Complexité algorithmique et notation Grand O

21/01/2019 | En français

▶ **Sommaire**

Ce billet date de plusieurs années, ses informations peuvent être devenues obsolètes.

Ça devait bien faire depuis 2004 que je n'avais pas remis la tête dans les algorithmes. Manifestement, c'est très important pour les gens formés à la science informatique.

Du coup, retour à l'entrainement pour moi avec en prime mes notes de lecture sur le sujet.

Notation Grand O

La notation Grand O est une métrique permettant de décrire le temps d'exécution des algorithmes.

Cette version de la notation sur laquelle l'industrie informatique s'est mise d'accord est une version *custom* (fusion des Grand Theta et Grand O) de la comparaison asymptotique mathématique.

La complexité algorithmique est le temps d'exécution d'un algorithme mesuré en terme d'accroissement de la taille des données en entrée. On écrit `O(n)` où `n` est une variable qui représente la taille des données en entrée. Il est possible d'avoir plusieurs éléments en entrée, on noterait alors par exemple `O(ab)` ou `O(a + b)` en fonction de la logique de l'algorithme.

On prononce soit :

- de l'ordre de `n` (*order of* `n`)
- Grand O de `n` (*Big O of* `n`)

Ce temps d'exécution peut être représenté sur un graphique dont l'axe des `x` correspond à la taille des données en entrée, et l'axe des `y` au temps.

Voici des exemples courants de différentes complexités :

Big-O	10 éléments en entrée	100 éléments en entrée
`O(1)`	1	1
`O(log n)`	3	7
`O(n)`	10	100
`O(n log n)`	30	700
$O(n^2)$	100	10000
$O(2^n)$	1024	2^{100}
`O(n!)`	3628800	100!

Complexité en temps

Le temps d'exécution d'un algorithme peut être exprimé de 3 façons différentes :

- Dans le meilleur des cas

- Dans le pire des cas
- Dans le cas moyen (ou *expected case*)

Le *meilleur* des cas est rarement discuté et la notation Grand O met le focus sur les cas *moyen* et *pire* qui sont parfois différents, mais souvent identiques.

Complexité en espace

Le temps n'est pas la seule chose qui compte dans un algorithme. Il faut aussi faire l'analyse de l'espace mémoire utilisé. Par exemple, si on a besoin de créer un tableau de taille `n`, on aura besoin de `O(n)` d'espace mémoire.

Un bon algorithme utilise le moins d'espace possible.

Constantes

L'utilisation d'une constante dans un algorithme n'aura d'influence sur son taux de croissance que d'un montant constant, de sorte qu'un algorithme à croissance linéaire conservera une croissance linéaire etc. Pour cette raison, la notation Grand O ne se préoccupe pas des constantes : `O(2n)` devient `O(n)`.

Ça ne veut pas dire que les constantes ne sont pas importantes, simplement qu'elles ne modifient pas fondamentalement la manière dont croît un algorithme.

Terme dominant

La complexité asymptotique est définie par le terme dominant d'une fonction de croissance.

Dit autrement : le terme dominant d'une expression Grand O est celui qui aura le plus d'influence sur la vitesse d'accroissement d'un algorithme.

Je reprends ici un exemple que j'ai trouvé bien fait. Pour une fonction de croissance `f(n) = 2n^2 + 4n + 6`, on peut écrire cette table avec les variations de `n` en entrée :

`n`	$2n^2$	`4n`	`6`	`f(n)`
1	2	4	6	12
10	200	40	6	246
15	450	60	6	516
20	800	80	6	886
25	1250	100	6	1356
30	1800	120	6	1926
35	2450	140	6	2596
40	3200	160	6	3336

On peut facilement voir que le terme qui domine la fonction est $2n^2$. Et comme on laisse tomber les constantes, on peut retirer le `2` et décrire cette fonction avec $O(n^2)$.

Évaluer Grand O, addition ou multiplication ?

Si on se retrouve face à un algorithme de la forme *faire ceci, puis quand c'est terminé faire cela*, il faut **additionner** les temps d'exécution :

```
for i in a:
    print(i)
for j in b:
    print(j)
```

Le temps d'exécution est alors de `O(i + j)`.

Si on se retrouve face à un algorithme de la forme *faire ceci pour chaque fois qu'on fait cela*, alors il faut **multiplier** les temps d'exécution :

```
for i in a:
    for j in b:
        print(i, j)
```

Le temps d'exécution est alors de `O(i * j)`.

Croissance logarithmique `O(log N)`

Un algorithme qui rétrécit le périmètre du problème en divisant continuellement les données par deux pour trouver une solution a toutes les chances d'avoir une croissance logarithmique notée `O(log N)`.

L'exemple typique est le cas d'une *binary search*. Le gros du travail se passe au début, mais la charge s'allège lentement au fur et à mesure de la réduction du périmètre du problème.

Temps d'exécution récursif

Dans le cas d'une fonction récursive, le temps d'exécution ressemblera souvent (mais pas toujours !) à $O(branches^{profondeur})$.

C.f. la méthode récursive pour calculer la suite de Fibonacci.

Temps amorti

Il y a des algorithmes comprenant des opérations coûteuses mais qui se produisent rarement : la vaste majorité des opérations durent `O(1)`, et occasionnellement l'une d'entre elles dure `O(n)`.

Le concept du temps amorti répartit le surcoût des opérations coûteuses sur les autres opérations. Il faut juste être capable de connaître la fréquence des cas les plus coûteux.

Pour aller plus loin

- Big-O notation explained by a self-taught programmer

- CompSci 101 - Big-O Notation
- Introduction à la complexité algorithmique
- Algorithmique avancée (PDF)
- Notes On Big-O Notation (PDF)
- Big Oh of various operations in current CPython
- A Gentle Introduction to Algorithm Complexity Analysis

‹ Avant

L'opérateur modulo

Après ›

Les fermetures en JavaScript

"The Internet? Is that thing still around?"

Autre citation | *Source : Homer Simpson*

marc@marcarea.com