Récursivité

Théorie et pratique

Walter Appel

Plan

Calcul de factorielle

$$n! = \prod_{i=1}^{n} i \qquad \textit{itératif}$$

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot (n-1)! & \text{sinon .} \end{cases} \qquad \textit{récursif}$$

Codage itératif

$$n! = \prod_{i=1}^{n} i \qquad \textit{itératif}$$

```
# Définition itérative
def factorielle(n):
    p = 1
    for i in range(1, n + 1):
        p *= i
    return p
```

Codage récursif

« Pour calculer 5!, je demande à mon voisin le résultat de 4!, je multiplie par 5 et j'annonce le résultat. »

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot (n-1)! & \text{sinon .} \end{cases} \qquad \textit{récursif}$$

```
# Définition récursive simple
def facto(n):
    if n <= 1:
        return 1
    else:
        return n * facto(n - 1)
```


Les trois lois de la récursivité

Loi 1

Une fonction récursive gère des cas de base

Loi 2

Une fonction récursive s'appelle elle-même dans un cas différent

Loi 3

La suite des appels doit nécessairement aboutir à un cas de base

Descente infinie

- Il ne faut surtout pas oublier la gestion du cas de base sous peine de provoquer une **descente infinie !** C'est une erreur fréquente...
- Dans la vraie vie, il n'y a évidemment jamais de véritable descente infinie. La pile de récursion a une taille finie. Lorsqu'elle déborde, on obtient le message suivant :

```
RuntimeError: maximum recursion depth exceeded
in comparison
```

- On peut modifier la taille de la pile de récursion (de taille ≈ 1000 par défaut) :

```
import sys
sys.setrecursionlimit(100000)
```


Modifions légèrement notre programme :

```
def facto(n):
    if n <= 1: return 1
    else :
        print ('==>'*n+'> appel de facto({})'.format(n-1))
        m = n*facto(n-1)
        print ('<--'*n+'> sortie de facto({})'.format(n-1))
        return m
```

```
>> facto(5)
==========> appel de facto(4)
========> appel de facto(3)
======> appel de facto(2)
====> appel de facto(1)
---> sortie de facto(1)
----> sortie de facto(2)
-----> sortie de facto(3)
-------> sortie de facto(4)
120
```

$$\begin{aligned} 5! &= \\ &= 5 \times 4! \\ &= 5 \times (4 \times 3!) \\ &= 5 \times (4 \times (3 \times 2!)) \\ &= 5 \times (4 \times (3 \times (2 \times 1!))) \\ &= 5 \times (4 \times (3 \times (2 \times 1))) \\ &= 5 \times (4 \times (3 \times 2)) \\ &= 5 \times (4 \times 6) \\ &= 5 \times 24 \\ &= 120 \end{aligned}$$

Exercices

Pour calculer x^n, on peut remarquer que

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

- Écrire un algorithme, puis un programme en Python, permettant de calculer récursivement x^n.
- Écrire un algorithme *non récursif* qui calcule x^n selon la même méthode !

There are more things...

- Une fonction récursive n'a pas forcément un argument numérique.

```
def reverse(s):
    if s == "":
        return s
    else:
        return reverse(s[1:]) + s[0]
```

```
reverse("truc")
  = reverse("ruc")+"t"
  = [reverse("uc")+"r"]+"t"
  = [{reverse("c")+"u"}+"r"]+"t"
  = [{reverse("")+"c"+"u"}+"r"]+"t"      # attention ici !
  = [{""+"c"+"u"}+"r"]+"t"
  = "curt"
```

- De plus, elle peut s'appeler plusieurs fois !

PGCD de deux nombres

```
# PGCD de deux entiers.
def pgcd(a, b)
    if b == 0:
        return a
    else:
        return pgcd(b, a % b)
```

Cette manière de programmer retranscrit très fidèlement l'algorithme d'Euclide, et est particulièrement simple à écrire !

- **Terminaison** : à chaque appel, on a

  $$0 \leqslant a\%b < b$$

  et arrêt lorsque $b = 0$. ☑
- **Validité** : on a transcrit l'algorithme d'Euclide en renvoyant le dernier reste non nul. ☑

PGCD d'une liste

On utilise le fait que

$$\text{PGCD}(\ell_0, \ell_1, \ldots, \ell_n) = \text{PGCD}\Big(\ell_0, \big(\text{PGCD}(\ell_1, \ldots, \ell_n)\big)\Big).$$

```
# PGCD d'une liste L de nombres entiers
def PGCD(L)
    if len(L) == 1 :
        return L[0]
    else :
        return pgcd(L[0], PGCD(L[1:]))
```

Ici, les arguments sont des objets, et la récursivité se fait avec une taille décroissante d'objets.

Une maladresse

L'algorithme de Syracuse part d'un entier $u_0 \geqslant 1$ et définit une suite $(u_n)_{n\geqslant 0}$ par la relation de récurrence

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

```
def syracuse(u0, k):          # Calcule le k-ième terme
    if k == 0:
        return u0
    else:

        if syracuse(u0, k-1) % 2 == 0:
            return syracuse(u0, k-1) // 2
        else:
            return syracuse(u0, k-1) * 3 + 1
```


Une maladresse

L'algorithme de Syracuse part d'un entier $u_0 \geqslant 1$ et définit une suite $(u_n)_{n\geqslant 0}$ par la relation de récurrence

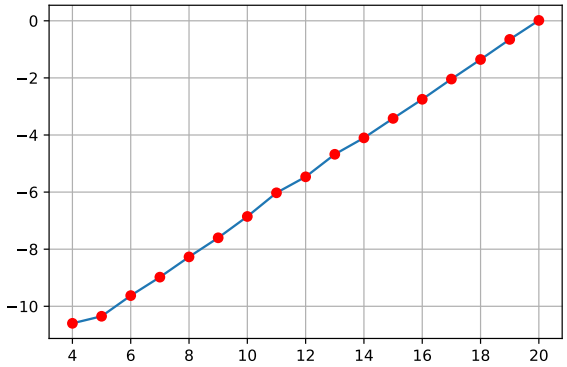
$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

```
def syracuse(u0, k):          # Calcule le k-ième terme
    if k == 0:
        return u0
    else:
        a = syracuse(u0, k-1)
        if a % 2 == 0:
            return a // 2
        else:
            return a * 3 + 1
```

Calcule `syracuse(1,20)` en $3 \cdot 10^{-5}$ s au lieu d'1 s !

Évaluer la complexité

On trace, en fonction de l'indice n de la suite, le *logarithme* du temps nécessaire à calculer u_n :



La pente est $\alpha \approx \frac{2}{3} \approx \ln 2$, donc on conjecture

$$C(n) = \Theta(e^{\alpha n}) = \Theta(2^n).$$

Le problème de la terminaison

Toute suite de Syracuse est réputée finir par « boucler »

$$1 \to 4 \to 2 \to 1 \to 4 \to 2 \to 1 \to 4 \to 2 \to 1 \to \cdots$$

```
def suivant(n):
    if n <= 1 :
        return 1
    else :
        if n % 2 == 0 :
            return n // 2
        else :
            return 3 * n + 1

def cte(n):
    if n == 1 :
        return 1
    else :
        return cte(suivant(n))
```

Définition formelle

On a besoin d'un ensemble bien ordonné (toute partie non vide admet un plus petit élément) E, n'admettant pas de suite décroissante infinie ; (typiquement, $\mathbb{N}$) et d'un sous-ensemble $\mathcal{B}$ de E, appelé **ensemble des cas de base**.

Une fonction f est **récursive** si elle s'écrit sous la forme

$$f(x) = \begin{cases} \varphi(x) & \text{si } x \in \mathcal{B} \\ \Phi\Big(x, f\big(\alpha_1(x)\big), \ldots, f\big(\alpha_p(x)\big)\Big) & \text{sinon} \end{cases}$$

Théorème

Si les $\alpha_k(x)$ sont « strictement plus petits » que x, alors l'appel récursif se termine nécessairement.

Exemple de la factorielle

Une fonction F est récursive si elle s'écrit sous la forme

$$f(x) = \begin{cases} \varphi(x) & \text{si } x \in \mathcal{B} \\ \Phi\Big(x, f\big(\alpha_1(x)\big), \ldots, f\big(\alpha_p(x)\big)\Big) & \text{sinon} \end{cases}$$

$$f(n) = 1 \qquad \text{si } n \in \{0\}.$$

$$f(n) = \underbrace{n \cdot f\big(\underbrace{n-1}_{\alpha_1(n)}\big)}_{\Phi\left(n,\, f(\alpha_1(n))\right)}$$

- Lorsque l'on travaille dans $\mathbb{N}$, on a classiquement $\alpha(n)$ de la forme $n-1$, $\lfloor n/2 \rfloor$,...
- Si x est un tableau, $\alpha(x)$ pourra être une partie (stricte !) du tableau.

Calculer 36 fois la même chose

Définissons

$$u_n = \begin{cases} 1 & \text{si } n = 0 \\ n\,u_{n-1} + u_{n-1}^2 & \text{sinon .} \end{cases}$$

- Écrire un programme itératif calculant u_n.
- Écrire un programme *récursif* calculant u_n.
- Quelles sont leurs complexités ?

Calculer 36 fois la même chose

Définissons la suite de Fibonacci par

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

Calculer 36 fois la même chose

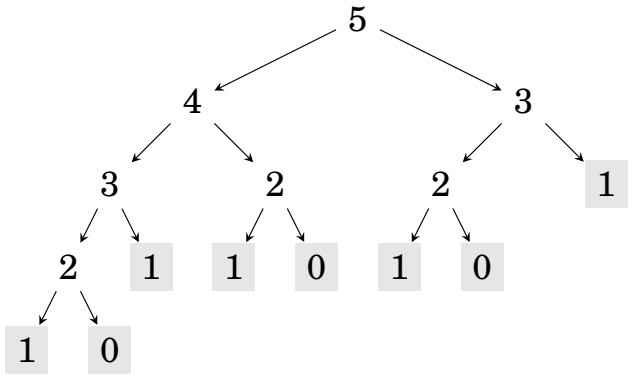
```
def fibo(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibo(n-1) + fibo(n-2)
```

- `fibo(5)` appelle `fibo(4)` et `fibo(3)`.
  mais...
- `fibo(4)` appelle `fibo(3)` et `fibo(2)`
  or...
- `fibo(3)` appelle `fibo(2)` et `fibo(1)`
- etc.

... et l'on peut noter que `fibo(3)` va être appelé deux fois, avec tout ce qu'il s'ensuit !

Calculer 36 fois la même chose

Sous forme plus schématique :



Le problème du temps

- Les calculs intermédiaires sont stockés dans une pile.
- La pile peut grossir de manière intempestive, comme la profondeur de l'arbre de récursion, mais surtout passe son temps à être *empilée et dépilée*. Le nombre d'appels varie
  - comme n dans l'algorithme calculant $n!$
  - « comme » $e^{\alpha n}$ dans Fibonacci (et plus précisément en $\Theta(\rho^n)$, où
    $$\rho = \frac{1+\sqrt{5}}{2} \approx 1,618...$$
    est le nombre d'or)...
    $$\rho^{50} \approx 28 \cdot 10^9 \qquad \rho^{100} \approx 8 \cdot 10^{20}$$
  - ... voire pire.

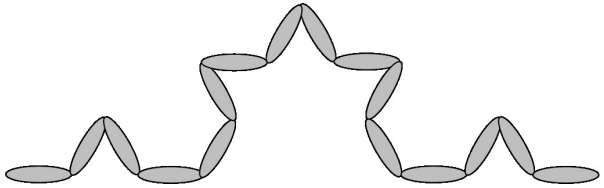
Exemples dans $\mathbb{N}$

- Calcul de $n!$, de a^n, de la suite de Fibonacci, de Syracuse...
- Nombres de Catalan : $C_0 := 1$ et $C_n := \sum_{k=0}^{n-1} C_k \, C_{n-1-k}$.
- Décomposition d'un entier en base 2 :
  $$42 = \underbrace{32}_{2^5} + \underbrace{8}_{2^3} + 2 = 101010.$$
- Dénombrement (nombre de surjections, coefficients du binôme...)

Flocon de von Koch

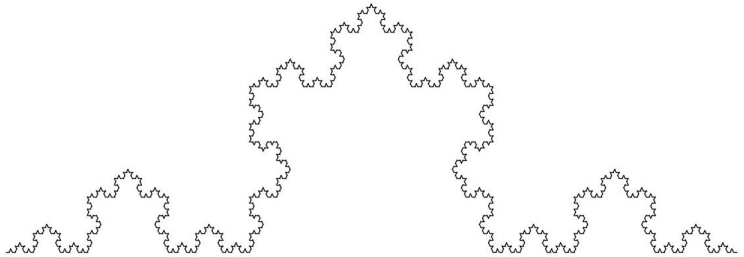
Une *branche de von Koch* est un objet constitué... de quatre branches de von Koch plus petites (1/3), disposées ainsi :

Par conséquent, il est aussi comme ceci :



Flocon de von Koch

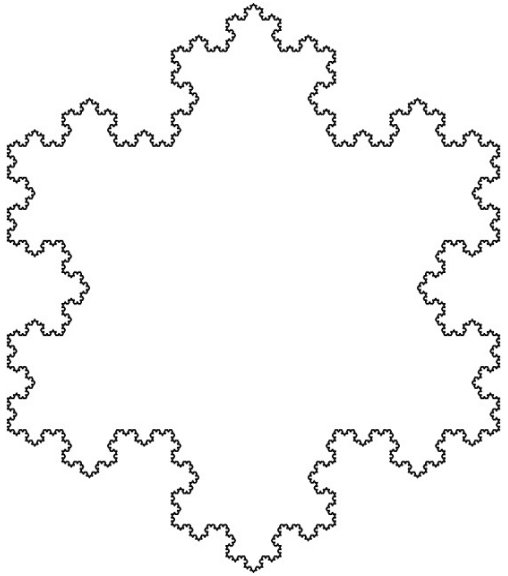
Mathématiquement, on a affaire à un **point fixe** (il existe, il est unique !) d'une certaine transformation...



Programmer von Koch

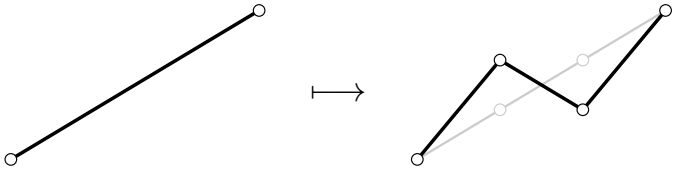
```
def UnPas(x,y,r,arg):
    return x+r*cos(arg) , y+r*sin(arg)

def VonKoch(n,x0,y0,r,theta):
    if n == 0:                                  # État de base !
        (xp,yp) = UnPas(x0,y0,r,theta)
        plot([x0,xp],[y0,yp],color='black')
    else:
        x,y = x0,y0
        VonKoch(n-1,x,y,r/3,theta)
        x,y = UnPas(x,y,r/3,theta)
        VonKoch(n-1,x,y,r/3,theta+pi/3)
        x,y = UnPas(x,y,r/3,theta+pi/3)
        VonKoch(n-1,x,y,r/3,theta-pi/3)
        x,y = UnPas(x,y,r/3,theta-pi/3)
        VonKoch(n-1,x,y,r/3,theta)
```


Flocon de von Koch



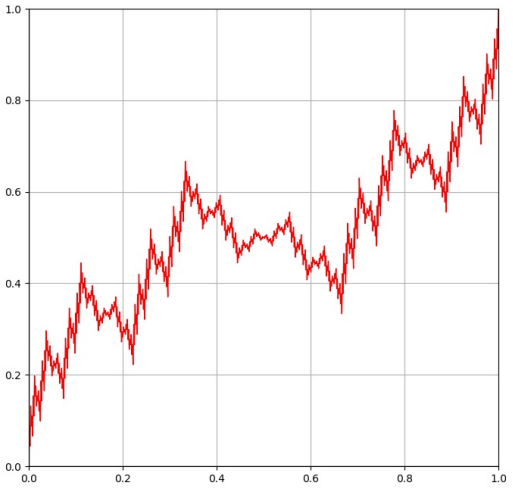
La courbe de Bolzano

Un des premiers exemples de fonctions continues sur $[0\,;1]$ mais dérivable en aucun point a été donné par Bolzano. Cette fonction f est la limite simple de la suite $(f_n)_{n\geqslant 0}$ dont les graphes sont obtenus itérativement en brisant chaque ligne selon le schéma suivant :



On obtient, en partant de la fonction identité sur $[0\,;1]$, la séquence ci-après

La courbe de Bolzano



Tri fusion

On sait fusionner deux listes en temps linéaire.

- Je séparer ma liste en deux...
- ... je demande à deux grouillots de me les trier...
- ... puis je les fusionne.

Chaque grouillot se cherche deux grouillots pour trier la moitié de sa propre liste.

Chaque étape demandera une fusion linéaire, et il y a environ $\log_2 n$ étapes.

La complexité du tri fusion est en $n \log_2 n$ ce qui est, asymptotiquement, optimal (comme déjà vu...).

Mémoïzation

L'ennui dans Fibonacci récursif-naïf, c'est qu'on calcule plein de fois la même chose ! L'idée est donc de *retenir* ces calculs intermédiaires, en les stockant par exemple dans un tableau.

```
L = [0,1] + [None]*200          # L est définie globalement

def fibmem(n):
    if L[n] == None:            # F_n n'a pas été calculé
        L[n] = fibmem(n-1) + fibmem(n-2)
    return L[n]                 # Dans tous les cas
```

Python n'arrive pas à calculer `fibo(50)` ... mais effectue `fibmem(100)` en une fraction de seconde.

Mémoïzation

Dans le même esprit, on peut utiliser une fonction récursive auxiliaire :

```
def Fibo(n)
    L = [0,1] + [None]*(n-1)
        # La taille est adaptée au problème
    def fifi(k)
        if L[k] == None:
            L[k] = fifi(k-1) + fifi(k-2)
        return L[k]
    return fifi(n)
```


Abaisser l'ordre de récursion

- Si l'on **vectorialise** le problème, cela transforme la suite récurrente d'ordre 2 en une suite récurrente d'ordre 1. L'arbre des appels récursifs n'explose plus : ce n'est plus qu'un tronc !
- On pose, pour tout $n \in \mathbb{N}^*$,

$$X_n = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

et

$$X_{n+1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} X_n.$$

Ce qui est bien

- C'est facile à écrire, c'est souvent la façon la plus *naturelle* (von Koch...).
- La *preuve de* ***correction*** est en général facile à rédiger.
- **On ne sait pas ce que fait l'ordinateur** derrière les instructions simples (baguette magique) !

Ce qui est mal

- C'est facile à écrire, c'est souvent la façon la plus *naturelle* (von Koch...). Explosion de la complexité en temps ; de même, la pile peut exploser !
- La *preuve de* ***correction*** est en général facile à rédiger. La *preuve de* ***terminaison*** est triviale... ou très difficile.
- **On ne sait pas ce que fait l'ordinateur** derrière les instructions simples (baguette magique) !

Passage par valeurs ou par adresse

On écrit une fonction récursive `appartient` testant l'appartenance, ou non, d'un objet x à une liste L.

```
def appartient(L, x):
    if len(L) == 0:
        return False
    else:
        return L[0] == x or appartient(L[1:], x)
```

Nota bene : L'algorithme se termine *aussitôt* que x est trouvé dans L, grâce à l'**évaluation paresseuse** de l'opérateur `or`.

```
>>> 1+1 == 2 or 1/0 = 42
True
```

Mais : **Explosion de la pile et du temps de calcul !!**
En effet, on empile à chaque étape une *nouvelle* liste, qui consomme du temps et de la mémoire.

Passage par valeurs ou par adresse

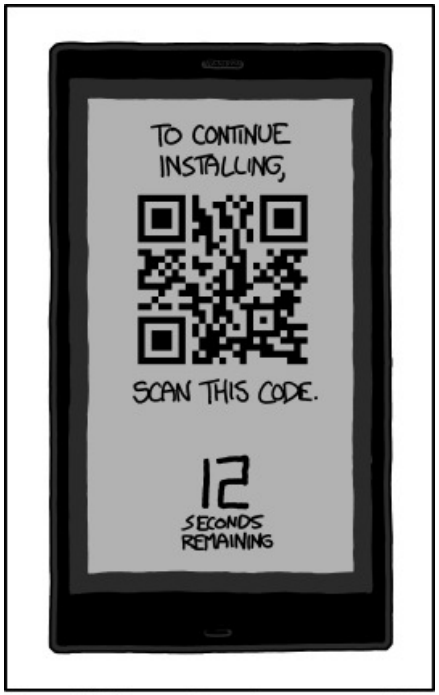
Au lieu de passer comme argument la **valeur** de la liste (`L[1:]` est un nouvel objet, obtenu par *slicing*), on peut se contenter de ne passer que l'**adresse** :

```
def appart2(L, x, i):
    n = len(L)
    if i == n:
        return False
    else:
        return x == L[i] or appart2(L, x, i+1)
```

(C'est Python qui se charge de comprendre que seule l'adresse de L compte...)

Passage par valeurs ou par adresse

```
import sys
from time import time
sys.setrecursionlimit(10**5)

L = [1]*(10**4)

t0 = time()
appartient(L,0)
t1 = time()
print(t1-t0)              # 1.398

t0 = time()
appart2(L,0,0)
t1 = time()
print(t1-t0)              # 0.0130
```

How to freak out a mobile app user