Calculabilité et Complexité (11X008) - Printemps 2024

6. Introduction à la complexité, notion d'ordre de grandeur

Enseignant: Arnaud Casteigts *Exercices: A. Berger, M. De Francesco, L. Heiniger*

Dans ce cours, nous commençons par donner une intuition du principe de complexité algorithmique. Ensuite, nous présentons les concepts d'ordre de grandeur asymptotiques. Finalement, nous terminons avec les notions de base de la complexité en temps et en espace.

6.1 Complexité

Jusqu'ici, nous avons vu dans ce cours les grands concepts de la calculabilité. Cela nous a permis de déterminer, pour un grand nombre de langages, s'ils étaient calculables ou non, et dans quelle mesure (sont-ils décidables, reconnaissables ?).

Mais cela nous informe "seulement" (même si c'est déjà beaucoup) de la calculabilité ou non du problème, d'un point de vue théorique : cela ne nous donne aucune information sur les ressources nécessaires à la résolution du problème.

Dans la pratique, nous ne sommes pas réellement capables de résoudre tout ce qui est calculable, car nous sommes limités principalement par deux ressources : le temps et l'espace.

Supposons que l'on découvre un algorithme pour calculer, avec certitude, sans aucune erreur, la météo de tout le mois suivant, seulement à partir de la météo observée aujourd'hui. Un tel algorithme parait alors être une excellente nouvelle : plus de risque d'être surpris par une pluie qu'on attendait pas, ou par un coup de froid le soir alors que l'on est sorti de chez soi en t-shirt le matin.

Mais supposons que cet algorithme prenne un temps immensément grand, par exemple dix ans, avant de donner une réponse. Est-il réellement utile ? Pouvons-nous réellement nous en servir d'un point de vue pratique ? Malheureusement non.

De la même manière, si cet algorithme peut répondre très vite, mais nécessite 10^{100} Teraoctets de mémoire, alors il n'est pas vraiment utilisable non plus.

Nous pouvons alors nous poser les questions suivantes :

- Quel est le temps nécessaire pour que l'algorithme réponde ?
- Quel est l'espace nécessaire à son exécution ?

- Ces algorithmes sont-ils utilisables en pratique ?
- Pouvons-nous comparer leur complexité ?
- Existe-il un algorithme efficace pour ce problème ?

C'est grâce à la théorie de la complexité que nous allons pouvoir répondre à ces questions.

6.2 Ordres de grandeur asymptotiques

Avant de s'intéresser à la complexité, faisons un léger détour par quelques notations mathématiques.

Soit les deux fonctions suivantes :

- $f(x) = 8 \cdot x^2 + 40 \cdot x + 50$
- $g(x) = x^3 - x^2 - x$

Laquelle de ces deux fonctions est la plus "grande" ? Pour des petites valeurs (0,1,2, etc), $f(x)$ est clairement plus grande. Mais pour des valeurs plus grandes, alors c'est $g(x)$ qui est plus grande : le facteur x^3, lorsque x est assez grand, deviendra beaucoup plus grand que tous les facteurs en x^2, x ou constants.

On peut alors simplifier de deux manières : si la fonction est une somme de plusieurs éléments, seul celui de plus grand ordre est conservé.
Et les facteurs constants peuvent également être ignorés : que nous ayons 0.1 x^3, x^3 ou 100 x^3 pour g, et 0.1 x^2, 8 x^2 ou 10000 x^2 pour f, lorsque x tend vers l'infini, c'est l'ordre de grandeur qui détermine quelle fonction croît le plus vite.

On s'intéresse alors à l'ordre de grandeur asymptotique de ces fonctions, en comparant l'ordre de grandeur des fonctions lorsque les variables deviennent immensément grandes (tendent vers l'infini).

Ici, on dit alors que g majore f, car g sera d'un ordre de grandeur asymptotique plus grand que f : g croît lorsqu'on tend x vers l'infini, g devient beaucoup plus grand, car le facteur x^3 devient beaucoup plus grand que tous les facteurs en x^2, x et constants. On note cela $f(x) = O(g(x))$:

Définition 6.1 (Notation "grand O"). *Soit f et g deux fonctions sur une variable réelle x. On dit que f est majorée par g (ou que g majore f), et on note :*
$f(x) = O(g(x))$
s'il existe des constantes N et c telles que :
$\forall x > N \quad |f(x)| \leq c \cdot |g(x)|$

Cela signifie, intuitivement, que f ne croît pas plus vite que g. Il est important de noter que cela inclus aussi la possibilité que f et g aient la même croissance (à une constante près, ici représentée par c).

Il s'agit souvent de la notation la plus utilisée en complexité : on peut avoir une complexité qui varie selon les cas, selon l'entrée, et on cherche souvent à donner une borne maximale. Par exemple, si la complexité de notre algorithme est $O(n^2)$ (i.e. est majorée par $g(x) = x^2$), on sait que notre complexité est quadratique ou mieux, mais en tout cas pas plus que quadratique.

De manière similaire, on peut définir la notation "opposée" qui indique que f est minorée par g, notée $f(x) = \Omega(g(x))$:

Définition 6.2 (Notation "grand Oméga"). *Soit f et g deux fonctions sur une variable réelle x. On dit que f est minorée par g (ou que g minore f), et on note :*
$f(x) = \Omega(g(x))$
s'il existe des constantes N et c telles que :
$\forall x > N \quad |f(x)| \geq c \cdot |g(x)|$

Et enfin, on peut définir la notation suivante, qui exprime le fait d'être du même ordre de grandeur :

Définition 6.3 (Notation "grand Thêta"). *Soit f et g deux fonctions sur une variable réelle x. On dit que f et g sont du même ordre de grandeur asymptotique, et on note :*
$f(x) = \Theta(g(x))$
s'il existe des constantes N, c_1 et c_2 telles que :
$\forall x > N \quad c_1 \cdot |g(x)| \leq |f(x)| \leq c_2 \cdot |g(x)|$

Cela signifie intuitivement que les deux relations ont le même ordre de grandeur, à une constante près. On peut noter que cela revient à dire que f est à la fois majorée et minorée par g, et donc que la notation Θ est équivalente à la combinaison des deux notations précédentes :

$f(x) = \Theta(g(x)) \Leftrightarrow f(x) = O(g(x)) \wedge f(x) = \Omega(g(x))$

Voici quelques exemples de fonctions et ordres associés :

fonction	exemples de O()	exemples de $\Omega()$	$\Theta()$
$f(n) = 8n^2 + 40n + 50$	$O(n^2)$, $O(n^5)$, $O(2^n)$	$\Omega(n^2), \Omega(n), \Omega(1)$	$\Theta(n^2)$
$f(n) = n^3 - n^2 - n$	$O(n^3)$, $O(n^4)$, $O(n!)$	$\Omega(n^3), \Omega(n \cdot log(n)), \Omega(n)$	$\Theta(n^3)$
$f(x) = 25 + log(n)$	$O(log(n))$, $O(n)$, $O(2^n)$	$\Omega(log(n)), \Omega(1)$	$\Theta(log(n))$
$f(x) = n \cdot log(n) + n + 1$	$O(n \cdot log(n))$, $O(n^2)$, $O(2^n)$	$\Omega(n \cdot log(n)), \Omega(n), \Omega(1)$	$\Theta(n \cdot log(n))$
$f(x) = 2^n$	$O(2^n)$, $O(3^n)$, $O(n!)$	$\Omega(2^n), \Omega(1.5^n), \Omega(n^3)$	$\Theta(2^n)$

6.2.1 Ordres de grandeurs courants

Voici une liste non exhaustive des ordres de grandeur les plus courants :

O(1)	majoré par une constante
O(log(n))	logarithmique
O(n)	linéaire
O(n log(n))	quasi-linéaire (ou linéarithmique)
O(n^2)	quadratique
O(n^c)	polynomiale (si c est entier positif)
O(c^n)	exponentielle (si $c > 1$)
O(n !)	factorielle

Voici un petit comparatif de l'évolution de quelques fonctions selon la taille de l'entrée, en supposant que toutes ces fonctions prennent un temps identique de 1s (1 seconde) pour une entrée de taille n = 10 bits :

entrée	10	20	30	40	50	100	200	400
1	1s	1s	1s	1s	1s	1s	1s	1s
$log(n)$	1s	1.3s	1.48s	1.6s	1.7s	2s	2.3s	2.6s
n	1s	2s	3s	4s	5s	10s	20s	40s
$n \cdot log(n)$	1s	2.6s	4.44s	6.4s	8.5s	20s	46s	1min44s
n^2	1s	4s	9s	16s	25s	1min40s	6min40s	26min40s
n^4	1s	16s	81s	4min16s	10min25s	2h46min40s	44h26min40s	$\simeq$ 9.37 jours
2^n	1s	17min4s	$\simeq$ 12 jours	$\simeq$ 34 ans	$\simeq$ 34800 ans	$\simeq 4 \cdot 10^{19}$ans	$\simeq 5 \cdot 10^{49}$ans	$\simeq 10^{110}$ans

On considère en général qu'un algorithme est utilisable en pratique si sa complexité est polynomiale ou mieux (et de degré pas trop élevé).

6.3 Complexité en temps et en espace

Pour terminer, nous allons définir les notions de complexité algorithmique en temps et en espace.

6.3.1 Complexité en temps

Commençons par la complexité en temps.

L'intuition est assez simple : la complexité en temps d'un algorithme, c'est le temps qu'il faut pour que l'algorithme réponde.

Mais dans la pratique, c'est un peu plus compliqué que cela : comment définit-on le temps dans le cadre d'un algorithme ? Comment gère-t-on le fait que l'algorithme peut mettre un temps différent selon l'entrée, et selon la taille de l'entrée ?

On définit donc la complexité en temps de la manière suivante :

Définition 6.4 (Complexité en temps). *la complexité en temps d'un algorithme/machine de Turing, pour une entrée de taille n, est une fonction de n, notée T(n), qui décrit le nombre d'étapes de calcul maximal effectué par l'algorithme/machine sur une entrée de taille n.*

Cela signifie donc que la complexité est une fonction de la taille de l'entrée : par exemple, une complexité en temps $f(n) = n^2 + n$ signifie que pour une entrée de taille n, l'algorithme mettra au maximum $f(n) = n^2 + n$ étapes de calcul pour répondre.

Il s'agit donc du "pire cas" : si le temps varie entre toutes les entrées de taille n, on prend le pire temps : cela assure que, quelque soit l'entrée, on réponde bien dans le temps indiqué.

Plutôt que de calculer le temps exact, on donne fréquemment donc plutôt une borne maximale : par exemple, si un algorithme est $O(n^2)$, alors on sait que son temps est au pire quadratique.

La difficulté, dans cette définition, est de définir ce qu'est une "étape" de calcul, car cela dépend du modèle utilisé.

Pour une machine de Turing, c'est assez simple : une étape, c'est une transition dans la machine formelle.

Dans un algorithme, c'est un peu plus flou : quelle est l'étape de "base" , qu'est-ce qui compte comme une étape de calcul ? Cela dépend en fait du modèle utilisé pour la machine qui applique l'algorithme.
On considère en général qu'il s'agit d'une opération "de base" dans l'ordinateur, i.e. une opération en langage machine (registre à registre).

Lorsqu'on veut analyser la complexité d'un algorithme, on fait parfois abstraction du modèle utilisé, et on pose ce qu'on considère comme opération unitaire (par exemple, dans un algorithme mathématique complexe, on pourrait poser que l'addition/multiplication prend un temps O(1)) : cela permet d'analyser la complexité de l'algorithme ou certains de ses aspects, sans dépendre de la machine utilisée. Il est important de toujours définir le modèle utilisé ou ce qui est considéré comme une opération unitaire (et si on définit des opérations comme unitaires, lorsqu'on applique l'algorithme, de bien comprendre quelle complexité ces opérations définies comme unitaires auront réellement dans le modèle utilisé).

Attention, la complexité en temps n'est donc pas directement mesurée en "temps" au

sens ou on l'entend couramment : elle est mesurée en étapes de calcul.
Elle est bien entendu liée au temps réel, mais le temps réel dépend aussi des caractéristiques de la machine utilisée, des conditions, etc.

6.3.2 Complexité en espace

Similairement, on peut définir la complexité en espace. Intuitivement, c'est la quantité d'espace (i.e. de mémoire) nécessaire pour la réalisation de l'algorithme :

Définition 6.5 (Complexité en espace). *La complexité en espace d'un algorithme/-machine de Turing, pour une entrée de taille n, est une fonction de n, notée S(n), qui décrit la quantité maximale de cases mémoire visitées au cours de l'algorithme/machine sur une entrée de taille n.*

De nouveau, on garde cette notion de pire cas pour la complexité en espace : il s'agit du maximum entre toutes les entrées de taille n.

Pour une machine de Turing, il s'agit du nombre de cases visitées (y compris si on ne modifie pas le contenu de la case) sur toutes les bandes utilisables en écriture.

Pour un algorithme en général, cela dépend de nouveau de ce qu'on considère comme une "case mémoire" (bit, octet, registre, etc).
On considère en général le plus petit espace adressable, pour nos ordinateurs actuels généralement 32/64 bits (ici, pour la complexité en espace, cela n'a pas vraiment d'impact, car cela ne change pas l'ordre de la complexité : qu'on ait 1 bit, 8 bits ou 32/64 bits, cela ne change que d'une constante).

6.3.3 Exemple

Considérons le problème suivant : on a une liste déjà triée en entrée, contenant n nombres de taille constante c. On cherche s'il existe dans cette liste la valeur 0.

Si l'on considère une machine de Turing : on parcourt la liste, élément par élément, jusqu'à soit trouver la valeur cherchée et accepter, soit atteindre la fin de la liste et refuser. Cela prendra donc un temps O(n) (on parcourt la liste de n éléments, chacun de longueur constante c, on a donc $c \cdot n$ étapes de calcul, ce qui est O(n)).

Concernant la complexité en espace, cette machine ne fait que lire l'entrée. On écrit simplement un 0/1 pour répondre, on ne visite qu'une seule case : on a donc une complexité en espace constante O(1).

Si l'on considère une machine RAM ou un ordinateur, qui permet un accès n'importe où dans les données : on peut alors appliquer une recherche par dichotomie. A chaque étape, on calcule la position à laquelle accéder (temps constant, c'est une opération logique bit à bit), puis on accède et lit l'élément à cette position et le compare à la valeur cherchée). On répète cela au maximum log(n) fois, ce qui prend un temps total $O(log(n))$.

Et concernant la complexité en espace ? Cette recherche nécessite d'écrire la position à laquelle chercher (valeur entre 1 et n, ce qui s'écrit sur $log_2(n)$ bits au maximum). On a donc une complexité en espace $O(log_2(n))$.

Pour un même problème, il existe nombre d'algorithmes plus ou moins efficaces : certains sont plus rapides, d'autres prennent moins d'espace, certains sont dépendants du modèle utilisé.
Il est souvent compliqué, voire impossible d'optimiser le temps et l'espace à la fois, et il est en général nécessaire de faire un compromis entre les deux.