

Mise à Niveau Complexité

Support de cours M1 Algorithmique Avancée (Année 2019-2020)

Florence Bannay¹

Compétences à consolider à partir de ce cours :

Évaluer la complexité d'un algorithme :

- Expliquer l'intérêt d'une analyse de complexité
- Définir la complexité spatiale ou temporelle
- Différencier la complexité en meilleur cas, en moyenne, en pire cas
- Exprimer une fonction f grâce aux notations O , Ω , Θ , o
- Évaluer la complexité d'un algorithme itératif
- Évaluer la complexité d'un algorithme récursif
- Lister les différentes fonctions caractérisant des complexités courantes, savoir les tracer, et pour chacune citer un exemple d'algorithme ayant cette complexité

I Introduction

L'analyse de la complexité des algorithmes est un aspect fondamental et nécessaire de toute activité de développement informatique. Cette analyse permet d'estimer le temps de calcul et l'espace mémoire nécessaire pour le traitement d'un ensemble de données².

Ces estimations servent à :

- respecter des contraintes fortes de temps d'exécution (systèmes temps-réel) et contraintes (plus faibles) d'utilisabilité (dans le domaine de la santé, de la robotique ou des communications et dans la vie réelle).
- minimiser le nombre de composants électroniques nécessaires à la réalisation de l'architecture d'exécution (systèmes embarqués).
- évaluer la consommation énergétique d'un système (systèmes embarqués, informatique verte (green computing))

II Définitions

La complexité pratique est une mesure précise des temps de calcul et occupations mémoire pour un **modèle de machine donné**.

La complexité théorique est un **ordre de grandeur** de ces coûts, exprimés de manière la plus **indépendante** possible des conditions pratiques d'exécution : nombre d'instructions exécutées (complexité temporelle) et l'espace mémoire occupé par le nombre d'informations supplémentaires (aux données entrées) à stocker (complexité spatiale).

Définition 1 (Complexité temporelle et spatiale d'un algorithme A). *Ce sont des fonctions dépendant de la taille des données n traitées par l'algorithme A . Soit D_n l'ensemble des données possibles de taille n . Soit d une donnée de D_n , on note*

- $T_A(n)$: le nombre d'instructions exécutées pendant l'exécution de $A(d)$.
- $S_A(n)$ l'espace mémoire nécessaire à l'exécution de $A(d)$ (sans compter la taille pour stocker d !).

1. Les principaux documents qui ont servi à réaliser ce cours sont les livres [?, ?, ?, ?, ?, ?], les supports de cours [?, ?, ?] et l'encyclopédie en ligne wikipedia [?].

2. Notons que l'espace mémoire étant limité, l'impératif sur l'espace, en pratique, est plus fort que sur le temps : on peut se permettre d'attendre la fin d'un algorithme mais on ne peut pas ajouter de la mémoire en cours d'exécution.

Exemple 1. *Maximum d'un tableau*

Function GETMAXTAB(T, n)
Input: $n = \text{taille de } T \geq 1$.
Output: \max de T .
$M \leftarrow T[1]$
for $i = 2$ to n do
if $T[i] > M$ then $M \leftarrow T[i]$

1 accès + 1 affectation, addition de deux fonctions en $\Theta(1)$ donc $\Theta(1)$. puis une boucle de $n-1$ étapes dans laquelle on fait une comparaison, deux accès et une affectation $(n-1) \times \Theta(1)$ donc algorithme en $\Theta(n)$.

On peut s'intéresser à la complexité en pire cas, dans le meilleur des cas et en moyenne.

Remarque 1. Pour pouvoir calculer T_{moy} et S_{moy} , il est nécessaire de connaître la loi de distribution des données (densité de probabilité), ce qui n'est pas toujours possible ou très difficile à estimer (par exemple, en traitement d'images ou du texte). $[T_{\min}, T_{\max}]$ et $[S_{\min}, S_{\max}]$ définissent les bornes de complexité pour un algorithme.

Complexité asymptotique

Pour comparer des algorithmes, on ne s'intéresse qu'à leurs comportements pour n grand. Ce comportement permet d'évaluer la notion de passage à l'échelle d'un algorithme. On cherche donc une mesure de complexité qui soit indépendante de tout langage de programmation et de la puissance de calcul/stockage de la machine.

- Les constantes n'importent pas
- On cherche un ordre de grandeur

Définition 2 (Complexité asymptotique et Notations de Landau). La complexité est dite asymptotique lorsqu'elle est formulée au moyen des notations O , Ω , Θ ou o . Ces notations permettent d'exprimer des ordres de grandeurs de fonctions positives sur n pour n suffisamment grand tels que : Soit $g : \mathbb{N} \mapsto \mathbb{R}^+$ une fonction positive :

Grand O : $O(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c > 0 \text{ et } \exists n_0 \in \mathbb{N} \text{ t.q. } \forall n \geq n_0, f(n) \leq c.g(n)\}$.

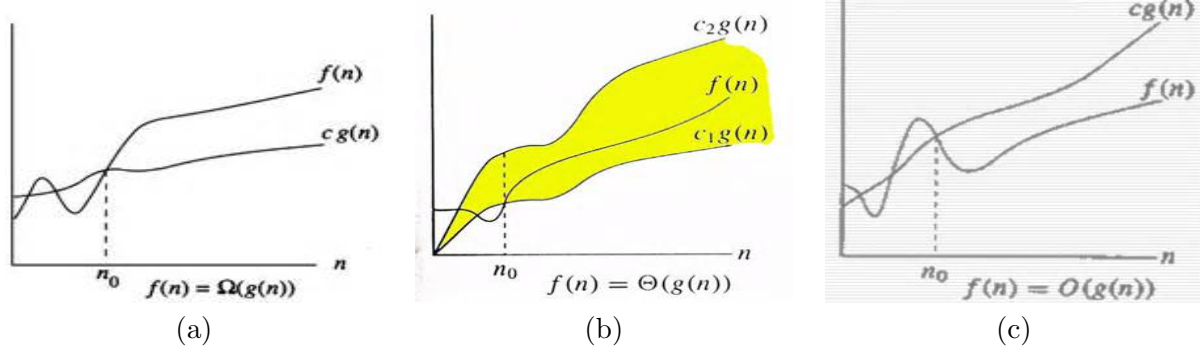
Quand $f \in O(g)$, on dit que g est une borne asymptotique supérieure de f (ou f dominée asymptotiquement par g).

Grand Oméga : $\Omega(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c > 0 \text{ et } \exists n_0 \in \mathbb{N} \text{ t.q. } \forall n \geq n_0, f(n) \geq c.g(n)\}$.

Quand $f \in \Omega(g)$, on dit que g est une borne asymptotique inférieure à f .

Grand Thêta : $\Theta(g) = O(g) \cap \Omega(g)$

Petit o : $o(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^{*+} \mid \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0\}$. On dit que f est négligeable devant g .



Propriété 1. Si on considère la relation R_X entre f et $g : f \in X(g)$

- R_O et R_Ω sont des pré-ordres (relations réflexives et transitives, assimilable à \leq et \geq respectivement)
- R_Θ est une relation d'équivalence (relation réflexive, transitive et symétrique)

Propriété 2. Si la fonction f peut s'écrire comme une somme finie d'autres fonction, alors celle qui croît le plus vite détermine l'ordre de $f(n)$.

En particulier, si une fonction peut être bornée par un polynôme en n alors quand n tend vers l'infini, on peut négliger les termes de degré moindre dans le polynôme.

On peut ignorer toutes les puissances de n à l'intérieur d'un logarithme. De façon similaire, les log avec différentes bases sont équivalents. Par contre les exponentiels avec différentes bases ne sont pas du même ordre. Par exemple, $3^n \notin O(2^n)$.

Exemple 2.

$n \in O(n)$	$n \in \Omega(n)$	$n \in \Theta(n)$
$7 + 1/n \in O(1)$	$7 + 1/n \in \Omega(1)$	$7 + 1/n \in \Theta(1)$
$\log_2(n) \in O(\log n)$	$\log_2(n) \in \Omega(\log n)$	$\log_2(n) \in \Theta(\log n)$
$n + \ln(n) \in O(n)$	$n + \ln(n) \in \Omega(n)$	$n + \ln(n) \in \Theta(n)$
$n^2 + 3n \in O(n^3)$	$n^2 + 3n \notin \Omega(n^3)$	$n^2 + 3n \notin \Theta(n^3)$ (mais $\Theta(n^2)$)

La première ligne peut se prouver en utilisant le fait que les relations R_O , R_Ω et R_Θ sont réflexives.

Pour analyser la fonction $7+1/n$ de la deuxième ligne, on se dit que 7 (la fonction constante) « croît plus rapidement » que $1/n$ (qui décroît). Cela revient donc à essayer de prouver que $7 + 1/n \in O(7) = O(1)$. Donc on essaye de prouver que $7 + 1/n$ est inférieur à une constante quand n est assez grand. On a $1/n \leq 1$ si $n > 0$ donc $7 + 1/n \leq 8$. Donc $7 + 1/n \in O(1)$ (avec $c = 8$, $n_0 = 1$ dans la définition). De même on a $1/n \geq 0$ si $n > 0$ donc $1/n + 7 \geq 7$ ainsi $7 + 1/n \in \Omega(1)$ (avec $c = 7$, $n_0 = 1$ dans la définition). Ainsi $7 + 1/n \in \Theta(1)$. Notons qu'il est impossible de prouver que $\forall n \geq n_0, 7 + 1/n \leq c \times 1/n$ puisque $\lim_{n \rightarrow \infty} 1/n = 0$.

Propriété 3.

1. Pour toutes fonctions f et g à valeurs strictement positives, $f \sim g$ et $\lim_{n \rightarrow +\infty} f(n) \neq 1$ implique $\log f \sim \log g$
 2. Pour toutes fonctions f , g et h , $f \sim g$ et $f \in \Theta(h)$ implique $g \in \Theta(h)$.
- avec $f \sim g$, lu « f est équivalente à g à l'infini », ssi $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 1$.

III Complexité des algorithmes définis par récurrence

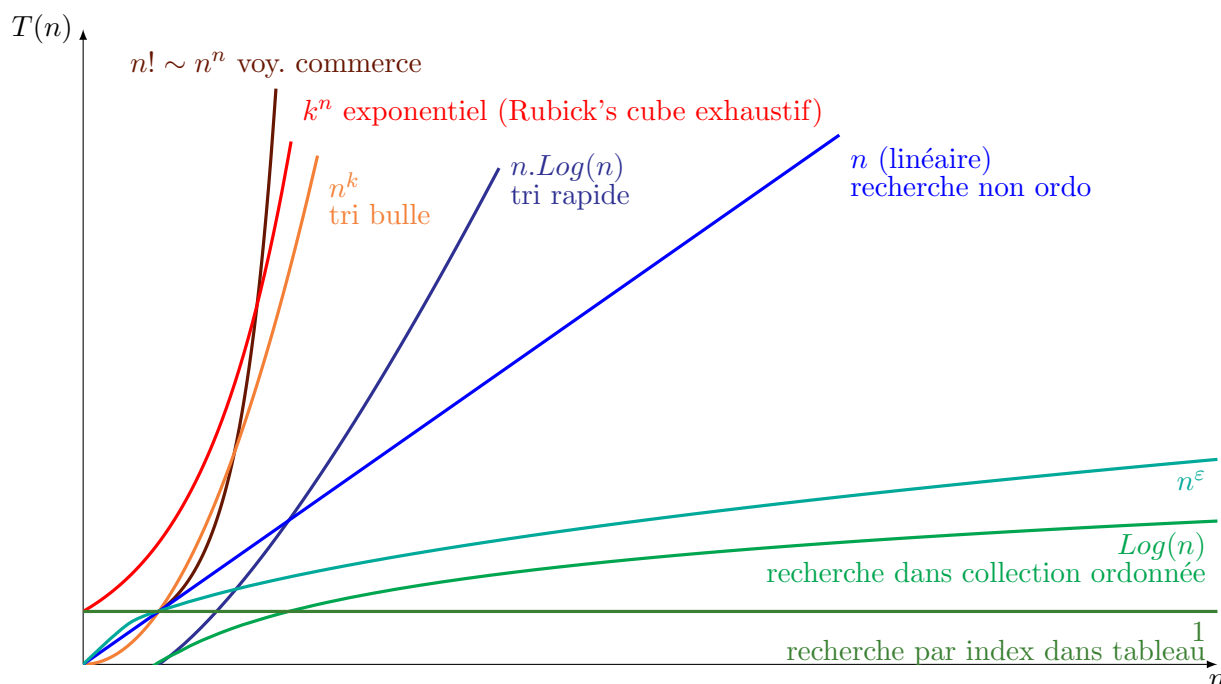
1 Complexité spatiale

La particularité des algorithmes définis par récurrence, ou plus généralement des algorithmes qui contiennent des appels à des fonctions, est qu'ils doivent stocker l'environnement d'exécution dans une pile. L'environnement d'exécution contient :

- la valeur des paramètres de l'algorithme
- la valeur de ses variables locales
- l'adresse de retour (adresse de l'instruction qui sera exécutée quand l'appel sera fini)

Chaque fois qu'un appel à un sous-programme est fait, le contexte d'exécution est empilé.

L'analyse de la complexité spatiale d'une fonction récursive doit donc calculer la taille maximale de la pile d'appel au cours de l'exécution de l'algorithme, ainsi que la taille de chaque contexte d'exécution.

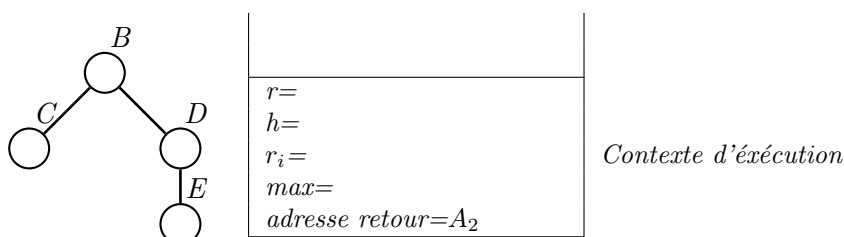
FIGURE 1 – Complexités courantes, Avec $0 < \varepsilon < 1$ et $k > 1$.

Exemple 3. Complexité spatiale de l'algorithme qui calcule la hauteur d'un arbre :

Function HAUTEUR(r)	
Input: r =racine d'un arbre	
Output: hauteur de l'arbre	
if r est une feuille then return (0)	
else	
$max \leftarrow 0$	
for chaque enfant r_i de r do	
$h \leftarrow \text{HAUTEUR}(r_i)$	A_1
if $h > max$ then $max \leftarrow h$	
return ($max + 1$)	

Function MAIN	
$B \leftarrow \text{CREERARBRE}()$	
$h \leftarrow \text{HAUTEUR}(B)$	A_2

Avec l'arbre ci-dessous, on a la pile d'exécution suivante après le premier appel à HAUTEUR depuis le MAIN :



Après la première exécution de Hauteur la pile devient :

$r=C$ $h=$ $r_i=$ $max=$ $adresse\ retour=A_1$	<i>Quand l'algorithme arrive à E il y a trois contextes d'exécutions dans la pile. La complexité spatiale de l'algorithme est donnée par la taille de la pile. Le nombre de contextes d'exécutions est le nombre maximum d'appels successifs, c'est donc la hauteur de l'arbre + 1. Chaque contexte occupe une place constante cte =taille de r, h, r_i et adresse de retour. Donc la complexité spatiale est en $\Theta(\text{ hauteur de l'arbre} +1) \times cte = \Theta(\text{hauteur de l'arbre})$</i>
$r=B$ $h=$ $r_i=C$ $max=0$ $adresse\ retour=A_2$	

2 Complexité temporelle

2.1 Récurrence linéaire

Lorsque l'on analyse des algorithmes définis par récurrence, et quelle que soit la manière de les implanter - récursivité, itératif avec ou sans mémorisation d'état, l'analyse de la complexité de ces algorithmes repose sur la définition d'une suite $(u_n)_{n \geq 0}$ des temps d'exécution de l'algorithme en fonction de la taille n des données. Cette définition correspond à une équation récurrente linéaire d'ordre k :

Définition 3. Une équation récurrente linéaire à coefficients constants d'ordre k est une équation de la forme :

$$\begin{cases} u_i = c_i \quad (0 \leq i \leq k-1) & \text{Conditions initiales - Cas simples} \\ u_n = \sum_{i=1}^k a_i \cdot u_{n-i} + g(n) \end{cases}$$

Une équation est homogène si le second membre $g(n) = 0$ pour tout n .

Exemple 4. Fibonacci : $u_0 = 1, u_1 = 1$ et $u_n = u_{n-1} + u_{n-2}$: équation récurrente homogène d'ordre $2 = k, c_0 = c_1 = 1, a_1 = a_2 = 1, g(n) = 0$.

Dans l'équation récurrente :

k : profondeur de la récurrence (i.e. le calcul de n dépend des calculs des niveaux $n-1$ à $n-k$)

c_i : temps de traitement des cas simples,

a_i : nombre d'appel récursif effectué sur des données de taille $n-i$

$g(n)$: temps nécessaire à la combinaison des appels récursifs.

Exemple 5. Maximum d'un tableau (appel GETMAXTABREC($T, n-1, T[n]$)),

Function GETMAXTABREC(T, k, M)
<p>Require: $k \leq n = \text{taille de } T, n \geq 1.$</p> <p>Ensure: renvoie le max de T.</p> <pre> if $k = 0$ then return (M) else if $T[k] > M$ then GETMAXTABREC($T, k - 1, T[k]$) else GETMAXTABREC($T, k - 1, M$) </pre>

comparaison + (1 branchement ou (comparaison + (accès + branchement + addition) ou (branchement + addition))))

Soit $u_n = T(\text{getMaxTabRec}(n))$.

cas simple $u_1 = 2$ (comparaison + branchement).

cas général $u_n = u_{n-1} + 5$

Pour trouver l'expression du temps de calcul, et par la suite de la classe de complexité, il faut résoudre la récurrence = trouver l'expression de u_n en fonction de n .

Exemple 6. Pour GETMAXTABREC, on peut trouver la complexité en écrivant $T(n) = T(n-1) + 5$ (suite arithmétique de raison 5 donc $T(n) = 5(n-1) + T(1)$, on retrouve en faisant $T(n) = T(n-2) + 10 = T(n-3) + 15 = T(n-k) + 5k$ on s'arrête quand $n-k = 1$ i.e. $k = n-1$ donc $T(n) = T(1) + 5n - 5 = 5n - 3$) d'où complexité en $\Theta(n)$.

En général, le calcul revient à résoudre des équations polynomiales. Il existe un théorème qui permet d'exprimer toute suite u_n basée sur une équation linéaire d'ordre k en fonction de n (en utilisant les racines de certains polynômes qu'on définit à partir de la suite u_n).

2.2 Récurrence diviser pour régner

Un cas particulier de récurrence concerne les algorithmes de type diviser pour régner. Ceux-ci procède en trois étapes :

Diviser : on sépare le problème en a sous-problèmes de taille n/b soit $D(n)$ le temps pour faire cette division,

Régner : on résout chaque problème, soit $a.T(n/b)$ le temps pour résoudre les a sous-problèmes

Combinaison : on combine les résultats soit $C(n)$ le temps pour construire la solution finale de taille n .


La relation de récurrence s'écrit donc $T(n) = a.T(n/b) + D(n) + C(n)$ et pour ce genre de récurrence, le théorème suivant s'applique :

Théorème 1 (Master Theorem : Résolution de récurrences « diviser pour régner »).

Soient $a \geq 1$ et $b > 1$ deux constantes, $f(n)$ une fonction et $T(n)$ définie pour $n \geq 0$ par récurrence de la façon suivante :

$$T(n) = aT(n/b) + f(n)$$

$T(n)$ à les bornes asymptotiques suivantes :

- Si $f(n) \in O(n^{\log_b a - c})$ pour une constante $c > 0$, alors $T(n) \in \Theta(n^{\log_b a})$  "O" puis "Θ"
- Si $f(n) \in \Theta(n^{\log_b a})$, alors $T(n) \in \Theta(n^{\log_b a} \log n)$
- Si $f(n) \in \Omega(n^{\log_b a + c})$ pour une constante $c > 0$, et si $af(n/b) \leq d.f(n)$ pour une constante $d < 1$ et n suffisamment grand, alors $T(n) \in \Theta(f(n))$

Remarque 1.

- Le remplacement des termes $T(n/b)$ par³ $T(\lfloor n/b \rfloor)$ n'affecte pas le résultat asymptotique.
- Le théorème ne couvre pas toutes les possibilités pour $f(n)$. Par exemple, on ne peut pas appliquer ce théorème si $f(n)$ est plus petite que $n^{\log_b(a)}$ mais pas polynomialement (trou entre cas 1 et 2).

Exemple 7. Maximum d'un tableau (appel $\text{GETMAXTABDC}(T, 1, n)$,

Function $\text{GETMAXTABDC}(T, i, j)$	
Require: $1 \leq i \leq j \leq n = \text{taille de } T, n \geq 1.$	— division : temps constant
Ensure: renvoie le max de T .	— règne : $2 \times T(n/2)$
<pre> if $i = j$ then return $T[i]$ else $M1 \leftarrow \text{GETMAXTABDC}(T, i, \lfloor (i+j)/2 \rfloor)$ $M2 \leftarrow \text{GETMAXTABDC}(T, \lfloor (i+j)/2 \rfloor + 1, j)$ if $M1 > M2$ then return $M1$ else return $M2$ </pre>	— comb. : temps constant
	Donc $T(\text{getMaxTabDC}(n)) = 2T(n/2) + cte$
	$a = b = 2$ donc $\log_b a = 1$, $f(n) = cte$ donc en $O(1)$. Donc $c = 1$
	dans la première formule du Master Theorem et $T(n)$ en $\Theta(n)$.

IV Complexité et temps d'exécution

Le tableau donne des temps de calcul, en fonction de la puissance du processeur en flops⁴ et de la complexité temporelle asymptotique de l'algorithme pour des données de taille $n = 10^6$.

flops	$\log n$	n	n^2	2^n
10^6	0.013ms	1s	278h	10000 ans
10^9	0.013 μ s	1ms	$\frac{1}{4}$ h	10 ans
10^{12}	0.013ns	1 μ s	1s	1 semaine

Loi de Moore. A coût constant, la rapidité des processeurs double tous les 18 mois. Les capacités de stockage suivent la même loi.

Constat. Le volume de données stockées dans les systèmes d'information augmente annuellement de façon exponentielle.

Conclusion. Mieux vaut travailler à réduire la complexité des algorithmes qu'attendre des décennies qu'un processeur surpuissant soit inventé ...

Ouverture : <https://lejournel.cnrs.fr/articles/la-multiplication-reinventee> (mars 2019)

3. Pour ne pas confondre la notion des tableaux et la partie entière, on utilisera $\lfloor \rfloor$ pour les indices des tableaux et $\lfloor \rfloor$ pour la partie entière

4. flops : floating operations per seconds.