# 1 Algorithmes récursifs

Une poupée russe, c'est une poupée avec une poupée russe à l'intérieur.

Figure 1: Les poupées russes (source Wikipedia)

Objectifs
- Découvrir une autre forme d'expression d'algorithmes
- Plusieurs types de récursivité
- La récursivité en Python

## 1.1 Introduction

En programmation, nombreux sont les problèmes qu'on résout en répétant plusieurs fois des séquences d'instructions.

Certains langages sont munis de structures de contrôles répétitive. C'est le cas notamment pour Python, qui dispose des boucles `pour` (`for`) et `tant que` (`while`).

Mais certains problèmes se résolvent simplement en résolvant un sous problème de même nature, mais plus simple...

Cette méthode de résolution s'appelle la **récursivité**.

### 1.1.1 Exemple 1 : factorielle

On souhaite calculer $n!$. On rappelle que pour tout entier $n \geqslant 0$ :

$$n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n$$

On peut déduire de cette définition une propriété importante

$\forall n. n \geqslant 1 \Rightarrow n! = n \times (n-1)!$

qui prouve que si on sait calculer $(n-1)!$ alors on sait calculer $n!$.

Par ailleurs, on sait que $0! = 1$. On sait donc calculer $1!$ puis $2!$, et par récurrence on peut établir qu'on sait calculer $n!$ pour tout entier $n \geqslant 0$.

### 1.1.2 Exemple 2 : dérivation d'une fonction numérique

Lorsqu'on doit calculer la dérivée une fonction numérique d'une variable , on a à notre disposition un ensemble de théorèmes de dérivation qu'on peut appliquer :

- $(u+v)' = u' + v'$
- $(u-v)' = u' - v'$
- $(uv)' = u'v + uv'$
- $\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{v^2}$
- $(u \circ v)' = (u' \circ v) \times v'$
- etc ...

Néanmoins pour appliquer ces règles il faut savoir dériver. Heureusement, on connait les dérivées des fonctions élémentaires :

- $x \longmapsto x^n$
- $x \longmapsto k$
- cos, sin, ....

Par conséquent, on sait calculer la dérivée de toutes les fonctions définies par opérations et fonctions usuelles. Encore une fois, le calcul est récursif.

#### 1.1.3 Exemple 3 : les tours de Hanoï

On dispose de trois tours côte à côte sur lesquelles on peut empiler des disques de diamètre croissant.

Initialement, seule la tour la plus à gauche contient des disques, les deux autres sont vides.

On ne peut déplacer qu'un disque à la fois et on ne doit pas poser un disque plus large sur un disque plus étroit.

Le problème des tours de Hanoï est de trouver la suite de déplacements qui permet de placer tous les disques sur la tour la plus à droite en respectant les contraintes imposées aux déplacements.

### 1.2 Algorithme récursif

#### 1.2.1 Définition

définition
Un algorithme de résolution d'un problème $P$ sur une donnée $a$ est dit *récursif* si parmi les opérations utilisées pour le résoudre, on trouve la résolution du même problème $P$ sur une donnée $b$. Dans un algorithme récursif, on nomme *appel récursif* toute étape de l'algorithme résolvant le même problème sur une autre donnée.

#### 1.2.2 Exemple 1 : factorielle

L'algorithme récursif de calcul de la factorielle distingue deux cas. Le premier cas ne nécessite aucun calcul, le second utilise la fonction `fact` pour calculer $(n-1)!$.

**Entrée :** $n \in \mathbb{N}$

**Sortie :** $n!$

1. **si** $n = 0$
2. **Renvoyer** 1
3. **sinon**
4. **Renvoyer** $n \times$ `fact`$(n-1)$
5. **fin si**

#### 1.2.3 Exemple 2 : Dérivation

Voici (une esquisse) de l'algorithme d'une fonction récursive de dérivation (nommée ici `derivee`).

**Entrée :** $f$ une fonction dérivable

**Sortie :** $f'$ la fonction dérivée

1. **si** $f$ est une fonction élémentaire de base
2. **Renvoyer** sa dérivée
3. **sinon si** $f = u + v$
4. **Renvoyer** `derivee`$(u)$ + `derivee`$(v)$
5. **sinon si** $f = u \times v$
6. **Renvoyer** `derivee`$(u) \times v + u \times$ `derivee`$(v)$
7. **sinon si** ...

### 1.2.4 Exemple 3 : Les tours de Hanoï

Et voici un algorithme récursif pour résoudre le problème des tours de Hanoi. Cet algorithme est celui d'une fonction nommée `hanoi` à trois paramètres

- le nombre de disques à déplacer
- la tour de départ où se trouvent ces disques
- la tour d'arrivée où les disques doivent être placés.

**Entrée :** $n \in \mathbb{N}$ le nombre de disques à déplacer, $d$ la tour où ils se trouvent, $a$ la tour où on doit les mettre.

**Action :** les déplacements des disques pour les amener sur le

1. **si** $n > 0$
2. $aux \leftarrow$ le piquet différent de $d$ et $a$
3. `hanoi`$(n-1, d, aux)$
4. `deplacer-disque`$(d, a)$
5. `hanoi`$(n-1, aux, a)$
6. **fin si**

on peut remarquer que dans cet algorithme il n'y a pas de **sinon**. Ceci s'explique par le fait que lorsque $n = 0$, il n'y a rien à faire.

### 1.2.5 Exécutions d'algorithmes récursifs

Voici le déroulement du calcul récursif de 4! :

$$\begin{aligned}
\texttt{fact}(4) &= 4 \times \texttt{fact}(3) \\
&= 4 \times 3 \times \texttt{fact}(2) \\
&= 4 \times 3 \times 2 \times \texttt{fact}(1) \\
&= 4 \times 3 \times 2 \times 1 \times \texttt{fact}(0) \\
&= 4 \times 3 \times 2 \times 1 \times 1 \\
&= 24
\end{aligned}$$

On note que des calculs restent en attente, jusqu'à ce qu'on atteigne le cas où il ne reste plus d'appel à la fonction `fact` dans l'expression à calculer.

### 1.2.6 Règles de conception

Il existe des algorithmes récursifs qui ne produisent aucun résultat.

En voici un exemple

**Entrée :** $n \in \mathbb{N}$

**Sortie :** $n!$

1. **Renvoyer** $n \times$ `fact2`$(n-1)$

L'évaluation de `fact2(1)` conduit à un calcul infini:

$$\begin{aligned}
\texttt{fact2}(1) &= 1 \times \texttt{fact2}(0) \\
&= 1 \times 0 \times \texttt{fact2}(-1) \\
&= \ldots
\end{aligned}$$

D'où la première régle de conception d'un algorithme récursif :

Règle 1
Tout algorithme récursif doit distinguer plusieurs cas dont l'un au moins ne doit pas contenir d'appels récursifs. Sinon il y a risque de cercle vicieux et de calcul infini.

Les cas non récursifs d'un algorithme récursif sont appelés *cas de base*. Les conditions que doivent satisfaire les données dans ces cas de base sont appelées *conditions de terminaison*

Même avec un cas de base un algorithme récursif peut ne produire aucun résultat. En voici un exemple :

**Entrée :** $n \in \mathbb{N}$

**Sortie :** $n!$

1. **si** $n = 0$, **alors**
2. **Renvoyer** 1
3. **sinon**
4. **Renvoyer** $\frac{\texttt{fact3}(n+1)}{n+1}$
5. **fin si**

Là encore, le calcul de `fact3(1)` donne lieu à un calcul infini, malgré la présence d'un cas de base. Mais cette fois le calcul infini est dû à des appels récursifs sur des données s'éloignant du cas de base.

D'où la seconde règle de conception d'un algorithme récursif :

Règle 2
Tout appel récursif doit se faire avec des données plus proches de données satisfaisant les conditions de terminaison.

La remarque suivante est assez utile, lorsqu'on souhaite prouver qu'un algorithme récursif s'arrête.

Théorème
Il n'existe pas de suite infinie strictement décroissante d'entiers positifs ou nuls.

Pour prouver la terminaison d'un algorithme récursif, on peut démontrer l'existence d'un **variant**

définition: Variant
supposons que pour calculer $f(x_1, x_2, ...)$, un algorithme récursif $f$ doit évaluer $f(y_1, y_2, ...)$.
Un *variant* $v$ de $f$ est une fonction :
- des paramètres de $f$ ;
- à valeurs dans $\mathbb{N}$ ;
- vérifiant $v(y_1, y_2, ...) < v(x_1, x_2, ...)$

Par exemple un variant pour la fonction `fact` est $v(n) = n$ (puisque pour calculer $f(n)$, on doit évaluer $f(n-1)$). Comme il n'existe pas de suite infinie d'entiers strictement décroissante, l'algorithme récursif se termine.

## 1.3 Type de récursivité

### 1.3.1 Récursivité simple ou linéaire

définition
Un algorithme récursif est *simple* ou *linéaire* si chaque cas qu'il distingue se résout en au plus un appel récursif.
Ainsi l'algorithme `fact` de calcul de $n!$ est récursif simple.

### 1.3.2 Récursivité terminale

Définition
Un algorithme récursif simple est *terminal* lorsque la valeur renvoyée est celle de l'appel récursif (aucune opération n'est donc effectuée sur cette valeur).

Généralement ce genre d'algorithme peut facilement être transformé en une boucle. D'ailleurs certains compilateurs sont capables de détecter ce genre de situation et de transformer automatiquement le code pour supprimer la récursivité.

Le schéma d'algorithme qui suit décrit les algorithmes récursifs terminaux. Les fonctions/instructions $p(x)$, $b(x)$, $c(x)$ auxquelles il fait appel ne font pas appel à $A$.

**Entrée :** $x$ la donnée que doit traiter $A$

1. **si** $p(x)$
2. **renvoyer** $b(x)$
3. **sinon**
4. **renvoyer** $A(c(x))$
5. **fin si**

On peut facilement transformer cet algorithme en un algorithme itératif équivalent :

**Entrée :** $x$ la donnée que doit traiter $A$

1. **tant que** $\neg p(x)$
2. $x \leftarrow c(x)$

3. **fin tant que*
4. **renvoyer** $b(x)$

### 1.3.3 Récursivité multiple

Un algorithme récursif est *multiple* si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs.

L'exemple des tours de Hanoï, ou encore celui de la dérivation sont des exemples de récursivité multiple.

> un algorithme à récursivité multiple ne peut pas être terminal.

### 1.3.4 Récursivité croisée ou mutuelle

La définition des algorithmes récursifs donnée plus haut qui les caractérise comme étant les algorithmes faisant appel à eux mêmes masque le phénomène des algorithmes mutuellement récursifs.

> définition
> Deux algorithmes sont *mutuellement récursifs* si l'un fait appel à l'autre et l'autre à l'un. On peut étendre cette définition à un nombre quelconque d'algorithmes.

En voici un exemple. Deux fonctions nommées `pair` et `impair` déterminant la parité ou l'imparité d'un entier.

**Entrée :** $n \in \mathbb{N}$

**Sortie :** *Vrai* si $n$ est pair, *Faux* sinon

1. **si** $n = 0$
2. **Renvoyer** *Vrai*
3. **sinon**
4. **Renvoyer** `impair`$(n-1)$
5. **fin si**

**Entrée :** $n \in \mathbb{N}$

**Sortie :** *Vrai* si $n$ est impair, *Faux* sinon

1. **si** $n = 0$
2. **Renvoyer** *Faux*
3. **sinon**
4. **Renvoyer** `pair`$(n-1)$
5. **fin si**

## 1.4 Récursivité en Python

Python permet très simplement l'écriture de fonctions récursives.

### 1.4.1 Factorielle en Python

```
def fact(n: int) -> int:
    if n == 0:
        res =  1
    else:
        res = n * fact(n-1)
    return res
```

### 1.4.2 Prédicats de parité en Python

```
def pair(n):
    if n == 0:
       res = True
    else:
       res = impair(n-1)
    return res

def impair(n):
    if n == 0:
```

```
        res = False
    else:
        res = pair(n-1)
    return res
```

### 1.4.3 Limitation de la récursivité en Python

Dans sa configuration par défaut, le langage Python limite le nombre d'appels récursifs.

Par exemple, le calcul de `fact(1000)` n'est pas possible. Il se termine par une exception `RecursionError`[1] :

```
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<console>", line 5, in fact
  File "<console>", line 5, in fact
  File "<console>", line 5, in fact
  [Previous line repeated 979 more times]
RecursionError: maximum recursion depth exceeded
```

La fonction `sys.getrecursionlimit` donne la profondeur maximale des calculs récursifs :

```
1000
```

Si on veut augmenter cette profondeur on peut utiliser la fonction `sys.setrecursionlimit` (cependant il faut rester raisonnable).

```
40238726007709377354370243392300398571937486421071463254379991042993851239862902059204420848696940480042568
```

## 1.5 Conclusion

La récursivité offre au programmeur un autre moyen, souvent élégant et concis, de résoudre des problèmes.

Par exemple,

- dans la programmation des jeux solitaires du type Sudoku, labyrinthes, ...
- dans la programmation des jeux à deux joueurs du type Échecs, Dames, Othello, ...
- et dans bien d'autres domaines encore ...

  ... comme le point de croix en broderie (cf ici)

## 1.6 Notes

[1] L'exception ``'RecursionErrorest apparue dans la version 3.5 de Python. Pour les versions antérieures, l'exception étaitRuntimeError“.