# La récursivité


## RÉSUMÉ

Comment calculer $5!$ sans connaître la multiplication de tous les nombres de 1 à 5 ? En remarquant que $5! = 5 \times 4!$, et que $4! = 4 \times 3!$, et ainsi de suite jusqu'à $1! = 1$. C'est l'idée de la **récursivité** : résoudre un problème en le ramenant à une version plus petite de lui-même, jusqu'à atteindre un cas simple qu'on sait résoudre directement. Ce mécanisme, où une fonction s'appelle elle-même, est à la fois élégant et puissant : il permet de résoudre naturellement des problèmes sur les arbres, les graphes, les fractales ou les jeux. Mais il faut le maîtriser avec rigueur, sous peine de boucles infinies ou de dépassements de mémoire.


## ① MÉMO

### Principe de la récursivité

Une fonction est **récursive** si elle s'appelle elle-même. Toute fonction récursive doit comporter :

1. Un **cas de base** (condition d'arrêt) qui renvoie un résultat sans appel récursif.
2. Un **cas récursif** qui réduit le problème à un sous-problème plus simple et appelle la fonction sur ce sous-problème.

Sans cas de base, la récursion est infinie et provoque une erreur `RecursionError`.

### Pile d'appels

Chaque appel récursif empile un **cadre d'exécution** (frame) sur la pile d'appels, contenant les variables locales et l'adresse de retour. Quand le cas de base est atteint, les cadres se dépilent un à un.
La profondeur maximale par défaut en Python est de 1 000 appels (`sys.getrecursionlimit()`).

### Récursivité vs itération

- Tout algorithme récursif peut être réécrit en itératif (et inversement).
- La récursivité est souvent plus élégante et naturelle pour les structures arborescentes.
- Elle consomme plus de mémoire (pile d'appels) qu'une boucle.
- La complexité en temps est identique, mais la complexité en espace peut être plus élevée.

### Pièges fréquents

- Oublier le cas de base, ce qui provoque une récursion infinie.
- Le cas récursif ne réduit pas le problème, ce qui provoque également une récursion infinie.
- Double appel récursif sans mémoïsation, ce qui entraîne une complexité exponentielle (ex. : Fibonacci naïf).
- Modifier un paramètre mutable (liste) partagé entre les appels.

**Erreurs classiques**

| Code erroné | Code correct | Explication |
|---|---|---|
| `def fact(n):`<br>`return n * fact(n-1)` | Ajouter `if n == 0:`<br>`return 1` | Sans cas de base, la récursion ne s'arrête jamais et provoque un `RecursionError` (dépassement de la profondeur maximale). |
| `def fib(n):`<br>`return fib(n-1) +`<br>`fib(n-2)` | Utiliser la mémoïsation ou la version itérative | La double récursion sans mémoïsation engendre une complexité exponentielle $O(2^n)$ : `fib(40)` prend plusieurs secondes. |
| `def f(n, L=[]):`<br>`L.append(n)`<br>`return f(n-1, L)` | `def f(n, L=None):`<br>`if L is None: L = []` | Un paramètre mutable par défaut est partagé entre tous les appels : la liste accumule les valeurs des appels précédents. |
| `def somme_rec(L):`<br>`return L[0] +`<br>`somme_rec(L)` | `somme_rec(L[1:])` | Si l'appel récursif ne réduit pas le problème (ici on repasse la même liste), la récursion est infinie. Il faut transmettre `L[1:]`. |

## ② EXEMPLES

**Programme 1 · Factorielle**

```
def fact(n):
    if n == 0:          # cas de base
        return 1
    return n * fact(n - 1)  # cas récursif
```

**Déroulement de `fact(4)` :**

| Appel | Résultat |
|---|---|
| `fact(4)` | attend `4 * fact(3)` |
| `fact(3)` | attend `3 * fact(2)` |
| `fact(2)` | attend `2 * fact(1)` |
| `fact(1)` | attend `1 * fact(0)` |
| `fact(0)` | renvoie `1` (cas de base) |

Les retours se font en sens inverse : $1 \times 1 = 1$, puis $2 \times 1 = 2$, puis $3 \times 2 = 6$, puis $4 \times 6 = 24$.

**Programme 2 · Puissance récursive**

```
def puissance(x, n):
    if n == 0:
```

```
        return 1
    return x * puissance(x, n - 1)
# Complexité : O(n) en temps et en espace (n appels empilés)
```

### Programme 3 · Fibonacci naïf (à NE PAS utiliser en production)

```
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)
# Complexité : O(2^n) ! Chaque appel engendre deux appels.
```

### Programme 4 · Fibonacci avec mémoïsation

```
def fib_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fib_memo(n - 1, memo) + fib_memo(n - 2, memo)
    return memo[n]
# Complexité : O(n) en temps et en espace
```

### Programme 5 · Somme des éléments d'une liste

```
def somme_rec(L):
    if len(L) == 0:
        return 0
    return L[0] + somme_rec(L[1:])
# Attention : L[1:] crée une copie, donc O(n^2) en espace !

# Version avec indice (meilleure)
def somme_rec_v2(L, i=0):
    if i == len(L):
        return 0
    return L[i] + somme_rec_v2(L, i + 1)
```

### Programme 6 · Dichotomie récursive

```
def dicho_rec(L, x, g, d):
    if g > d:
        return -1
    m = (g + d) // 2
    if L[m] == x:
        return m
    elif L[m] < x:
        return dicho_rec(L, x, m + 1, d)
    else:
        return dicho_rec(L, x, g, m - 1)
```

## ③ EXERCICES

**Exercice 1** ***Trace d'appels*** Donner la trace complète (pile d'appels et valeurs de retour) pour :

1. `fact(5)`
2. `fib(5)` (version naïve) : dessiner l'arbre des appels et compter le nombre total d'appels.

**Exercice 2** ***Fonctions récursives***

1. Écrire une fonction récursive `longueur(L)` qui renvoie la longueur d'une liste sans utiliser `len`.
2. Écrire une fonction récursive `renverser(chaine)` qui renvoie une chaîne inversée.
3. Écrire une fonction récursive `est_palindrome(mot)` qui teste si un mot est un palindrome.

**Exercice 3** ***Puissance rapide récursive*** La puissance rapide exploite les relations :

- $x^0 = 1$ ;
- $x^n = (x^{n/2})^2$ si $n$ est pair ;
- $x^n = x \times x^{n-1}$ si $n$ est impair.

1. Écrire la fonction récursive `puissance_rapide(x, n)`.
2. Donner la trace pour `puissance_rapide(2, 10)`.
3. Quelle est la complexité de cette fonction ? Comparer avec la version naïve.

## SOLUTIONS DES EXERCICES

**Corrigé de l'exercice 1.** **1. Trace de `fact(5)` :**

| Appel | Action | Résultat |
|---|---|---|
| `fact(5)` | appelle `fact(4)` | attend |
| `fact(4)` | appelle `fact(3)` | attend |
| `fact(3)` | appelle `fact(2)` | attend |
| `fact(2)` | appelle `fact(1)` | attend |
| `fact(1)` | appelle `fact(0)` | attend |
| `fact(0)` | cas de base | renvoie `1` |
| `fact(1)` | $1 \times 1$ | renvoie `1` |
| `fact(2)` | $2 \times 1$ | renvoie `2` |
| `fact(3)` | $3 \times 2$ | renvoie `6` |
| `fact(4)` | $4 \times 6$ | renvoie `24` |
| `fact(5)` | $5 \times 24$ | renvoie `120` |

Six appels au total, profondeur maximale de la pile : 6.

**2. Arbre des appels de `fib(5)` :**

Pour calculer `fib(5)`, Python appelle `fib(4)` et `fib(3)`. Chacun engendre à son tour deux appels, et ainsi de suite jusqu'aux cas de base.

Au total, 15 appels sont nécessaires, dont beaucoup sont redondants : `fib(2)` est calculé trois fois, `fib(3)` deux fois. C'est cette explosion qui explique la complexité exponentielle $O(2^n)$.

**Corrigé de l'exercice 2.**

```python
def longueur(L):
    if L == []:
        return 0
    return 1 + longueur(L[1:])

def renverser(chaine):
    if len(chaine) <= 1:
        return chaine
    return chaine[-1] + renverser(chaine[:-1])

def est_palindrome(mot):
    if len(mot) <= 1:
        return True
    if mot[0] != mot[-1]:
        return False
    return est_palindrome(mot[1:-1])
```

**Vérification :**

— `longueur([3, 1, 4])` renvoie 3 ;
— `renverser("python")` renvoie `"nohtyp"` ;
— `est_palindrome("kayak")` renvoie `True` ;
— `est_palindrome("python")` renvoie `False`.

**Corrigé de l'exercice 3.**

```python
def puissance_rapide(x, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        demi = puissance_rapide(x, n // 2)
        return demi * demi
    else:
        return x * puissance_rapide(x, n - 1)
```

**Trace de `puissance_rapide(2, 10)` :**

| **Appel** | $n$ | **Parité** | **Action** |
|---|---|---|---|
| `puissance_rapide(2, 10)` | 10 | pair | calcule `pr(2,5)`$^2$ |
| `puissance_rapide(2, 5)` | 5 | impair | calcule $2\times$ `pr(2,4)` |
| `puissance_rapide(2, 4)` | 4 | pair | calcule `pr(2,2)`$^2$ |
| `puissance_rapide(2, 2)` | 2 | pair | calcule `pr(2,1)`$^2$ |
| `puissance_rapide(2, 1)` | 1 | impair | calcule $2\times$ `pr(2,0)` |
| `puissance_rapide(2, 0)` | 0 | | renvoie 1 (cas de base) |

Remontée : 1, puis $2 \times 1 = 2$, puis $2^2 = 4$, puis $4^2 = 16$, puis $2 \times 16 = 32$, puis $32^2 = 1024$. Six appels au lieu de dix pour la version naïve.

**Complexité :** $O(\log n)$ car on divise $n$ par 2 à presque chaque étape. La version naïve est en $O(n)$.