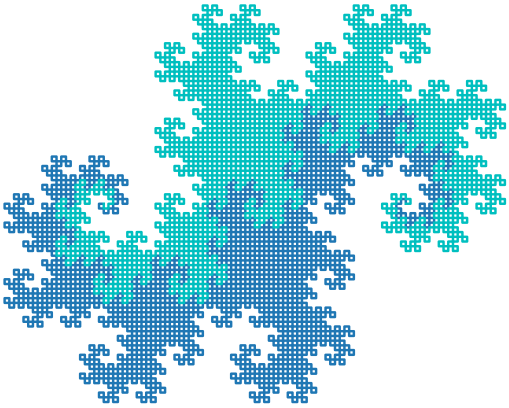
Introduction à la récursivité



Objectifs

A la fin de la séquence d'enseignement les élèves doivent :

- distinguer une boucle itérative d'une boucle récursive
- pouvoir traduire un algorithme récursif en langage **Python**
- pouvoir passer d'une boucle récursive simple à une boucle itérative et inversement
- connaitre les problèmes de complexité spatiale et complexité temporelle associées aux méthodes récursives

Table des matières

1 Schémas itératifs - schémas récursifs

1.1 Boucles conditionnelles et boucles inconditionnelles

Il existe des boucles dites conditionnelles, basées sur un test booléen. Tant que le test est vrai, la boucle se répète; dès que le test devient faux, on sort de la boucle. C'est le principe de la boucle itérative `while`.

La variable `res` est successivement multipliée par les valeurs `n`, `n-1`, ..., `2`.

```
def fac(n):
    res = 1
    while n > 1:
        res *= n
        n -= 1
    return res
```

```
def fac(n):
    res = 1
    for i in range(2, n+1):
        res *= i
    return res
```

Les boucles itératives inconditionnelles sont basées sur un itérable dont les valeurs seront successivement prise par le variant de boucle. C'est le principe de la boucle `for`.

La variable `res` est successivement multipliée par les valeurs `2`, `3`, ..., `n-1` puis `n`.

1.2 Boucles récursives

Il est des cas où le programme s'exprime plus clairement en ayant recours à une fonction qui lors de son exécution s'appelle elle-même. Il s'agit alors de boucles récursives.

`fac(n)` renverra `n*fac(n-1)` quand `fac(n-1)` sera calculé (quand `fac(n-2)` sera calculé...).

```
def fac(n):
    if n < 2:
        return 1
    else:
        return n * fac(n-1)
```

1.3 Quel type de boucle utiliser ?

Nous avons déjà vu qu'une boucle inconditionnelle `for` peut s'écrire comme une boucle conditionnelle `while`, par exemple avec un test sur le nombre d'éléments itérés.

```
def fac(n):
    res = 1
    for i in range(2, n+1):
        res *= i
    return res
```

```
def fac(n):
    res = 1
    i = 2
    while i <= n :
        res *= i
        i += 1
    return res
```

Aussi, certaines boucles conditionnelles peuvent s'écrire de manière inconditionnelle quand on peut estimer le nombre maximal d'itérations.

Dans le cas de l'algorithme de Newton, dont la vitesse de convergence est quadratique, le module `scipy.optimize` propose une version d'algorithme avec, par défaut, un nombre maximum d'itérations de 50.

Versions itératives :

```
def newton(f, fp, x0, eps = 10**-8):
    while abs(f(x0)) > eps:
        x0 = x0 - f(x0)/fp(x0)
    return x0
```

```
def newton(f, fp, x0, eps = 10**-8):
    for i in range(50):
        x0 = x0 - f(x0)/fp(x0)
        if abs(f(x0)) < eps:
            return x0
    return x0
```

Version récursive :

```python
def newton(f, fp, x0, eps = 10**-8):
    if abs(f(x0)) < eps:
        return x0
    else:
        return newton(f, fp, x0 - f(x0)/fp(x0), eps = 10**-8)
```

Il apparait donc qu'on peut écrire les algorithmes récursifs sous forme itérative. Quelques critères pour choisir :

- comment le problème est-il énoncé ? itératif ? récursif ?
- quelle forme est la plus simple à manipuler ? ex : back-tracking pour résoudre un sudoku à hypothèses.
- quid du temps de calcul (ou du nombre d'opérations élémentaires) ? complexité temporelle.
- quid de l'occupation en mémoire (taille de la pile) ? complexité spatiale ?

Attention ! le nombre de niveaux de récurrences autorisées par défaut par **Python** est de 1024. Cependant, il est possible de changer cette valeur :

```python
import sys
sys.setrecursionlimit(2000)
```

2 Algorithmes récursifs

2.1 Illustration

Lorsque résoudre un problème (composé de sous-problèmes. . .) revient à résoudre le même problème mais avec un autre argument, une forme récursive est envisageable.

Exemple : pour calculer $n!$, on peut considérer que cela revient à calculer $(n-1)!$ qu'on multiplie ensuite par n.

Pour que le problème s'arrête, il lui faut une condition d'arrêt : $n \leq 1$, pour $n!$ renvoie 1.

Attention ! Il faut aussi que cette condition d'arrêt puisse être rencontrée sinon l'algorithme tourne sans fin !

Exemple : Changement salle à 13h . . .

Le prof n'est pas motivé pour appeler ses 48 élèves pendant la pause déjeuner et choisit donc d'appeler le premier et le dernier élève de la liste alphabétique.
Au premier, il confit la tâche d'appeler les 24 premiers élèves de la liste (en fait 23 élèves, car l'élève ne s'appellera pas lui même) ; au second, les 24 derniers. L'élève 1 appelle l'élève 2 et l'élève 24 ; l'élève 48 appelle l'élève 25 et l'élève 47.
Chacun passe au plus 2 appels. Si chaque appel prend une minutes, le prof aurait mis 48 minutes à appeler tout le monde. Avec l'appel récursif, tous les élèves sont prévenus en 9 min (et aurait pu faire mieux. . .).

Diviser les problèmes en deux sous problèmes permet d'améliorer la complexité temporelle des algorithmes. Ce le principe de base du *diviser pour mieux régner* du prochain cours.

2.2 Structure

Dans un algorithme récursif, on distingue deux parties :

- **l'initialisation**, liée à la condition d'arrêt. On parle aussi de cas de base.
- **l'hérédité**, contenant au final l'appel récursif.

Même s'il est possible de faire le contraire, le mieux est de commencer par la condition d'arrêt. L'hérédité pouvant contenir de nombreuses lignes, on gagne un cran d'indentation et on a moins de chance d'oublier la condition d'arrêt.

2.3 Récursivité terminale et récursivité non terminale

Une fonction récursive est dite terminale (ou récursive finale) si lors de l'hérédité, l'appel récursif est la dernière instruction à être évaluée. Ainsi aucun traitement ne doit être effectué à la remonté d'un appel récursif (sauf le retour de la valeur).

```python
def fctterminale(n):
    ...
    return fctterminale(n-1)
```

```python
def fctnonterminale(n):
    ...
    return n * fctnonterminale(n-1)
```

```python
def fac_NT(n):
    if n < 2:
        return 1
    else:
        return n * fac_NT(n-1)
```

```python
def fac_T(n, acc):
    if n < 2:
        return acc
    else:
        return fac_T(n-1, n*acc)
```

On peut démontrer que tout fonction récursive non terminale peut s'écrire de façon terminale. De même, toute fonction récursive terminale peut s'écrire de façon itérative.

Cependant, ce n'est pas parce que c'est possible, que c'est facile ! Il faut prendre l'habitude de passer d'une forme à l'autre (itérative/récursive) avec les algorithmes simples pour être en mesure d'utiliser la forme la moins difficile d'un problème complexe.

Dans le cas de `fac_NT`, les calculs se font à la remontée, ce qui oblige à stocker les valeurs intermédiaires. C'est plus lent.

```
 Fac_NT(4)
    ↑____ return 4 * Fac_NT(3)
  24              ↑____ return 3 * Fac_NT(2)
                  6              ↑____ return 2 * Fac_NT(1)
Fac_T(4, 1)                      2              ↑____ return 1
    └───→ return Fac_T(3, 4)                    1
                  └───→ return Fac_T(2, 12)
                                 └───→ return Fac_T(1, 24)
                                                └───→ return 24
```

Dans le cas de `Fac_T`, aucune valeur n'est stockée ; l'accumulateur est passé en argument de la fonction.

Attention ! copier des listes peut devenir très long suivant leur taille. Mieux vaut les manipuler à la même adresse.

3 Suites numériques

3.1 Suites définies de façon explicite

Il existe des suites pouvant se mettre directement sous la forme $u_n = f(n)$.

La suite est définie de façon explicite ; il suffit de connaître f.

3.2 Suites récurrentes

Dans une suite récurrentes, les termes sont définis par une relation de récurrence à partir d'un ou plusieurs termes précédents.

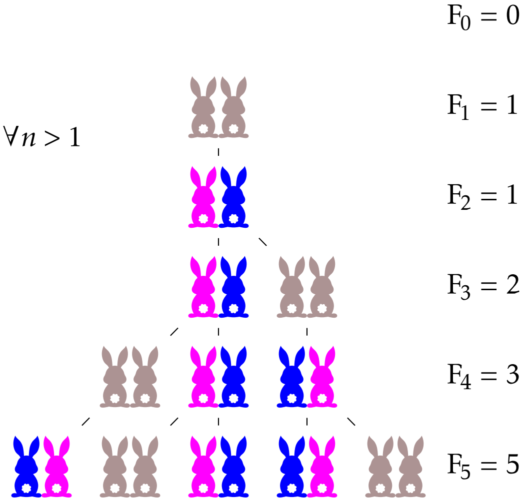
Exemple : $u_n = f(u_{n-1})$, $u_n = g(u_{n-2})$, $u_n = h(u_{n-1}, u_{n-2})\dots$

Il faut donc connaitre les conditions initiales puis utiliser la relation de récurrence jusqu'à obtenir u_n.

3.3 Suite de Fibonacci

3.3.1 Définition et illustration

La suite de Fibonacci est définie par :
$$\begin{cases} F_0 & = & 0 \\ F_1 & = & 1 \\ F_n & = & F_{n-1} + F_{n-2} \quad \forall n > 1 \end{cases}$$



Cette suite est associée (avec ces valeurs initiales $F_0 = 0$ et $F_1 = 1$) à l'évolution du nombre de couples de lapins au sein d'une population qui suit les règles suivantes :

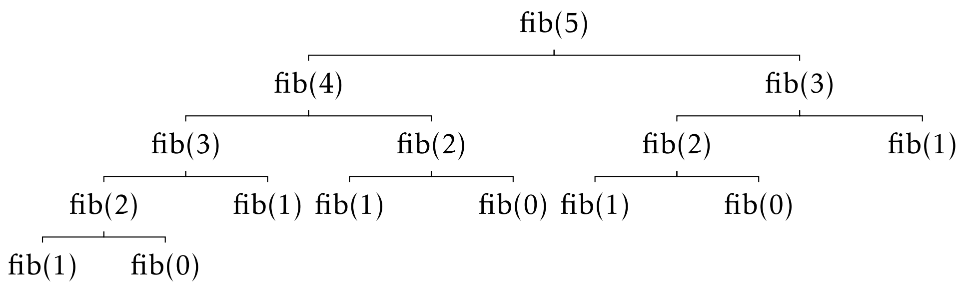
- chaque saison, un couple de lapins adultes depuis au moins une saison, donne naissance à un couple de lapereaux.
- il faut attendre une saison pour qu'un couple de lapereaux devienne adulte.
- un couple de lapereaux apparait saison 1.

3.3.2 Versions récursives

A première vu, la relation de récurrence invite à choisir une méthode récursive non terminale avec au rang n, l'appel de $F(n-1)$ et $F(n-2)$:

```
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)
```

Hélas, si l'algorithme est simple, le coût, lui, devient vite très élevé car on recalcule plusieurs fois les mêmes coefficients.

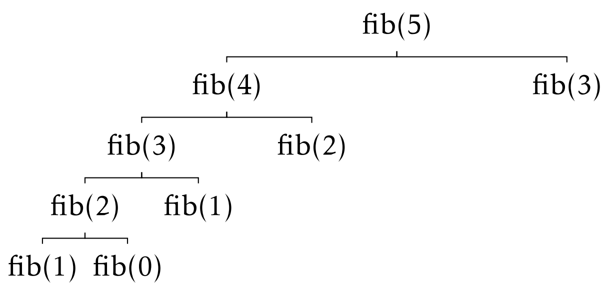


Au total on aura calculé :

- 1 fois fib(4)
- 2 fois fib(3)
- 3 fois fib(2)
- 5 fois fib(1)
- 3 fois fib(0)

Une des solutions consiste à utiliser une liste pour stocker les valeurs déjà calculées :

```
def fib(n, L):
    if n < 2:
        return n
    if len(L) <= n :
        Fibn = fib(n-1, L) + fib(n-2, L)
        L.append(Fibn)
    return L[n]
```



3.3.3 Versions itératives

Dans sa version itérative, il suffit de calculer les termes au fur et à mesure.

Avec stockage des valeurs :

```
def fib(n):
    if n < 2:
        return n
    L = [0, 1]
    for i in range(2, n+1):
        L.append(L[-1] + L[-2])
    return L[-1]
```

Sans stockage des valeurs :

```
def fib(n):
    if n < 2:
        return n
    x0, x1 = 0, 1
    for i in range(2, n+1):
        x0, x1 = x1, x0 + x1
    return x1
```

3.3.4 Version explicite

Le énième terme de la suite de Fibonacci est bien connu :

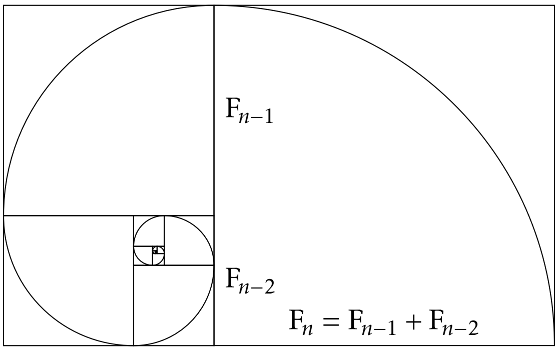
$$F_n = \frac{1}{\sqrt{5}}\cdot\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

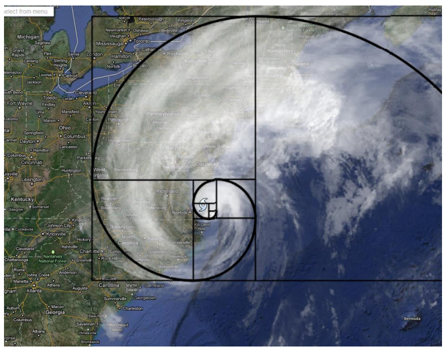
```
def fib(n):
    x = m.sqrt(5)
    yp = (1 + x)/2
    ym = (1 - x)/2
    return (yp**n + ym**n)/x
```

4 Côté artistique

4.1 Récursivité dans la nature

4.2 Encore Fibonacci...





4.3 Quid de LaTeX ?

Flocons de neige
de Von Koch