• Les algorithmes proposés dans les chapitres précédents sont itératifs, cela signifie qu'ils sont conçus à l'aide de boucles *for* ou *while*. Ce chapitre présente une autre manière d'écrire un algorithme avec la notion de **récursivité**. La récursivité est une notion fondamentale en informatique théorique et en programmation qui a la réputation d'être difficile, déroutante et mystérieuse.

Définition : Un algorithme récursif est un algorithme récursif.

• Cette définition illustre bien le concept de récursivité, **une fonction récursive est une fonction qui fait appel à elle-même**. D'ailleurs, comme l'a dit Stephen Hawking :

"To understand recursion, one must first understand recursion."

• Les objectifs de ce chapitre sont les suivants :

- ▶ savoir écrire un algorithme récursif et comprendre son fonctionnement,
- ▶ être capable de transformer un programme récursif en programme itératif et réciproquement,
- ▶ étudier la validité et la complexité d'une fonction récursive.

1 Généralités et premiers exemples

1.1 Calcul de la factorielle d'un entier

1.1.1 Version itérative

• On va présenter ici l'exemple le plus classique pour appréhender la notion de récursivité. On rappelle l'algorithme itératif suivant qui renvoie $n!$.

```
def fact1(n):
    """renvoie n! où n est un entier naturel"""
    p = 1
    for i in range(1, n + 1):
        p = p * i
    return(p)
```

• Cette fonction se comprend bien, elle se base sur la formule suivante :

$$n! = 1 \times 2 \times ... \times (n-1) \times n$$

On remarque que si $n = 0$ alors on ne rentre pas dans la boucle et la fonction renvoie bien 1 conformément à la convention $0! = 1$. On sait également que la complexité de cet algorithme, en terme de nombre de multiplications, est linéaire.

1.1.2 Version récursive

• Voici un algorithme récursif qui réalise également le calcul de $n!$.

```
def fact2(n):
    """renvoie n! où n est un entier naturel"""
    if n == 0:
        return(1)
    else:
        return(n * fact2(n - 1))
```

• Cette fonction se comprend très bien puisqu'elle fait appel à la définition mathématique suivante de la factorielle :

$$\begin{cases} 0! = 1 \\ \forall n \in \mathbb{N}^*,\ n! = n \times (n-1)! \end{cases}$$

• Par exemple lorsque l'on fait appel à $fact2(3)$, la fonction fait appel à $fact2(2)$ qui fait appel à $fact2(1)$ et qui fait appel à $fact2(0)$. Cette dernière fonction renvoie 1 et les différents produits s'effectuent pour renvoyer $n!$. Cette fonction fait appel à elle-même, c'est la particularité de la récusivité.

• Nous verrons que pour de très nombreux problèmes mathématiques, il est naturel d'utiliser une fonction récursive.

1.2 Principes généraux

L'exemple précédent permet de mettre en évidence quelques principes concernant l'implémentation d'une fonction récursive.

• **Une fonction récursive doit contenir une ou plusieurs conditions d'arrêt**, sinon le programme peut boucler indéfiniment. Dans l'exemple de la fonction $fact2$, c'est la condition $if\ n\ ==\ 0$ qui assure l'arrêt de la fonction. Plus précisément, la suite des valeurs de n passées en paramètre est une suite strictement décroissante d'entiers naturels, nous allons donc atteindre la condition d'arrêt (c'est-à-dire la valeur $n = 0$) en un nombre fini d'étapes.

• Par contre, si l'on essaie $fact2(-2)$, l'algorithme ne se termine pas puisqu'il y a aura les appels successifs : $fact2(-3)$, $fact2(-4)$, $fact2(-5)$...et l'on ne tombera jamais sur la condition d'arrêt. Toutefois, il y a un mécanisme de sécurité en Python qui limite le nombre d'appels récursifs. Par défaut cette limite est de l'ordre du millier et s'il y a un dépassement, un message d'erreur sera affiché :

```
>>> fact2(1200) # on teste avec n=1200
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<tmp 1>", line 13, in fact2
    return(n * fact2(n - 1))
  File "<tmp 1>", line 13, in fact2
    return(n * fact2(n - 1))
  File "<tmp 1>", line 13, in fact2
    return(n * fact2(n - 1))
  [Previous line repeated 984 more times]
  File "<tmp 1>", line 10, in fact2
    if n == 0:
RecursionError: maximum recursion depth exceeded in comparison
```

• Il est toutefois possible de modifier le nombre maximal d'appels récursifs avec les instructions suivantes :

```
import sys # un module qui gère certains paramètres du système
sys.setrecursionlimit(5000) # la limite est fixée à 5000
```

• Dans certains environnements l'usage de la récursivité n'est pas permis, c'est le cas dans le secteur des logiciels embarqués dans les véhicules terrestres ou aériens ou encore dans les standards de codage de la NASA (c'est la règle 1 des 10 règles fondamentales de codage de la NASA).
En effet, une erreur dans ce type de contexte pourrait avoir des conséquences importantes.

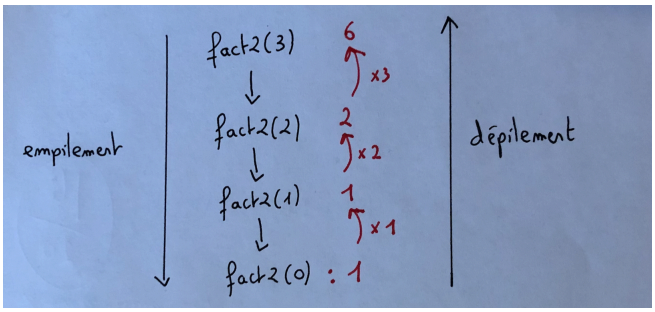
• **Les valeurs passées en paramètre dans les différents appels récursifs doivent être différentes**, sinon la fonction s'exécute à chaque appel de façon identique et continue de s'exécuter indéfiniment. Par exemple, il est évident qu'il est inutile d'implémenter la fonction suivante :

```
def test(n):
    return(test(n))
```

1.3 Pile de récursivité

• Il existe deux structures linéaires courantes en informatique : les **files** et les **piles**. Dans une file, c'est le premier arrivé qui est le premier sorti, on peut penser à une file d'attente dans une boulangerie. Dans une pile, c'est le dernier arrivé qui est le premier sorti, on peut penser à une pile de livres posée sur une table : c'est le livre posé en dernier, sur le dessus de la pile, qui sera pris en premier. C'est ce mécanisme de pile qui se retrouve dans la plupart des fonctions récursives.

• Voici précisément ce qu'il se passe lorsque l'on fait appel à $fact2(3)$:



Les différents appels sont **empilés** par ordre décroissant de n puis dépilés en partant du dernier.

• Lors de l'appel à $fact2(n)$, la multiplication $n \times fact2(n-1)$ est mise en attente puisque la valeur de $fact2(n-1)$ n'est pas connue. La mémoire de la machine est donc sollicitée pour stocker les informations en attente. On peut contourner cela en stockant les informations dans un paramètre supplémentaire, appelé un accumulateur, cela permet de doter la fonction d'un peu de mémoire. Voilà ce que cela donnerait pour la fonction factorielle :

```
def fact3(n, acc):
    """renvoie n! en conservant les résultats au fur et à mesure"""
    if n > 0:
        return(fact3(n - 1, acc * n))
    else:
        return(acc)
```

• Cette fonction renvoie la valeur de $acc * n!$ ainsi $fact3(n, 1)$ renvoie $n!$. Il n'y a aucune opération en attente hormis les appels récursifs.

2 Complexité d'une fonction récursive

2.1 Quelques exemples

• La plupart du temps, la complexité d'une fonction récursive se détermine à l'aide d'une relation de récurrence. On reprend l'exemple fondamental de la factorielle :

```
def fact2(n):
    """renvoie n! où n est un entier naturel"""
    if n == 0:
        return(1)
    else:
        return(n * fact2(n - 1))
```

On note $T(n)$ le nombre d'opérations effectuées lors de l'appel de $fact2(n)$. À chaque appel, on a deux opérations élémentaires effectuées : le test $n == 0$ et la multiplication. On en déduit la relation de récurrence :

$$\begin{cases} T(0) = 1 \\ \forall n \geq 1,\ T(n) = T(n-1) + 2 \end{cases}$$

C'est une suite arithmétique, on en déduit que pour tout $n \in \mathbb{N}$, $T(n) = 1 + 2n$. La complexité est linéaire.

• Voici un autre exemple :

```
def test(n):
    """une fonction pas très utile"""
    if n == 0:
        return ([])
    else:
        return([test(n - 1), test(n - 1), test(n - 1)])
```

On note également $T(n)$, le nombre d'opérations effectuées lors de l'appel de $test(n)$. On a $T(0) = 1$ ce qui correspond au test $n == 0$ et pour $n \geq 1$, $T(n) = 3T(n-1) + 1$ puisque l'on fait 3 fois appel à $test(n-1)$ et que l'on effectue une fois le test $n == 0$. On a :

$$\begin{cases} T(0) = 1 \\ \forall n \geq 1,\ T(n) = 3T(n-1) + 1 \end{cases}$$

C'est une suite arithmético-géométrique, on en déduit que :

$$\forall n \in \mathbb{N},\ T(n) = \frac{3^{n+1} - 1}{2}$$

La complexité est en $O(3^n)$, c'est une complexité exponentielle.

2.2 L'exemple de suite de Fibonacci

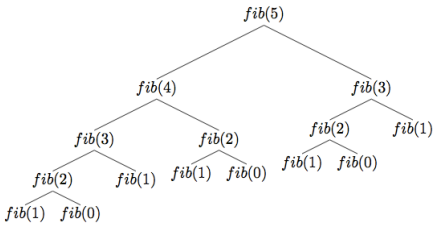
• La suite de Fibonacci étant elle-même définie par récurrence, son calcul semble bien se prêter à une implémentation récursive. On rappelle la définition de la suite de Fibonacci :

$$\begin{cases} F_0 = 0 \text{ et } F_1 = 1 \\ \forall n \in \mathbb{N},\ F_{n+2} = F_{n+1} + F_n \end{cases}$$

Voici un algorithme récursif qui renvoie les termes de la suite de Fibonacci, c'est un algorithme très naturel car il traduit simplement la formule de calcul :

```python
def fib(n):
    """renvoie le n-ième terme de la suite de Fibonacci"""
    if n == 0 or n == 1:
        return(n)
    else:
        return(fib(n - 1) + fib(n - 2))
```

• En faisant quelques tests, il apparait que pour des valeurs supérieures à 30, le calcul devient lent. En effet, à chaque étape le nombre d'appels récursifs est quasiment multiplié par 2, voici ce qu'il se passe si l'on calcule $fib(5)$:



• On se rend compte que notre algorithme souffre d'un problème de mémoire, il passe son temps à calculer des valeurs qu'il a déjà calculées mais qu'il n'a pas mémorisées. On peut pallier cela en créant un dictionnaire avec les valeurs déjà calculées, tout en conservant le caractère récursif de l'algorithme :

```python
memoire = {} # création du dictionnaire vide
def fib2(n):
    """renvoie le n-ième terme de la suite de Fibonacci en sauvegardant les valeurs calculées"""
    if n in memoire: # si c'est une valeur déjà calculée
        return(memoire[n])
    if n == 0 or n == 1:
        v = n
    else:
        v = fib2(n - 1) + fib2(n - 2)
    memoire[n] = v # on sauvegarde la nouvelle valeur pour les prochains appels
    return(v)
```

• L'algorithme est alors considérablement plus rapide mais nous n'évitons pas le problème de limitation du nombre maximal d'appels récursifs par rapport à l'algorithme de Fibonacci itératif.

• Grâce à cet exemple, nous garderons en tête qu'il faut faire attention à l'explosion du nombre d'appels récursifs.

3 Retour sur quelques algorithmes classiques

3.1 Exponentiation rapide

• On rappelle le principe de l'exponentiation rapide pour calculer x^n où $n \in \mathbb{N}$ et x est un flottant.

$$\begin{cases} x^n = (x^k)^2 \text{ si } n \text{ est pair avec } n = 2k \\ x^n = x \times (x^k)^2 \text{ si } n \text{ est impair avec } n = 2k+1 \end{cases}$$

Cela nous permet d'écrire directement l'algorithme récursif suivant :

```
def expo(x, n):
    """exponentiation rapide récursive"""
    if n == 0:
        return(1)
    else:
        if n % 2 == 0:
            return(expo(x ** 2, n // 2)) # cas de l'exposant pair
        else:
            return(x * expo(x ** 2, n // 2)) # cas de l'exposant impair
```

• Cet algorithme est plus simple à comprendre que sa version itérative. On peut démontrer que la complexité ne change pas, elle est en $O(\log(n))$.

3.2 Recherche dans une liste triée

• L'algorithme de recherche d'un élément dans une liste triée a lui aussi sa version récursive :

```
def recherche_dico_rec(L, x):
    """Renvoie True si l'élément x se trouve dans la liste triée L et False sinon"""
    if L == []:
        return(False)
    else:
        m = len(L) // 2 #le milieu de la liste
        if x == L[m]: # cas où l'élément est au milieu
            return(True)
        elif x > L[m]:
            return(recherche_dico_rec(L[m + 1:], x)) # moitié droite de la liste
        else:
            return(recherche_dico_rec(L[:m], x) # moitié gauche de la liste
```

• La complexité est toujours en $O(\log(n))$.

4 Validité d'un algorithme récursif

• Nous avons vu que la terminaison d'un algorithme récursif est lié à la condition d'arrêt qui doit figurer dans l'algorithme. La correction d'un algorithme récursif se démontre par récurrence. À titre d'exemple, justifions la correction de l'algorithme donnant la factorielle d'un entier que l'on rappelle ici :

```
def fact2(n):
    """ n est un entier positif """
    if n == 0:
        return(1)
    else:
        return(n * fact2(n - 1))
```

On considère la propriété suivante :

$$P_n \ : \text{" la fonction fact2(n) renvoie } n!\text{"}$$

▶ P_0 est vraie puisque c'est la condition d'arrêt.

▶ Supposons que la propriété P_n soit vraie pour un entier naturel n fixé. La fonction $fact2(n+1)$ renvoie $(n+1) \times fact2(n)$. Or $fact2(n)$ vaut $n!$ par hypothèse de récurrence donc $fact2(n+1)$ renvoie bien $(n+1) \times n! = (n+1)!$. On en déduit que P_{n+1} est vraie ce qui termine la récurrence.

▶ D'après le principe de récurrence, la propriété est démontrée et la correction de l'algorithme est assurée.

• Les fonctions récursives qui illustrent ce cours se comprennent relativement bien mais ce n'est pas toujours le cas. On termine ce cours en mentionnant la célèbre fonction d'Ackermann :

```
def ack(m, n):
    if m == 0:
        return(n + 1)
    elif m > 0 and n == 0:
        return(ack(m - 1, 1 ))
    else:
        return(ack(m - 1, ack(m, n - 1)))
```

• Comprendre ce que renvoie la fonction et démontrer sa validité est déjà beaucoup moins aisé. Le calcul de $ack(4,4)$ met les ordinateurs à rude épreuve.