

Complexité d'un algorithme

La notion de complexité d'un algorithme va rendre compte de l'efficacité de cet algorithme. Pour un même problème, par exemple trier un tableau, il existe plusieurs algorithmes, certains algorithmes sont plus efficaces que d'autres (par exemple un algorithme A mettra moins de temps qu'un algorithme B pour résoudre exactement le même problème, sur la même machine). Il existe 2 types de complexité : une complexité en temps et une complexité en mémoire. Nous nous intéresserons ici uniquement à la complexité en temps. La complexité en temps est directement liée au nombre d'opérations élémentaires qui doivent être exécutées afin de résoudre un problème donné. L'évaluation de ce nombre d'opérations élémentaires n'est pas toujours facile, on rencontre souvent des cas litigieux.

Prenons un exemple avec l'algorithme ci-dessous qui permet de déterminer si la valeur x est présente dans le tableau t :

Var t : tableau d'entiers

Var x, i : entier

Var tr : booléen

DEBUT

tr ← FAUX

i ← 1

tant que $i \leq \text{longueur}(t)$ **et que** $tr == \text{FAUX}$:

si $t[i] == x$:

tr ← VRAI

fin si

i ← $i + 1$

fin tant que

renvoyer la valeur de tr

FIN

Si vous avez du mal à comprendre cet algorithme, faites le « tourner à la main » avec avec $t=[5,8,15,53]$ et $x=12$, puis avec $x=15$

Il y a 2 cas à traiter :

- L'entier recherché est bien présent dans le tableau, il se trouve à la position d'index j

- L'entier recherché n'est pas présent dans le tableau.

On s'intéressera toujours au **pire des cas**, c'est-à-dire ici lorsque x n'est pas dans le tableau (*ok, il y a pire : lorsque x est en dernière position, mais on va l'ignorer, vous comprendrez bientôt pourquoi ça n'a pas d'importance*).

Regardons le nombre de fois que chaque opération élémentaire se fait :

DEBUT

1 fois **tr** ← FAUX

1 fois **i** ← 0

$n+1$ fois **tant que** $i \leq \text{longueur}(t) - 1$ **et que** $tr == \text{FAUX}$:

n fois **si** $t[i] == x$:

0 fois **tr** ← VRAI

fin si

n fois **i** ← $i + 1$

fin tant que

1 fois **renvoyer la valeur de tr**

FIN

Au total nous avons :

$1 + 1 + n + 1 + n + 0 + n + 1$

= **$3n + 4$ opérations élémentaires**, n étant le nombre de variables présentes dans le tableau.

Nous voyons que la complexité dépend de la taille du tableau, plus le tableau est grand et plus le nombre d'opérations élémentaires à effectuer est important.

Pour effectuer des comparaisons entre plusieurs algorithmes, nous allons raisonner sur des tableaux de grande taille, car plus les tableaux sont grands et plus les différences entre les algorithmes seront flagrantes.

Pour comparer des algorithmes, nous allons uniquement nous intéresser à ce que l'on appelle "l'ordre de grandeur asymptotique". Cet "ordre de grandeur asymptotique" concerne les cas où l'on prend n très très grand. On note cet "ordre de grandeur asymptotique" avec un O majuscule. Pour le cas qui nous intéresse, nous aurons :

$3n+4 = O(n)$

Au final on dira donc que la complexité de notre algorithme "x est-il présent dans le tableau t ?" est $O(n)$

Comment obtient-on cette notation O à partir du nombre d'opérations élémentaires ?

En supprimant avec violence la constante et le coefficient ! $3n+4$, c'est proche de n (si si).

Ici nous avons simplement supprimé la constante (4) et le coefficient devant le n (c'est-à-dire 3), il reste donc uniquement n d'où le $3n+4 = O(n)$.

Dans le cas où nous avons un polynôme de degrés quelconque, par exemple pour $6n^2+3n+10$, il suffit de :

- **supprimer la constante**
- **garder uniquement le n qui possède l'exposant le plus grand**
- **supprimer le coefficient devant le n**

Par exemple, $6n^2+3n+10 = O(n^2)$ car " $6n^2+3n+10$ est dominée asymptotiquement par n^2 "

On dira aussi que la complexité « est en n^2 »

Exercices :

1/ Écrivez un algorithme permettant de trouver le plus grand entier présent dans un tableau. Vous ferez "tourner à la main" votre algorithme en utilisant le tableau $t = [3,5,1,8,4,2]$. Vous déterminerez ensuite la complexité de votre algorithme.

2/ Écrivez un algorithme permettant de calculer la moyenne de tous les entiers présents dans un tableau. Vous ferez "tourner à la main" votre algorithme en utilisant le tableau $t = [3,5,1,8,4,2]$. Vous déterminerez ensuite la complexité de votre algorithme.

Les complexités du pire au mieux

$O(e^n)$ (e : exponentielle)

$O(n^2)$

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

$O(n)$

$O(\log(n))$

Comparatif des complexités en courbes :

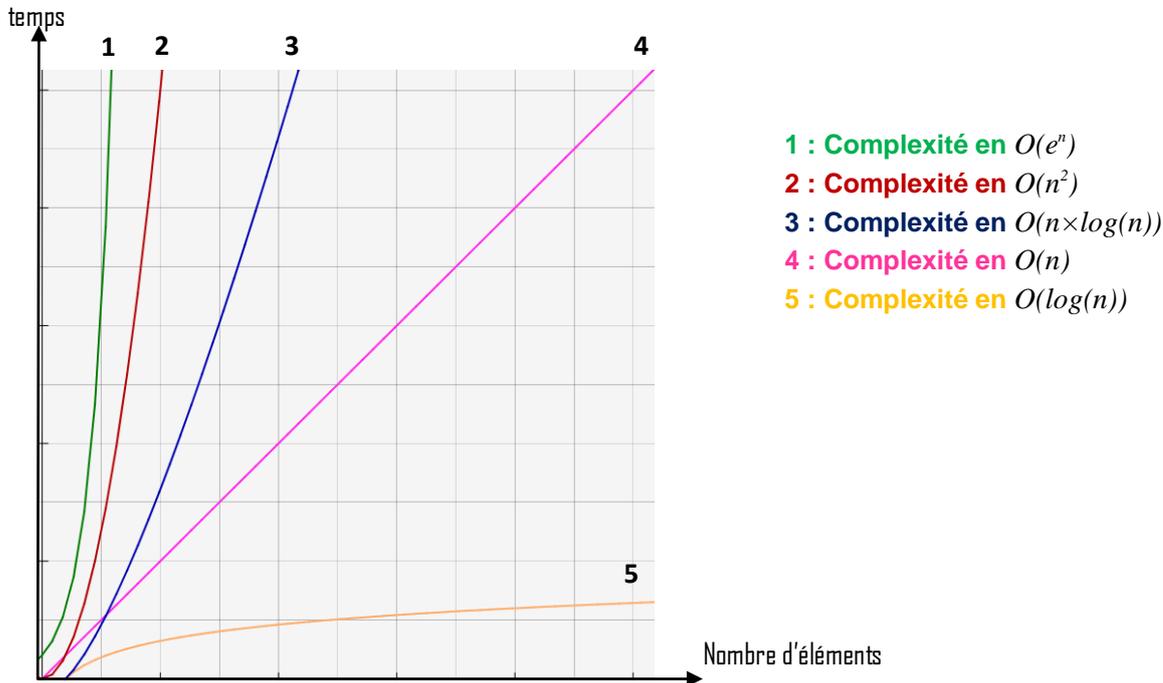


Tableau comparatif (avec un temps d'exécution de 10ns/instruction) :

Notation	Type de complexité	n = 5	n = 10	n = 20	n = 50	n = 250	n = 1 000	n = 10 000	n = 1 000 000
$\Theta(1)$	complexité constante	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns
$\Theta(\log(n))$	complexité logarithmique	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns	60 ns
$\Theta(n)$	complexité linéaire	50 ns	100 ns	200 ns	500 ns	2.5 μ s	10 μ s	100 μ s	10 ms
$\Theta(n \log(n))$	complexité linéarithmique	40 ns	100 ns	260 ns	850 ns	6 μ s	30 μ s	400 μ s	60 ms
$\Theta(n^2)$	complexité quadratique (polynomiale)	250 ns	1 μ s	4 μ s	25 μ s	625 μ s	10 ms	1 s	2.8 heures
$\Theta(n^3)$	complexité cubique (polynomiale)	1.25 μ s	10 μ s	80 ms	1.25 ms	156 ms	10 s	2.7 heures	316 ans
$\Theta(n^{\log(n)})$		30 ns	100 ns	492 ns	7 μ s	5 ms	10 s	3.2 ans	10^{20} ans
$\Theta(e^n)$	complexité exponentielle	320 ns	10 μ s	10 ms	130 jours	10^{59} ans
$\Theta(n!)$	complexité factorielle	1.2 μ s	36 ms	770 ans	10^{48} ans