

Récurtivité

Fiche de synthèse

① FONCTION RÉCURSIVE

Définition 1 (Fonction récursive) Une fonction est **récursive** quand elle s'appelle elle-même. Elle comporte au moins un **cas de base**, traité sans appel récursif, qui arrête les appels, et un **appel récursif** sur un problème plus petit, qui se rapproche du cas de base.

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

Pour $n \geq 1$, $n! = n \times (n - 1)!$, et $0! = 1$: la fonction traduit directement cette définition.

✎ Écrire une fonction récursive.

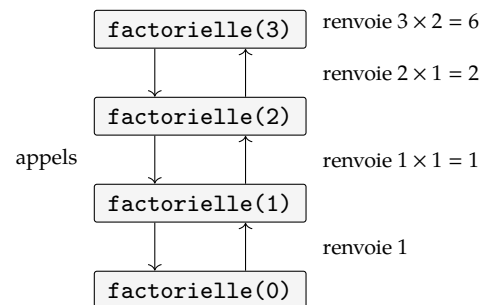
1. Trouver le cas le plus simple et sa réponse : le cas de base.
2. Exprimer le résultat à l'aide du même problème sur une donnée plus petite : $n - 1$, la liste privée de son premier élément `liste[1:]`, la chaîne privée de son premier caractère.
3. Vérifier que chaque appel se rapproche du cas de base, sinon les appels ne s'arrêtent jamais.

```
def somme(liste):
    if len(liste) == 1:
        return liste[0]
    else:
        return liste[0] + somme(liste[1:])

def inverse(s):
    if len(s) == 0:
        return s
    else:
        return inverse(s[1:]) + s[0]
```

② LA PILE D'APPELS

Définition 2 (Pile d'appels) À chaque appel de fonction, Python **empile** un contexte qui contient les paramètres et les variables locales de cet appel. Quand l'appel se termine, son contexte est **dépilé** et son résultat est transmis à l'appel qui l'a déclenché.

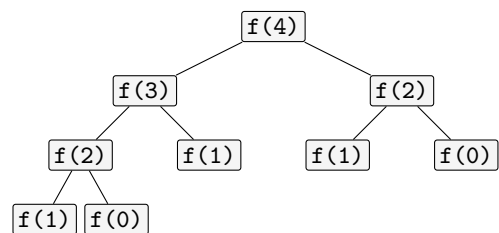


Les appels s'empilent jusqu'au cas de base, puis les résultats remontent de bas en haut. `factorielle(n)` demande $n + 1$ appels : la mémoire utilisée est proportionnelle à n .

③ APPELS MULTIPLES

```
def fibonacci(n):
    if n == 0 or n == 1:
        return n
    else:
        return (fibonacci(n - 1)
                + fibonacci(n - 2))
```

Chaque appel en déclenche deux. L'**arbre des appels** de `fibonacci(4)` montre que `fibonacci(2)` est calculé deux fois et `fibonacci(1)` trois fois :



Le nombre d'appels croît de façon exponentielle : `fibonacci(35)` en demande près de trente millions.

Définition 3 (Mémoïsation) La **mémoïsation** consiste à ranger chaque résultat calculé, par exemple dans un dictionnaire, pour le relire au lieu de le recalculer.

```
memo = {}

def fibonacci_memo(n):
    if n in memo:
        return memo[n]
    if n == 0 or n == 1:
        resultat = n
    else:
        resultat = (fibonacci_memo(n - 1)
                    + fibonacci_memo(n - 2))
    memo[n] = resultat
    return resultat
```

```
        + fibonacci_memo(n - 2))
memo[n] = resultat
return resultat
```

Chaque valeur n'est plus calculée qu'une fois : le nombre d'appels devient proportionnel à n .

④ LIMITES

- Python limite la profondeur de la pile d'appels à environ 1000 : `factorielle(1500)` provoque l'erreur `RecursionError: maximum recursion depth exceeded`. `sys.setrecursionlimit(5000)` relève cette limite, au prix de plus de mémoire.
- Toute fonction récursive peut s'écrire avec une boucle, et inversement. On choisit la forme la plus lisible : la récursivité convient aux structures elles-mêmes récursives, comme les arbres.

À retenir.

- Une fonction récursive a au moins un cas de base et un appel récursif sur un problème plus petit.
- Chaque appel empile un contexte dans la pile d'appels, dépilé quand l'appel se termine.
- Des appels qui recalculent les mêmes valeurs rendent le calcul exponentiel ; la mémorisation les évite.
- La profondeur de récursivité est limitée à environ 1000 appels en Python.