcd_eucl(a,b)` qui renvoie le PGCD de deux entiers en utilisant la propriété.
   Pour la recherche du cas simple, on pourra utiliser la fait que si a est un multiple de b, alors $PGCD(a;b) = b$.
   Reprendre les tests de l'exercice précédent.

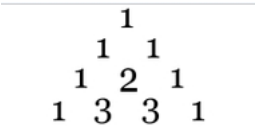
Exercice 7 - Les coefficients de binôme et le triangle de Pascal.

On sait que les coefficients du binôme sont faciles à visualiser avec le triangle de Pascal.

On rappelle que $\binom{n}{p}$ désigne le nombre de manières de choisir p éléments parmi n. Il existe une formule qui permet de calculer directement ce nombre de combinaisons, mais ici nous utiliserons, après l'avoir démontré en classe, que :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

On a rappelé ci-contre les 4 premières lignes du triangle de Pascal. La ligne 0 contient juste 1, la ligne 1 contient $(1;1)$, la ligne 3 contient $(1;2;1)$ ...



1. Représenter le début du triangle de Pascal (6 lignes, de la ligne 0 à la ligne 5), et associer à chaque nombre le coefficient binomial correspondant.

2. Donner, en utilisant la formule donnée, une définition récursive de $\binom{n}{p}$.

3. Programmer une fonction récursive `binome(n,p)` qui renvoie $\binom{n}{p}$. Tester votre fonction en vérifiant que `binome(5,0)` renvoie 1, que `binome(5,5)` renvoie 1 et que `binome(5,3)` renvoie 10. Vous pouvez procéder avec d'autres tests en vérifiant les valeurs écrites à la première question.

4. Écrire une fonction `pascal(n)`, qui à partir de la valeur de n entière choisie, écrit le **triangle de Pascal**, en utilisant la fonction `binome(n,p)`. Indication : on pourra utiliser une liste pour stocker les valeurs d'une ligne et les afficher...

   Pour aller plus loin, je vous suggère de lire la page suivante :

   `https://fr.wikipedia.org/wiki/Triangle_de_Pascal`

   On y apprend notamment qu'il y a un lien entre le triangle de Pascal et la suite de Fibonacci, alors qu'au départ rien ne permet de le deviner!

Exercice 8 - Calcul d'une somme maximale.

On considère un tableau de nombres de n lignes et p colonnes.
Les lignes sont numérotées de 0 à $n-1$ et les colonnes sont numérotées de 0 à $p-1$.
La case en haut à gauche est repérée par $(0,0)$ et la case en bas à droite par $(n-1,p-1)$.
On appelle **chemin** une succession de cases allant de la case $(0,0)$ à la case $(n-1,p-1)$, en n'autorisant que des déplacements case par case : soit vers la droite, soit vers le bas.
On appelle **somme d'un chemin** la somme des entiers situés sur ce chemin.
Par exemple, pour le tableau T suivant :

$T =$

4	**1**	**1**	3
2	0	**2**	1
3	1	**5**	**1**

- un chemin est $(0,0),(0,1),(0,2),(1,2),(2,2),(2,3)$ (en gras sur le tableau) ;
- la somme du chemin précédent est 14 ;
- $(0,0),(0,2),(2,2),(2,3)$ n'est pas un chemin.

L'objectif de cet exercice est de déterminer la somme maximale pour tous les chemins possibles allant de la case $(0,0)$ à la case $(n-1,p-1)$.

1. On considère tous les chemins allant de la case $(0,0)$ à la case $(2,3)$ du tableau T donné en exemple.
   (a) Un tel chemin comprend nécessairement 3 déplacements vers la droite. Combien de déplacements vers le bas comprend-il ?
   (b) La longueur d'un chemin est égal au nombre de cases de ce chemin. Justifier que tous les chemins allant de $(0,0)$ à $(2,3)$ ont une longueur égale à 6.
2. En listant tous les chemins possibles allant de $(0,0)$ à $(2,3)$ du tableau T, déterminer un chemin qui permet d'obtenir la somme maximale et la valeur de cette somme.
3. On veut créer le tableau T' où chaque élément $T'[i][j]$ est la somme maximale pour tous les chemins possibles allant de $(0,0)$ à (i,j).
   (a) Recopier et compléter sur votre copie le tableau T' donné ci-dessous associé au tableau T.

$T =$

4	1	1	3
2	0	2	1
3	1	5	1

et $T' =$

4	5	6	...
6	...	8	10
9	10	...	16

   (b) Justifier que si j est différent de 0, alors : $T'[0][j] = T[0][j] + T'[0][j-1]$.
4. Justifier que si i et j sont différents de 0, alors : $T'[i][j] = T[i][j] + max(T'[i-1][j], T'[i][j-1])$.
5. On veut créer la fonction récursive somme_max ayant pour paramètres un tableau T, un entier i et un entier j. Cette fonction renvoie la somme maximale pour tous les chemins possibles allant de la case $(0,0)$ à la case (i,j).
   (a) Quel est le cas de base, à savoir le cas qui est traité directement sans faire appel à la fonction somme_max ? Que renvoie-t-on dans ce cas ?
   (b) À l'aide de la question précédente, écrire en Python la fonction récursive somme_max .
   (c) Quel appel de fonction doit-on faire pour résoudre le problème initial ?

Exercice 9 - Retour sur la puissance $n^{ième}$ de x.

On rappelle la définition récursive de la fonction **puissance** vue dans le cours :

$$puissance(n,x) = \begin{cases} 1 & \text{si } n = 0 \\ x \times puissance(n-1,x) & \text{si } n > 0 \end{cases}$$

1. Programmer en Python la fonction puissance(n,x), puis vérifier que puissance(3,5) renvoie 125. (Normalement cette question est déjà traitée en cours !)
2. On remarque que cette formulation peut être simplifiée. En effet on sait que si $n = 1$, alors $x^1 = x$, on voit qu'il n'y a pas besoin de faire d'appel récursif pour cette valeur. **Écrire** la nouvelle fonction récursive, et **programmer** en Python la fonction puissance_bis(n,x) qui correspond. Tester cette nouvelle fonction.
3. On peut également définir ici une fonction avec plusieurs cas récursifs. Par exemple ici, on peut distinguer dans le calcul de x^n le cas où n est pair, et le cas où n est impair. En effet :
   - si n est pair, alors $x^n = x^{\frac{n}{2}\times 2} = \left(x^{\frac{n}{2}}\right)^2$
   - si n est impair (et donc $n-1$ est pair), alors $x^n = x \times x^{n-1} = x \times \left(x^{\frac{n-1}{2}}\right)^2$.

   Ainsi, on peut avec la fonction carre(x) qui renvoie $x * x$ et la fonction puissance_ter(n,x) définie de manière récursive par :

$$puissance\_ter(n,x) = \begin{cases} 1 & \text{si } n = 0 \\ carre(puissance\_ter(n/2,x) & \text{si } n > 0 \text{ et } n \text{ est pair} \\ x \times carre(puissance\_ter((n-1)/2,x) & \text{si } n > 0 \text{ et } n \text{ est impair} \end{cases}$$

   ► Programmer la fonction carre(x) et la fonction puissance_ter(n,x), et tester avec puissance_ter(0,2), puissance_ter(1,2), puissance_ter(2,2), puissance_ter(3,2) et puissance_ter(4,2) qui donnent respectivement 1, 2, 4, 8 et 16.

   Cette manière de définir la fonction **puissance** est bien plus efficace que la première manière, car le nombre d'appels récursifs est bien plus petit.

Exercice 10 - Un exemple de récursion imbriquée.

Les occurrences de la fonction en cours de définition peuvent également être **imbriquées**, c'est à dire qu'on appelle l'image de l'image. Par exemple, la fonction $f_{91}(n)$ ci-dessous, que l'on doit à John McCarthy (informaticien et lauréat du prix Turing en 1971), est définie avec deux occurrences imbriquées, de la manière suivante :

$$f_{91}(n) = \begin{cases} n-10 & \text{si } n > 100 \\ f_{91}(f_{91}(n+11)) & \text{si } n \leq 100 \end{cases}$$

1. Déterminer $f_{91}(101)$, $f_{91}(102)$ et $f_{91}(103)$.
2. Déterminer $f_{91}(100)$.
3. Vérifier que $f_{91}(99) = 91$ à la main.
4. Toujours à la main, déterminer $f_{91}(98)$ et $f_{91}(97)$.
5. Programmer en Python la fonction f91, et tester cette fonction avec les valeurs calculées.
6. Afficher, pour des valeurs de n allant de 0 à 110 les valeurs de $f_{91}(n)$. Quelle conjecture peut-on faire ?

Exercice 11 - Un exemple de récursion mutuelle.

Il est possible, et parfois nécessaire, de définir plusieurs fonctions récursives **en même temps**, quand ces fonctions font référence les unes aux autres. On parle alors de définitions **récursives mutuelles**. Par exemple, les fonctions $a(n)$ et $b(n)$ ci-dessous, inventées par Douglas Hofstadter (il y fait référence dans son ouvrage Gödel, Escher, Bach : Les Brins d'une pour lequel il a obtenu le prix Pulitzer en 1980) sont définies par récursion mutuelle de la manière suivante :

$$a(n) = \begin{cases} 1 & \text{si } n = 0 \\ n - b(a(n-1)) & \text{si } n > 0 \end{cases}$$

$$b(n) = \begin{cases} 0 & \text{si } n = 0 \\ n - a(b(n-1)) & \text{si } n > 0 \end{cases}$$

1. Calculer à la main $a(1)$, $b(1)$, $a(2)$ et $b(2)$.
2. Programmer en Python les fonctions `a(n)` et `b(n)` , et vérifier que `a(7) = 5` et `b(7) = 4`.
3. Écrire un algorithme qui affine les valeurs de $a(n)$ et $b(n)$ pour les valeurs de n de 0 à 20.
4. Faire une conjecture sur les valeurs obtenues pour $a(n)$ et $b(n)$.

Exercice 12 - Le nombre de chiffres d'un entier.

Écrire une fonction récursive `nombre_de_chiffres(n)` qui, à partir d'un entier n positif ou nul, renvoie son nombre de chiffres.
Par exemple, `nombre_de_chiffres(34127)` doit renvoyer 5.

Exercice 13 - Le nombre de bits d'un entier.

En s'inspirant de l'exercice précédent, écrire une fonction récursive `nombre_de_bits_1(n)` qui, à partir d'un entier n positif ou nul, renvoie son nombre de chiffres.
Par exemple, `nombre_de_bits_1(255)` doit renvoyer 8 et `nombre_de_bits_1(254)` doit renvoyer 7
Remarque : cette fonction est très utile dans de nombreux algorithmes bas niveau et porte le nom de **popcount** dans la littérature (abréviation de *population count*) . Elle est souvent implémentée par les unités arithmétiques et logiques des processeurs sur des entiers de taille fixe (32 ou 64 bits).

Pour les exercices suivants, on utilisera pour la réalisation des graphiques le module **Turtle**, dont un mémento est donné en dernière page.

Exercice 14 - Les Flocons de Koch.

Voici le principe de la courbe de Koch pour $n = 0$, $n = 1$ et $n = 2$ avec pour longueur $d = 200$ pixels.

1. Analyser le passage d'une figure à l'autre.
2. Donner une définition récursive de la construction.
3. Programmer une fonction récursive koch(n,d) qui permet de tracer le diagramme de Koch à l'ordre n avec pour longueur d, et ce à l'aide du module **Turtle**.

   Voici une aide, à consulter seulement si après vous être acharner sans retenue, l'inspiration ne vous est pas venue. Cette aide pourra être utilisée dans les prochains exercices...

```
from turtle import *

def koch(n,d):
  if n == 0 :
    forward(d)
  else :
    koch(n-1,d/3)
    left(60)
    koch(n-1,d/3)
    right(120)
    koch(n-1,d/3)
    left(60)
    koch(n-1,d/3)

up()
goto(-200,0)
down()
koch(4,400)
```

Ci-dessous le lien vers la page des Flocons de Koch :

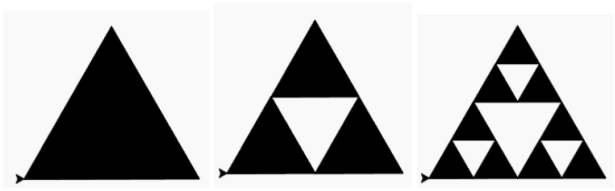
`https://fr.wikipedia.org/wiki/Flocon_de_Koch`

Approfondissements (à faire après avoir fini les autres exercices)

4. Proposer un code pour la réalisation du Flocon de Koch (voir la page WEB).
5. Parmi les variantes proposées sur la page WEB indiquée, proposer un code pour réaliser l'une d'elles, en 2D et/ou en 3D.

Exercice 15 - Les Triangles de Sierpiński.

Voici le principe des Triangles de Sierpinski pour $n = 0$, $n = 1$ et $n = 2$ avec pour longueur $d = 200$ pixels.



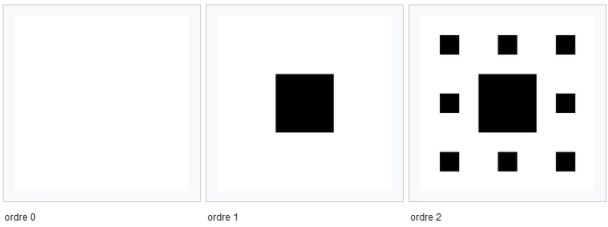
Pour en savoir plus sur cette figure, rendez-vous sur le site :

`https://fr.wikipedia.org/wiki/Triangle_de_Sierpi%C5%84ski`

1. Analyser le passage d'une figure à l'autre.
2. Donner une définition récursive de la construction.
3. Programmer en Python une fonction récursive `triangles(n,d)` qui permet de tracer le diagramme de Sierpinski à l'ordre n avec pour longueur d, et ce à l'aide du module **Turtle**.

Exercice 16 - Le Tapis de Sierpiński.

Voici le principe du Tapis de Sierpinski pour $n = 0$, $n = 1$ et $n = 2$.



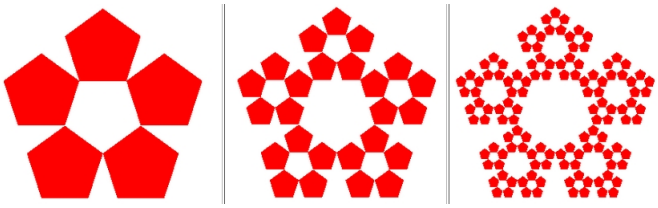
Pour en savoir plus sur cette figure, rendez-vous sur le site :

`https://fr.wikipedia.org/wiki/Tapis_de_Sierpi%C5%84ski`

1. Analyser le passage d'une figure à l'autre.
2. Donner une définition récursive de la construction.
3. Programmer en Python une fonction récursive `tapis(n,d)` qui permet de tracer le Tapis de Sierpinski à l'ordre n avec pour longueur d en pixels, et ce à l'aide du module **Turtle** (voir une aide sur ce module plus loin).

Exercice 17 - Les fractales de Sierpiński.

Dans les exercices précédents, nous avons découvert la réalisation de figures fractales (avec un procédé de récursion) construites avec des triangles et des carrés. Ce mécanisme est généralisable, comme avec la figure ci-dessous avec des pentagones :



Pour en savoir plus sur ce genre de figures, rendez-vous sur le site :

`https://mathcurve.com/fractals/sierpinski/sierpinski.shtml`

1. Analyser le passage d'une figure à l'autre.
2. Donner une définition récursive de la construction.
3. Programmer en Python une fonction récursive `pentagones(n,d)` qui permet de tracer les pentagones de Sierpinski à l'ordre n avec pour longueur d en pixels, et ce à l'aide du module **Turtle** (voir une aide sur ce module plus loin).
4. **Approfondissements** (à faire après avoir fini les autres exercices) Parmi les autres figures proposées sur la page WEB, en choisir une, en 2D ou en 3D, et proposer un algorithme en Python pour la réaliser.

Le module Turtle

La bibliothèque **Turtle** permet de réaliser des figures géométriques de la même manière qu'avec **Scratch**, mais au lieu de manipuler des blocs, on écrit un code Python qui d'ailleurs s'organise aussi en blocs...

On utilise les fonctions de ce module dans un programme après un : from turtle import *
Pour vérifier que le module est installé sur votre système, saisir puis exécuter le programme suivant, qui normalement doit afficher le résultat sur la droite.

```
from turtle import *

forward(100)
left(120)
forward(100)
left(120)
forward(100)
```

Il est possible d'utiliser le module Turtle en ligne, par exemple à l'adresse :

`http://fe.fil.univ-lille1.fr/apl/2018/turtle.html`

Liste des commandes les plus utiles avec Turtle

instruction	description
goto(x, y)	aller au point de coordonnées (x, y)
forward(d)	avancer de la distance d
backward(d)	reculer de la distance d
left(a)	pivoter à gauche de l'angle a
right(a)	pivoter à droite de l'angle a
setheading(a)	fixer la direction avec l'angle a
circle(r , a)	tracer un cercle d'angle a et de rayon r
dot(r)	tracer un point de rayon r
up()	relever le crayon (et interrompre le dessin)
down()	redescendre le crayon (et interrompre le dessin)
width(e)	fixe à e l'épaisseur du trait
color(c)	sélectionner la couleur c pour les traits
begin_fill()	activer le mode remplissage
end_fill()	désactiver le mode remplissage
fillcolor(c)	sélectionner la couleur c pour le remplissage
write('texte')	écrit un texte
clear()	efface tout ce qui est tracé
reset()	tout effacer et recommencer
speed(s)	définir la vitesse de déplacement
setup(L, l)	définit la longueur et la hauteur de la fenêtre.

Au départ, la tortue est en $(0,0)$, orientée à 0^o, c'est à dire vers la droite. La fenêtre par défaut fait $950 = 2 \times 475$ pixels de large, et $800 = 2 \times 400$ pixels de haut. Le point $(0,0)$ est au centre de l'écran. Il est possible de modifier cette fenêtre avec **setup** et **clear**.

Les couleurs sont notamment "black", "blue" , "red", "green" ... ou bien en RGB comme (0.3,0.3,0.3) pour un gris foncé.