

INTRODUCTION À LA NOTION DE COMPLEXITÉ

I. Un premier exemple

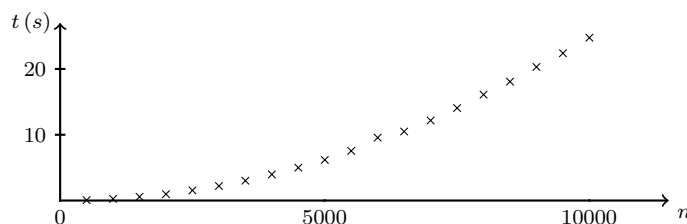
On souhaite écrire une fonction qui prend en argument un entier n et qui renvoie $\sum_{i=1}^n \sum_{j=1}^i j^3$.

Une première méthode consiste à imbriquer deux boucles for :

```
def double_somme_1(n):
    res = 0
    for i in range(1, n+1):
        for j in range(1, i+1):
            res = res + j**3
    return res
```

Avec cette méthode, on calcule plusieurs fois la même quantité. Par exemple, $\sum_{j=1}^{n-1} j^3$ est calculée deux fois : lors du calcul de $\sum_{j=1}^{n-1} j^3$, mais aussi lors de celui de $\sum_{j=1}^n j^3$.

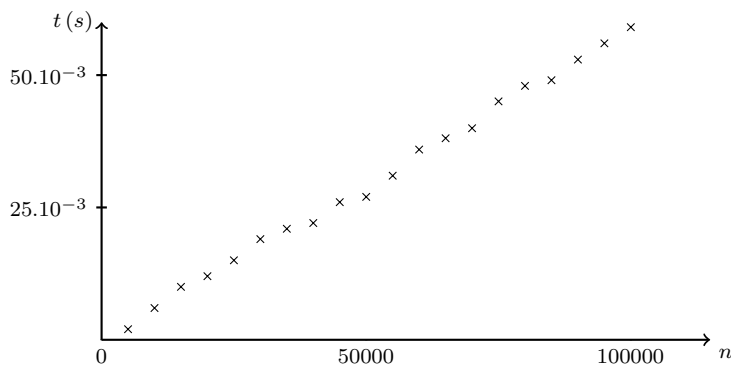
Par conséquent, le temps de calcul de la fonction augmente très rapidement lorsque n augmente.



Pour ne pas refaire plusieurs fois le même calcul, on peut garder en mémoire la valeur courante de $\sum_{j=1}^i j^3$:

```
def double_somme_2(n):
    res = 0
    somme_j = 0 #vaut 1**3 + 2**3 + ... + i**3
    for i in range(1, n+1):
        somme_j = somme_j + i**3
        res = res + somme_j
    return res
```

On peut vérifier expérimentalement que la deuxième fonction est plus rapide que la première (sur une même machine : 5ms pour $n = 10^5$, contre près de 25s avec la fonction précédente).



II. Complexité en temps

1) Notion de complexité et exemple

Plus généralement, lorsqu'il existe plusieurs algorithmes pour traiter un même problème, le temps d'exécution est un critère essentiel. Par conséquent, il est primordial de pouvoir estimer le temps d'exécution relatif de plusieurs algorithmes (même si en réalité ce temps dépend d'autres paramètres).

On va évaluer le temps d'exécution, ou complexité (temporelle) de l'algorithme en comptant le nombre des opérations les plus caractéristiques de l'algorithme ; les opérations choisies doivent s'exécuter en un temps constant.

Comme le temps d'exécution dépend de l'entrée de l'algorithme, on étudiera la complexité en fonction d'un paramètre n mesurant la complexité de l'entrée, le plus souvent la taille de l'entrée ou sa valeur.

Dans l'exemple précédent, on va évaluer le nombre d'additions et le nombre de calculs de cubes pour chacune des deux fonctions :

- Lorsqu'on appelle la fonction `double_somme_1` sur un entier n :
 - Pour chaque itération de la boucle d'indice j , il y a une addition et un cube.
 - Pour une itération de la boucle d'indice i , il y a i itérations de la boucle d'indice j , donc i additions et i cubes ; la première itération comprend donc 1 addition / cube, la deuxième 2 additions / cubes, ..., la $n^{\text{ème}}$ n additions / cubes.

Il y a donc $1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$ additions, de même pour les cubes.

- Dans la fonction `double_somme_2`, il y a n itérations de la boucle, chacune comprenant deux additions et un calcul de cubes. Il y a donc $2n$ additions et n calculs de cubes.

2) Complexité : comportement asymptotique

En réalité, les différentes opérations élémentaires considérées ne demandent pas le même temps de calcul. Nos calculs de complexité ne fournissent qu'une estimation ; ce qui nous intéresse, c'est d'avoir un ordre de grandeur.

Par exemple, il ne serait pas pertinent de dire que la fonction `double_somme_2` a une complexité en $3n$; on se contentera de dire que sa complexité est proportionnelle à n .

Par ailleurs, on s'intéressera uniquement au terme dominant de la complexité ; par exemple, la fonction `double_somme_1` a une complexité proportionnelle à $n^2 + n$, mais pour $n \geq 1$, $n^2 + n \leq 2n^2$. Autrement dit, à partir d'un certain rang, $n^2 + n$ est majoré par le produit de n^2 et d'une constante ; on dira alors que la quantité $n^2 + n$ est « un grand O de n^2 » et on écrira $n^2 + n = O(n^2)$.

D'une manière générale, on décrira la complexité d'un algorithme sous la forme d'un $O(f(n))$ avec f une fonction usuelle, ce qui signifiera qu'à partir d'un certain rang, son coût est majoré par le produit de $f(n)$ et d'une constante.

Dans notre exemple, la fonction `double_somme_1` a une complexité en $O(n^2)$ et la fonction `double_somme_2` a une complexité en $O(n)$; comme n^2 croît bien plus vite que n , la fonction `double_somme_2` est plus efficace.

III. Complexité en mémoire

Une autre ressource utilisée lors de l'exécution d'un algorithme est la mémoire.

Ici, les fonctions `double_somme_1` et `double_somme_2` utilisent une taille mémoire fixée (respectivement 1 et 2 entiers), donc ont une complexité en mémoire en $O(1)$.