Théorie : complexité d'un algorithme

Par Vincent JARC alias Kaji9 (/membres/kaji9-16325)

Difficulté	Note	Durée
Moyen		45 jours

Mis à jour le vendredi 26 avril 2013

Thématiques
Ada (/informatique/ada/tutoriels), Programmation (/informatique/programmation/tutoriels)

Algorithmes de tri plus rapides (/informati... **Mesures de complexité des algorithmes**

Complexité

Si vous avez dors et déjà testé les programmes ci-dessus sur de grands tableaux, vous avez du vous rendre compte que certains étaient plus rapides que d'autres, plus efficaces. C'est ce que l'on appelle l'**efficacité** d'un algorithme. Pour la mesurer, la quantifier, on s'intéresse davantage à son contraire, la **complexité**. En effet, pour mesurer la complexité d'un algorithme il "suffit" de mesurer la quantité de ressources qu'il exige (mémoire ou temps-processeur).

En effet, le simple fait de déclarer un tableau exige de réquisitionner des emplacements mémoires qui seront rendus inutilisables par d'autres programmes comme votre système d'exploitation. Tester l'égalité entre deux variables va nécessiter de réquisitionner temporairement le processeur, le rendant très brièvement inutilisable pour toute autre chose. Vous comprenez alors que parcourir un tableau de 10 000 cases en testant chacune des cases exigera donc de réquisitionner au moins 10 000 emplacements mémoires et d'interrompre 10 000 fois le processeur pour faire nos tests. Tout cela ralentit les performances de l'ordinateur et peut même, pour des algorithmes mal ficelés, saturer la mémoire entraînant l'erreur **STORAGE_ERROR : EXCEPTION_STACK_OVERFLOW**.

D'où l'intérêt de pouvoir comparer la complexité de différents algorithmes pour ne conserver que les plus efficaces, voire même de prédire cette complexité. Prenons par exemple l'algorithme dont nous parlions précédemment (qui lit un tableau et teste chacune de ses cases) : si le tableau a 100 cases, l'algorithme effectuera 100 tests ; si le tableau

à 5000 cases, l'algorithme effectuera 5000 tests ; si le tableau a

$$n$$

cases, l'algorithme effectuera

$$n$$

tests ! On dit que sa complexité est en

$$O(n)$$

.

L'écriture O

Ca veut dire quoi ce zéro ?

Ce n'est pas un zéro mais la lettre O ! Et

$$O(n)$$

se lit "*grand O de n*". Cette notation mathématique signifie que la complexité de l'algorithme est proportionnelle à

$$n$$

(le nombre d'éléments contenus dans le tableau). Autrement dit, la complexité vaut "grosso-modo"

$$n$$

.

C'est pas "grosso modo égal à n" ! C'est exactement égal à n ! :colere2:

Prenons un autre exemple : un algorithme qui parcourt lui aussi un tableau à

$$n$$

éléments. Pour chaque case, il effectue deux tests : le nombre est-il positif ? Le nombre est-il pair ? Combien va-t-il effectuer de tests ? La réponse est simple : deux fois plus que le premier algorithme, c'est à dire

$$2 \times n$$

. Autrement dit, il est deux fois plus lent. Mais pour le mathématicien ou l'informaticien, ce facteur 2 n'a pas une importance aussi grande que cela, l'algorithme a toujours une complexité proportionnelle au premier, et donc en

$$O(n)$$

.

Mais alors tous les algorithmes ont une complexité en

$$O(n)$$

si c'est comme ça !

Non, bien sûr. Un facteur 2, 3, 20 ... peut être considéré comme négligeable. En fait, tout facteur constant peut être considéré comme négligeable. Reprenons encore notre algorithme : il parcourt toujours un tableau à

$$n$$

éléments, mais à chaque case du tableau, il reparcourt tout le tableau depuis le début pour savoir s'il n'y aurait pas une autre case ayant la même valeur. Combien va-t-il faire de tests ? Pour chaque case, il doit faire

$$n$$

tests et comme il y a

$$n$$

cases, il devra faire en tout

$$n \times n = n^2$$

tests. Le facteur n'est cette fois pas constant ! On dira que cet algorithme a une complexité en

$$O(n^2)$$

et, vous vous en doutez, ce n'est pas terrible du tout. Et pourtant, c'est la complexité de certains de nos algorithmes de tris ! 

Donc, vous devez commencer à comprendre qu'il est préférable d'avoir une complexité en

$$O(n)$$

qu'en

$$O(n^2)$$

. Malheureusement pour nous, en règle général, il est impossible d'atteindre une complexité en

$$O(n)$$

pour un algorithme de tri (sauf cas particulier comme avec un tableau déjà trié, et encore ça dépend des algorithmes ). Un algorithme de tri aura une complexité optimale quand celle-ci sera en

$$O(n \times \ln(n))$$

.

Quelques fonctions mathématiques

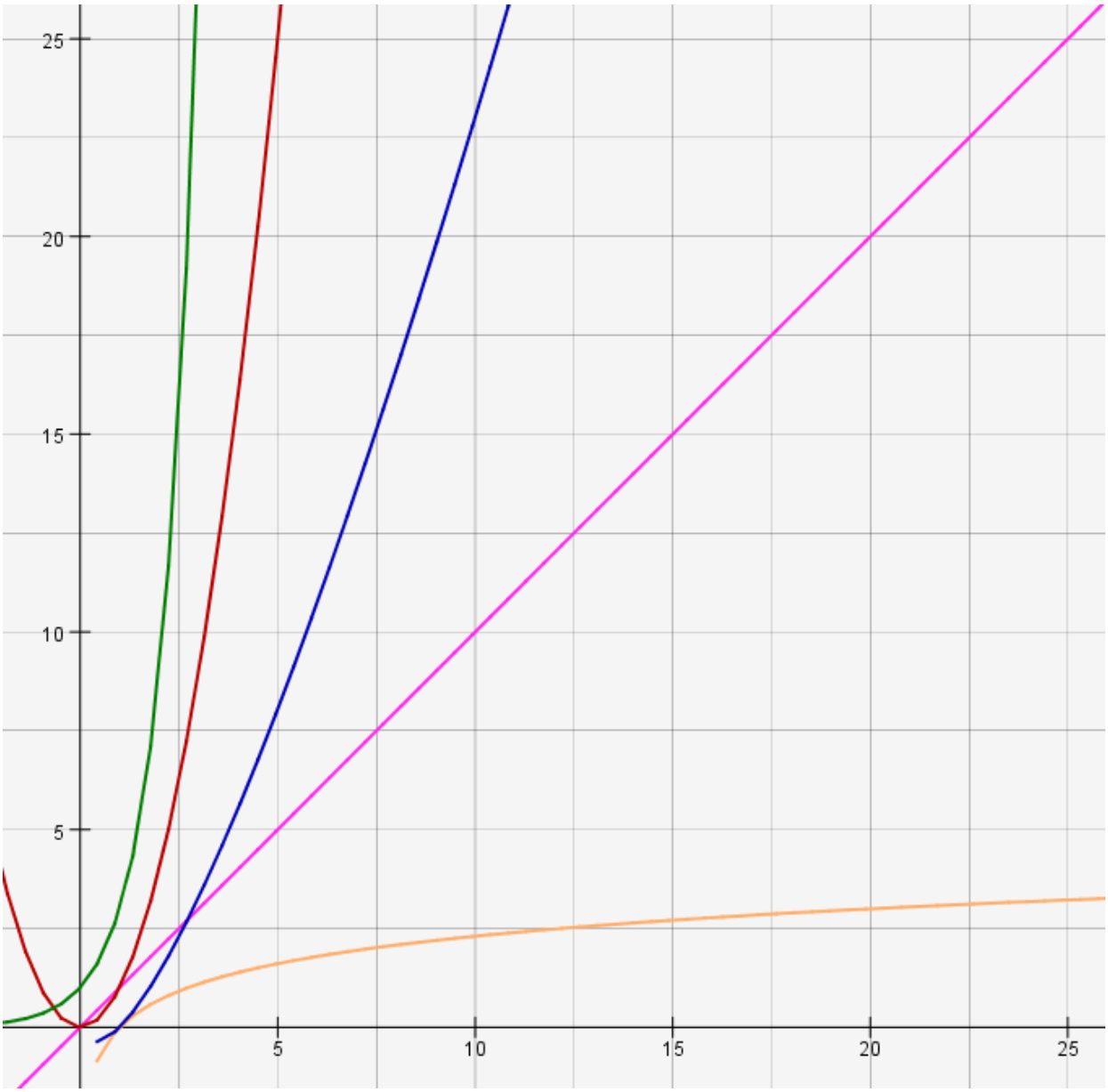
Qu'est-ce que c'est encore que cette notation

$$\ln(n)$$

?

C'est une fonction mathématique que l'on appelle le **logarithme népérien**. Les **logarithmes** sont les fonctions mathématiques qui ont la croissance la plus faible (en fait elles croissent très très vite au début puis ralentissent par la suite). Inversement, les fonctions **exponentielles** sont celles qui ont la croissance la plus rapide (très très lentes au début puis de plus en plus rapide).

Mon but ici n'est pas de faire de vous des mathématiciens, encore moins d'expliquer en détail les notions de fonction ou de "grand O", mais pour que vous ayez une idée de ces différentes fonctions, de leur croissance et surtout des complexités associées, je vous en propose quelques-unes sur le graphique ci-dessous :



Légende :
Complexité en

$$O(e^n)$$

(e : exponentielle)
Complexité en

$$O(n^2)$$

Complexité en

$$O(n \times \log(n))$$

(log : logarithme)
Complexité en

$$O(n)$$

Complexité en

$$O(\log(n))$$

Plus l'on va vers la droite, plus le nombre d'éléments à trier, tester, modifier ... augmente. Plus l'on va vers le haut, plus le nombre d'opérations effectuées augmente et donc plus la quantité de mémoire ou le temps processeur nécessaire augmente. On voit ainsi qu'une complexité en

$$O(\log(n))$$

ou

$$O(\ln(n))$$

, c'est peu ou prou pareil, serait parfaite (on parle de complexité logarithmique) : elle augmente beaucoup moins vite que le nombre d'éléments à trier ! Le rêve. Sauf que ça n'existe pas pour nos algorithmes de tri, donc oubliez-la.

Une complexité en

$$O(n)$$

serait donc idéale (on parle de complexité linéaire). C'est possible dans des cas très particuliers : par exemple l'algorithme de tri par insertion peut atteindre une complexité en O(n) lorsque le tableau est déjà trié ou "presque" trié, alors que les algorithmes de tri rapide ou fusion n'en sont pas capables. Mais dans la vie courante, il est rare que l'on demande de trier des tableaux déjà triés.

On comprend donc qu'atteindre une complexité en

$$O(n \times \ln(n))$$

est déjà satisfaisant (on parle de complexité linéarithmique), on dit alors que la **complexité est asymptotiquement optimale**, bref, on ne peut pas vraiment faire mieux dans la majorité des cas.

Une complexité en

$$O(n^2)$$

, dite complexité quadratique, est courante pour des algorithmes de tri simples (voire simplistes), mais très mauvaise. Si au départ elle ne diffère pas beaucoup d'une complexité linéaire (en

$$O(n)$$

), elle finit par augmenter très vite à mesure que le nombre d'éléments à trier augmente. Donc les algorithmes de tri de complexité quadratique seront réservés pour de petits tableaux.

Enfin, la complexité exponentielle (en

$$O(e^n)$$

) est la pire chose qui puisse vous arriver. La complexité augmente à vite grand V, de façon exponentielle (logique me direz-vous). N'ajoutez qu'un seul élément et vous aurez beaucoup plus d'opérations à effectuer. Pour un algorithme de tri, c'est le signe que vous coder comme un cochon. Attention, les jugements émis ici ne concernent que les algorithmes de tri. Pour certains problèmes, vous serez heureux d'avoir une complexité en seulement

$$O(n^2)$$

.

Algorithmes de tri plus rapides (/informati... **Mesures de complexité des algorithmes**