Récursivité

Langages et programmation

1 Introduction

Une fonction est **récursive** si, lors de son exécution, elle s'appelle elle-même. Pour éviter que la fonction ne s'appelle à l'infini, elle doit comporter une **condition de terminaison** (ou **cas de base**).

Toute fonction récursive a donc deux composantes :

— un ou plusieurs **cas de base** qui renvoient directement un résultat ;

— un ou plusieurs **appels récursifs** qui réduisent le problème à un cas plus simple.

2 Premiers exemples

2.1 Somme des éléments d'une liste

Version itérative :

```
def somme(L):
    s = 0
    for x in L:
        s += x
    return s
```

Version récursive :

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

Cas de base : la liste vide a pour somme 0. Appel récursif : la somme de `L` est le premier élément plus la somme du reste.

2.2 Factorielle

La factorielle de n, notée $n!$, est définie par : $0! = 1$ et $n! = n \times (n-1)!$ pour $n \geq 1$.

Version itérative :

```
def factorielle(n):
    f = 1
    for k in range(1, n + 1):
        f *= k
```

```
    return f
```

Version récursive :

```
def factorielle(n):
    if n == 0:
        return 1
    return n * factorielle(n - 1)
```

2.3 Suite de Fibonacci

La suite de Fibonacci est définie par : $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$.

Version récursive (naïve) :

```
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Cette version a une complexité exponentielle car elle recalcule les mêmes valeurs de nombreuses fois.

2.4 Inversion d'une chaîne

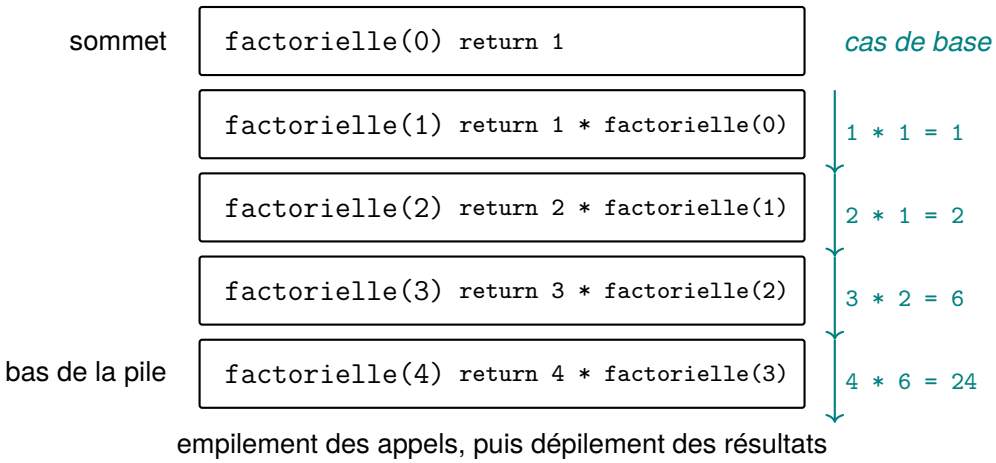
```
def inverse(s):
    if s == "":
        return ""
    return inverse(s[1:]) + s[0]

print(inverse("Python"))  # nohtyP
```

③ La pile d'appels

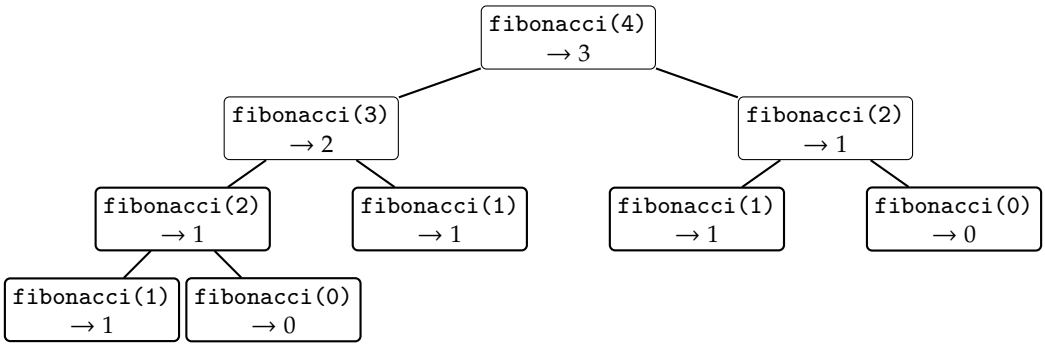
Chaque appel récursif empile un nouveau **cadre d'exécution** dans la **pile d'appels**. Ce cadre contient les paramètres et les variables locales de l'appel en cours.

Déroulons l'exécution de `factorielle(4)` :



À chaque appel, un cadre est empilé. Quand le cas de base est atteint, les cadres sont dépilés un par un et chaque résultat est transmis à l'appelant.

Pour Fibonacci, chaque appel en génère deux. Voici les appels déclenchés par `fibonacci(4)` :



Cette figure s'appelle l'**arbre des appels** : chaque case est un appel, reliée aux appels qu'il déclenche. On y lit que `fibonacci(2)` est calculé deux fois et `fibonacci(1)` trois fois. Plus `n` grandit, plus ces calculs répétés se multiplient : le nombre d'appels croît de façon exponentielle, et `fibonacci(35)` en demande près de trente millions.

La **mémoïsation** évite ces calculs répétés. On range chaque résultat dans un dictionnaire la première fois qu'on le calcule, puis on le relit au lieu de le recalculer :

```
memo = {}

def fibonacci_memo(n):
    if n in memo:                  # déjà calculé : on relit le résultat
        return memo[n]
    if n == 0 or n == 1:
        resultat = n
    else:
        resultat = fibonacci_memo(n - 1) + fibonacci_memo(n - 2)
    memo[n] = resultat             # on range le résultat avant de le renvoyer
    return resultat

print(fibonacci_memo(35))          # 9227465, obtenu immédiatement
```

Chaque valeur n'est alors calculée qu'une seule fois : le nombre d'appels passe d'une croissance exponentielle à une croissance linéaire en n. Le chapitre sur la programmation dynamique généralise cette idée.

④ Limites de la récursivité

4.1 Profondeur de pile

Python limite la profondeur de la pile d'appels à environ 1 000 cadres. Au-delà, il lève une `RecursionError` :

```
factorielle(3000)
# RecursionError: maximum recursion depth exceeded
```

On peut augmenter cette limite avec `sys.setrecursionlimit()`, mais ce n'est pas toujours recommandé pour des raisons de performance et de consommation mémoire.

4.2 Récursivité vs itération

Toute fonction récursive peut être réécrite de manière itérative, et inversement. Le choix dépend du problème : la récursivité est souvent plus naturelle pour les structures récursives (arbres, expressions imbriquées, fractales), tandis que l'itération est plus efficace en mémoire.