Récursivité

En mathématiques, vous êtes nombreux à avoir vu les suites en spécialité de 1ère. Une suite définie par récurrence simple s'écrit sous la forme $u_{n+1} = f(u_n)$.
Si on "descend" d'un rang, on obtient $u_{n+1} = f(f(u_{n-1}))$, et plus généralement
$u_{n+1} = \underbrace{f(f(f(\dots f(u_0))))}_{n \text{ fois.}}$.

En informatique, la récurisivité se rapproche de ce type de raisonnement. Une fonction récursive est une fonction qui s'appelle elle-même.

Un premier exemple

La fonction factorielle est la fonction notée $!$. Vous avez peut-être déjà vu cette fonction en mathématiques. Elle s'applique sur les entiers naturels et vaut :

$$\begin{cases} 0! = 1 \\ n! = n \times n-1 \times n-2 \times \cdots \times 1 \end{cases}$$

On lit "factorielle n" de préférence.

Voici un des multiples algorithme de calcul de la factorielle en itératif.

Algorihtme : factorielle itérative
Donnée : un entier naturel n
Renvoie : $n!$
Début

```
f ← 1
si n ≠ 0
    i ← 1
    Tant que i < n faire
        i ← i + 1
        f ← f × i
    Fin Tant que
Renvoyer f
```

fin

Ci-dessous le code correspondant. On a rajouté une ligne de code permettant de mesurer le temps d'exécution de la fonction, calculé en moyenne sur un grand nombre de fois (attention cela prend quelques secondes supplémentaires)

```
%timeit (fact(6))
```

Dans l'affichage du temps, "mean" signifie moyenne et std. dev. signifie "écart-type"

In [ ]:
```
def fact(n) :
    """
    Calcul la factorielle de n
    @param n : entier positif ou nul
    @return f : entier strictement positif, égal à n!
    """
    f = 1
    if n != 0 :
        i = 1
        while i < n :
            i = i + 1
            f = f * i
    return f

print(fact(6))

%timeit (fact(6))
```

Comme vous pouvez le constater avec quelques tests, la fonction factorielle est une fonction qui croît très rapidement, plus encore que l'exponentielle. Pour l'exemple, si un programme de complexité $O(n!)$ met un temps de $1,2\mu s$ pour $n = 5$ alors il mettra un temps 10^{48} ans pour $n = 50$, contre une centaine de jours pour un programme de complexité exponentielle.
Voyons ci-dessous un algorithme de calcul de la factorielle en version récursive.

Algorihtme : factorielle récursive
Donnée : un entier naturel n
Renvoie : $n!$
Début

fonction fact(*n*)

si *n* = 0

renvoyer 1

sinon

renvoyer *n* × fact(*n* - 1)

fin si

fin

Le code est ci-dessous. Vous pouvez comparer l'efficacité des deux fonctions avec `timeit`

In [ ]:
```
def fact(n) :
    if n == 0:
        return 1
    else :
        return n * fact(n - 1)

assert( fact(6) == 720)
assert(fact(10) == 3628800)
assert(fact(0) == 1)
assert(fact(1) == 1)
print(fact(6))

%timeit (fact(6))
```

Copiez ce code (sans les `assert`), et visualisez-en l'exécution sur Python Tutor (http://pythontutor.com/visualize.html#mode=edit). Vous y verrez notamment l'évolution de la "pile d'appels".

Exercices

Tout doit bien sûr être programmé en récursif ! Utilisez Spyder de préférence, ou un autre IDE.

1. Calculer x^n pour n entier positif (attention il y a un petit piège)
2. Exponentiation rapide : calculer x^n pour n entier strictement positif avec la méthode d'exponentiation rapide :

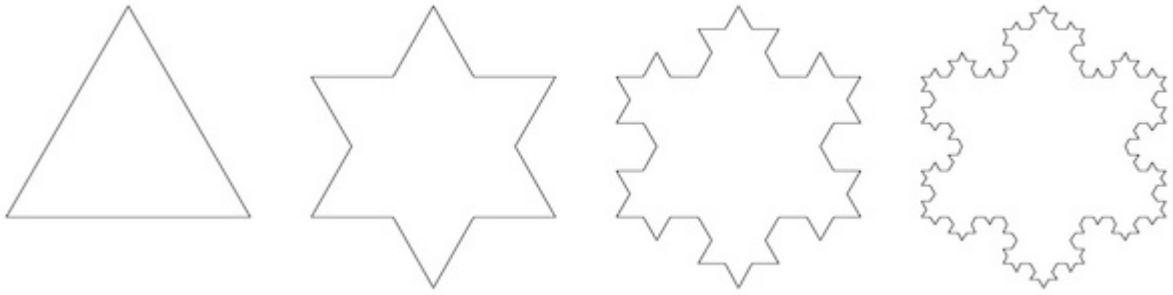
$$\text{puissance}(x, n) = \begin{cases} x & \text{si } n = 1 \\ \text{puissance}(x^2, n/2) & \text{si } n \text{ est pair} \\ x \times \text{puissance}(x^2, (n-1)/2) & \text{si } n \text{ est impair} \end{cases}$$

1. Calculer le nombre de chiffres nécessaires à l'écriture d'un nombre en base 2. Pour rappel : c'est comme celà qu'on a commencé à définir la fonction $x \to \log_2(x)$.
2. Facultatif : calculer le pgcd de deux entiers positifs grâce à l'algorithme d'Euclide (dite également méthode des divisions euclidiennes successives).
   Exemple : pgcd de 221 et 782
   $$221 = 782 \times 0 + 221$$
   on décale le 782 avant le égal et on divise par 221
   $$782 = 221 \times 3 + 119$$
   on décale le 221 avant le égal et on divise par 119
   $$221 = 119 \times 1 + 102$$
   on décale le 119 avant le égal et on divise par 112
   $$221 = 119 \times 1 + 102$$
   on décale le 119 avant le égal et on divise par 112
   $$221 = 119 \times 1 + 102$$
   $$119 = 102 \times 1 + 17$$
   on décale le 102 avant le égal et on divise par 17
   $$221 = 119 \times 1 + 102$$
   $$102 = 17 \times 6 + 0$$
   Le pgcd de 221 et 783 est égal au dernier reste non nul, soit 17.
3. Ecrire une fonction qui détermine si une chaîne de caratères est un palindrôme. un palindrôme est un mot ou une phrase qui peut se lire dans les deux sens, comme "radar" ou "élu par cette crapule" (on écrira les chaines sans caractères diacritiques).
4. Un peu plus dur : écrire une fonction qui, étant donné une chaîne de caractères, renvoie lune chaîne donnant la chaîne d'entrée écrite à l'envers . *Exemple* : "et voilà !" donnera "! àliov te".
5. Un exercice rigolo Peut-être certains d'entre vous ont déjà utilisé le module `turtle`de Python. Ce module permet de programmer une "tortue" qui dessine. Les instructions principales sont :

   - `forward(nombre)` : avancer de nombre (en pixels)
   - `backward(nombre)` : avancer de nombre
   - `left(angle)`: tourner de angle vers la gauche

- `right(angle)` : tourner de angle vers la droite
- `clear()` : effacer le dessin
- `goto(x,y)` : aller au point de coordonnées spécifiées
- up() : lever le crayon (pour se déplacer sans dessiner)
- down() : baisser le crayon (pour dessiner à nouveau)

On va travailler sur le flocon de Von Koch, qui est une structure fractale (https://fr.wikipedia.org/wiki/Fractale). Partant d'un triangle équilatéral, on découpe chaque segment en trois, on enlève le tiers médian que l'on remplace par les deux côtés de même taille d'un triangle équilatéral vers l'extérieur. Et on itère jusqu'à $+\infty$. Comme vous le voyez, la construction récursive est apparente. La figure ci-dessous donne les quatre premières étapes de la construction.



() Le but de cet exercice est de tracer une approximation du flocon de Von Koch, et de conjecturer ses propriétés sur le périmètre et la surface.

Un morceau de code pour vous lancer :

```
import turtle

def FractaleKoch(n, longueur) :
    if n==0 :
        turtle.forward(longueur)
    else :
        FractaleKoch(n-1, longueur/3)
        turtle.left(60)
        FractaleKoch(n-1, longueur/3)
        turtle.right(120)
        FractaleKoch(n-1, longueur/3)
        turtle.left(60)
        FractaleKoch(n-1, longueur/3)

longueur_segment_initial=400
turtle.up()
turtle.goto(longueur_segment_initial/40 - 500, longueur_segment_init
ial/100)
turtle.down()
FractaleKoch(3, longueur_segment_initial)
turtle.up()

turtle.ht()
turtle.exitonclick()
```

Recopier et compléter ce code de manière à tracer le flocon pour un rang donné, et donner son aire et son périmètre.
Conjecturer la valeur de l'aire et du périmètre en +∞. Plus mathématiquement : que valent $\lim\limits_{n\to+\infty}(\text{aire}_n)$ et $\lim\limits_{n\to+\infty}(\text{périmètre}_n)$?
Remarque : prouver ces conjectures est un exercice intéressant de spécialité mathématiques sur les suites, assez facile.

Deux problèmes

Pour ceux qui vont vite.

1. Cette énigme a été proposée dans un grand quotidien du soir.
   On dispose d'un paquet de n cartes numérotées de 0 à *n* ($n \geq 0$), la carte 0 étant au-dessus du paquet, la carte *n* au-dessous du paquet. On prend la carte supérieure et on la replace au-dessous du paquet. On prend la suivante et on la jette. Et ainsi de suite.
   Quelle sera la dernière carte survivante ?
   Écrire une fonction qui retourne la valeur de la carte survivante. Donner la liste des cartes survivantes pour tous les paquets de taille inférieure ou égale à 128 cartes. Que conjecturez-vous?
   Un cadeau pour ceux qui font spécialité mathématiques : démontrer le résultat conjecturé.
2. Théorème de Lagrange
   Tout entier positif possède au moins une décomposition en la somme d'au plus 4 carrés parfaits.
   Exemple : 34 = 9 + 25 = 9 + 9 + 16 = 1 + 1 + 16 + 16 = 1 + 4 + 4 + 25
   Écrire une fonction qui, étant donné un entier positif *n*, donne toutes ses décompositions

D'autres types de récursivité

Récursivité croisée

On définit deux suites (un) et (vn) au moyen des équations suivantes :

$$\begin{cases} u_0 = 1 \\ v_0 = 1 \\ u_{n+1} = 3u_n + 2v_n \quad \text{si } n \geq 0 \\ v_{n+1} = 2u_n + 3v_n \quad \text{si } n \geq 0 \end{cases}$$

Écrire, pour chacune des deux suites, une fonction qui, étant donné un entier naturel n donné, calcule son terme d'indice n.

Récursivité imbriquée

La fonction f_{91} est définie pour tout entier naturel par :

$$f(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ f\,(f(n + 11)) & \text{sinon} \end{cases}$$

1. Calculer à la main $f_{91}(99)$
2. Ecrire une fonction récursive permettant de calculer la valeur de $f_{91}(n)$ pour n donné.
3. Calculer et stocker dans une liste toutes les valeurs de $f_{91}(n)$ pour n de 0 à 200. Que constatez vous ?

In [ ]:
```
def f91(n: int) -> int:
    """
    fonction f91 de McCarthy
    """


l_res = []
for i in range (200):
    l_res.append(f91(i))

print(l_res)
```

Que pensez-vous de la terminaison du calcul de $f_{91}(n)$?

Exercices

Quelles réflexions vous inspirent les fonctions récursives suivantes ? Je vous conseille d'éditer la cellule (par un double clic) afin de noter vos réponses précises, pour pourvoir réviser.

1. $ f(n) = \left{

```
\begin{array}{c c}
  1 & \textrm{si } n = 0 \\
  n \times f(n + 1) & \textrm{sinon }\\
\end{array}
```

   \right.
   $
   Réponse :
2. $ f(n) = \left{

```
\begin{array}{c c}
  1 & \textrm{si } n = 0 \\
  n \times f(n - 2) & \textrm{sinon }\\
\end{array}
```

   \right.
   $
   Réponse :
3. $ f(n) = \left{

```
\begin{array}{c c}
  1 & \textrm{si } n = 0 \\
  n \times f(n - 1) & \textrm{si } n \gt 1\\
\end{array}
```

   \right.
   $
   Réponse :

Des limites de la récursivité

On veut calculer un terme de rang donné de la suite de Fibonacci, dont la définition par récurrence sur $\mathbb{N}$ est :

$$\begin{cases} u_0 = 0 \\ u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \end{cases}$$

Ecrire le programme récursif donnant la valeur de rang donné d'un élément de la suite de Fibonacci. Testez-le en calculant quelques valeurs, de plus en plus grandes.

In [ ]:
```
def fiboRec(rang):



def fiboRecC(rang , compteur = 0):


n = 30
print(fiboRec(n))
```

Comme vous le constatez peut-être, ce programme est lent... Vous pouvez visulaiser la pile d'appels avec Python Tutor (http://pythontutor.com/visualize.html#mode=edit).

Rajoutez un compteur pour compter le nombre d'appels de la fonction (soyons fous, une variable globale est autorisée, mais sans c'est mieux). Placez quelques points $(n; y)$ dans un repère, où n est le rang du terme calculé de la suite de Fibonacci, y est le nombre d'appel de la fonction. Conjecturer ensuite la nature de la complexité de l'algorithme récursif de Fibonacci (constante, logarithmique, linéaire, log-linéaire, quadratique, exponentielle, factorielle).
Vous pouvez en plus utiliser l'instruction `%timeit (fibo(n))`, pour avoir les temps d'exécution.

Ecrivez ensuite un programme plus simple, de complexité spatiale minimale, et plus rapide, pour calculer un terme donné de la suite de Fibonnacci. Donner la complexité temporelle du programme construit.

Merci à Mmes. et Ms. Babolonski, Conchon, Filiâtre, Nguyen, Theilaud pour leurs précieuses idées.

Frederic Mandon (mailto:frederic.mandon@ac-montpellier.fr), Lycée Jean Jaurès - Saint Clément de Rivière - France (2020)