Récursivité

1. Définition et idée générale

Un algorithme est **récursif** s'il s'appelle lui-même sur un « cas **plus petit** » du ***même* problème**. Il faut toujours un **cas de base *atteignable*** pour arrêter la récursion : on parle de **condition d'arrêt**.

Dangers !

Sans *condition d'arrêt*, ou sans réduction **réelle** de la « taille » du problème lors de l'appel récursif, un programme récursif bouclera à l'infini.

💭 **Remarque : pourquoi y a-t-il peu de risques en Python ?** ▼

Python embarque un mécanisme qui empêche une boucle infinie de survenir. Une exception `RecursionError` est levée lorsqu'on approche des 1000 itérations. **Pour la culture**, il est possible de modifier cette limite :

```python
import sys
sys.setrecursionlimit(10000)
```

2. Fonctionnement

Chaque appel « non conclusif » d'une fonction récursive déclenche la **mémorisation** de cet appel.

Cette mémorisation consiste à stocker dans une *pile* (la *pile d'appels*) **un nouveau « cadre » décrivant précisément cet appel** (nom de la fonction, noms et valeurs des paramètres).

Quand on arrive enfin au cas de base, on peut achever les appels mis en attente dans la pile (successivement et « à rebours », puisqu'il s'agit d'une *pile* : le ***dernier*** *appel* mémorisé est traité *en premier*).

Lorsqu'un appel dépilé se termine, on passe au *précédent* : la pile d'appels nous permet de reprendre précisément là où l'appel précédent avait été « mis en pause » (le moment où il a été empilé).

💭 **Remarque : appel non conclusif et mémorisation ?** ▼

- *« Non conclusif »* signifie que l'appel ne peut pas *à ce stade* renvoyer de résultat.
- La *mémorisation de **l'appel*** n'intervient pas qu'en cas de récurisivité : elle intervient *aussi* lorsqu'une fonction en appelle une autre.

💭 **Remarque : pile ?** ▼

Rappel : une pile est une structure de données qui fonctionne comme une pile d'assiettes. Le **dernier** « objet » à y entrer est *empilé au sommet*, et c'est le **premier** à être « dépilé » (on ne retire jamais ailleurs qu'au sommet, dans une pile !).

✏️ **Exemple : visualisation du calcul récursif de la puissance avec PythonTutor** ▼

Cliquez sur ce lien pour voir le déroulement des appels successifs, leur empilement, puis leur résolution avant d'obtenir le résultat.

Voici une illustration statique, jusqu'au moment où le code atteint la condition d'arrêt.

1. **Définition de la fonction et appel initial**

```
1  def puissance(a, n):
2      if n == 0:
3          return 1
4      else:
5          return puissance(a, n-1) * a
6
7  puissance(2, 5)
```

line that just executed
next line to execute

2. **Évolutions successives de la pile d'appels**

Étape 1	➡	Étape 2	➡	Étape 3
puissance a 2 n 5		puissance a 2 n 4		puissance a 2 n 3
Étape 4	➡	Étape 5	➡	Étape 6
puissance a 2 n 2		puissance a 2 n 1		puissance a 2 n 0 Return value 1

3. **Dépilements et calculs successifs**

   - Le dernier appel empilé est **`puissance(2, 0)`** (à l'étape 6). Comme `n=0`, **la condition d'arrêt est atteinte** dans le code : **cet appel renvoie `1`**.
   - L'avant-dernier appel empilé est **`puissance(2, 1)`** (à l'étape 5). On a `n=1`, le calcul en attente était `puissance(2, 0) * 2` et on vient d'obtenir `1` comme valeur de retour de l'appel `puissance(2, 0)`. Le calcul peut donc être terminé : `1 * 2` donne `2`. **Cet appel renvoie donc `2`**.

- L'antépénultième appel empilé est **`puissance(2, 2)`** (à l'étape 4). Pour `n=2`, le calcul en attente était `puissance(2, 1) * 2` et on vient d'obtenir `2` comme valeur de retour de l'appel `puissance(2, 1)`. Le calcul peut donc être terminé : `2 * 2` donne `4`. **Cet appel renvoie donc `4`**.
- L'appel précédemment empilé est **`puissance(2, 3)`** (à l'étape 3). Pour `n=3`, le calcul en attente était `puissance(2, 2) * 2` et on vient d'obtenir `4` comme valeur de retour de l'appel `puissance(2, 2)`. Le calcul peut donc être terminé : `4 * 2` donne `8`. **Cet appel renvoie donc `8`**.
- L'appel précédemment empilé est **`puissance(2, 4)`** (à l'étape 2). Pour `n=4`, le calcul en attente était `puissance(2, 3) * 2` et on vient d'obtenir `8` comme valeur de retour de l'appel `puissance(2, 3)`. Le calcul peut donc être terminé : `8 * 2` donne `16`. **Cet appel renvoie donc `16`**.
- Enfin, on revient au tout premier appel empilé : **`puissance(2, 5)`** (à l'étape 1). Pour `n=5`, le calcul en attente était `puissance(2, 4) * 2` et on vient d'obtenir `16` comme valeur de retour de l'appel `puissance(2, 4)`. Le calcul **global** peut donc être terminé : `16 * 2` donne `32`. **Cet appel renvoie donc `32`, qui est bien la valeur de 2^5**.

3. Quand employer la récursivité ?

La récursivité est naturelle quand un problème se définit à partir d'une version « plus petite » de lui-même. C'est le cas du calcul de la factorielle ou de celui des termes de la suite de Fibonacci, en mathématiques.

De nombreuses situations peuvent être exprimées ainsi :

- la somme des entiers d'une liste : c'est le résultat du premier entier de cette liste additionné à la somme des entiers de la liste *à partir du deuxième* ;
- la multiplication d'un nombre a par un entier n : si $n = 1$, le résultat est juste a (cas de base) ; sinon, $a \times n = a \times (n - 1) + a$ (on utilise la multiplication de a par l'entier $n - 1$, ce qui est une version « plus petite » du problème initial) ;
- la recherche d'une valeur dans une liste : si la valeur est à la position `0` de la liste, alors on renvoie `True` ; sinon, on renvoie le résultat de la recherche de cette valeur *à partir de la position 1* dans cette liste.

En NSI, le parcours d'arbres ou de graphes *en profondeur* ou le tri fusion sont des exemples archétypiques.

💭 **Remarque : mathématique *vs* informatique** ▼

- En mathématiques, on parle de *définition par récurrence* ; en programmation, on parle de *fonction récursive* ou de *récursivité*.
- Autre différence :
  - en mathématique, la *démonstration par récurrence* s'efforce de « gravir les marches d'un escalier » : on montre que si la propriété est vraie pour n, alors elle l'est pour $n + 1$;
  - en informatique, pour calculer $f(n)$ avec une fonction récursive, le programme fait *l'inverse* : il *descend* d'abord jusqu'au cas de base, **puis** *remonte* en utilisant les résultats intermédiaires.

    C'est un peu comme si on descendait un escalier et qu'on ramassait, uniquement en remontant, des chaussettes tombées du panier à linge. Concrètement :

- on part de la « marche `n` » (appel `f(n)`), puis on descend successivement aux « marches `n-1` », « `n-2` », etc., jusqu'au cas de base (par exemple la « marche `0` ») ;
- arrivé en bas, on trouve ce qu'on cherchait (la « chaussette » correspond à la valeur du cas de base), et on remonte marche par marche (en ramassant les autres chaussettes, c'est-à-dire en complétant les calculs en attente) jusqu'à la marche `n`.

Autre image mentale : imaginez une phrase inscrite sur des poupées russes, chacune portant un unique mot. Le début de la phrase est écrit sur la plus petite, « tout au fond ». Pour lire la phrase, on doit ouvrir la grande, puis celle qu'elle contient, etc., jusqu'à la plus petite. Là, on peut lire le premier mot de la phrase, puis on lit les mots portés par les poupées de plus en plus grandes. À la fin, on a reconstitué toute la phrase.

3.1. Exemple simple

Factorielle n, qu'on écrit $n!$ en mathématiques, vaut par définition $1 \times 2 \times \cdots \times (n-1) \times n$ (avec, **par convention**, $0! = 1$: c'est la condition d'arrêt). Donc $n! = (n-1)! \times n$ (formule de récurrence) : on a bien une définition *récursive*. Concrètement, en Python, on écrira :

```python
def factorielle(n):
    if n <= 1:
        return 1
    else:
        return n * factorielle(n - 1)
```

Ici, le cas de base est `n <= 1`. Chaque appel *réduit le problème* en passant de `n` à `n - 1`.

3.2. Exemple plus avancé

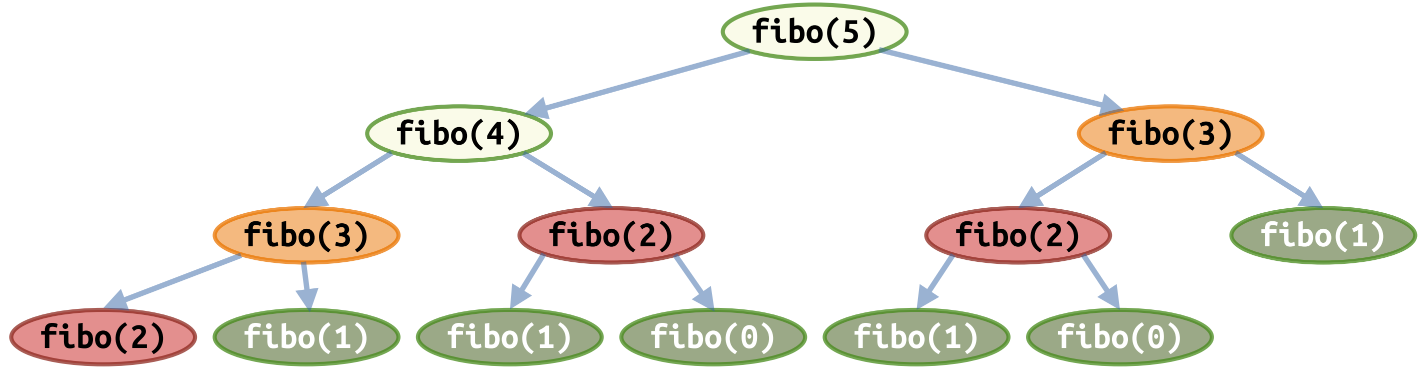
La suite de Fibonacci est également définie par récurrence :

- $F_0 = 0$ et $F_1 = 1$ (conditions d'arrêt) ;
- $F_n = F_{n-1} + F_{n-2}$ (formule de récurrence).

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

Cette version est correcte, mais elle est ***très** coûteuse en temps*, car elle recalcule plusieurs fois les mêmes valeurs. Outre le fait qu'elle illustre bien l'immédiateté avec laquelle une fonction récursive traduit un problème récursif en code, elle permet d'illustrer l'intérêt de la programmation dynamique (voir la fiche dédiée).

Illustration de la pile d'appels qui découle du calcul de `fibo(5)` :



> ⚠️ **Attention à l'efficacité !**
>
> Une implémentation naïve de la récursivité peut entraîner une explosion du nombre d'appels. Dans l'illustration ci-dessus, pour calculer `fibo(5)`, il faudra :
>
> - calculer une fois `fibo(4)` ;
> - calculer **deux fois** `fibo(3)` ;
> - calculer ***trois* fois** `fibo(2)`.
>
> Et cela serait pire si l'on voulait calculer `fibo(6)`, `fibo(7)`, etc. !
>
> **Note :** pas besoin de *calculer* `fibo(1)` ou `fibo(0)`, car ce sont les deux cas de bases (les valeurs de retour sont connues : `fibo(0)` vaut `0` et `fibo(1)` vaut `1`).

4. Points d'attention

- Le cas de base doit être ***vraiment* atteignable**.
- L'appel récursif doit être réalisée sur un cas **« plus simple »** ou **« plus petit »**.
- Une approche récursive peut être (très) inefficace.
- En NSI, il faut surtout comprendre la structure générale : un cas d'arrêt, un appel récursif, puis éventuellement une combinaison des résultats.

5. Liens avec d'autres notions

- La **programmation dynamique** évite les recalculs dans certaines récursions.
- Le **tri fusion** utilise la récursivité.
- Certains parcours d'**arbres** et de **graphes** sont naturellement récursifs.
- Le **problème de l'arrêt** rappelle qu'une récursion peut ne jamais se terminer.
- La **notion de pile** (d'appels) est à connaître.