Calculabilité et Complexité (11X008) - Printemps 2024

7. Classes de complexité basiques

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

Dans ce cours, nous revenons (rapidement) sur les notations asymptotiques et définissons deux classes de complexité génériques pour le temps et l'espace. Puis nous passons en revue quelques classes bien connues qui en sont des cas particulier, en s'attardant sur le cas du temps polynomial, qui est l'une des plus importantes.

7.1 Notations asymptotiques (rappels et compléments)

Nous rencontrons souvent les notations O, Ω, Θ (prononcer "grand o", "grand omega" et "grand theta") et parfois aussi les notations o et ω ("petit o", "petit omega"). Ces notations ne sont pas exclusives à l'informatique. Elles permettent de manipuler des expressions mathématiques en ignorant des détails jugés non significatifs, afin de simplifier les calculs.

Mathématiquement, elles se définissent en termes de limites (voir le cours précédent). Une autre façon de les définir revient à :

- ignorer les termes dominés (contribution marginale quand $n \to \infty$)
- ignorer les facteurs constants (facteurs qui ne font pas intervenir n)

Exemple : $2n^3 + 3n^2 + 4$. On ignore les termes dominés : $2n^3 \cancel{+3n^2 + 4}$, puis on ignore les facteurs constants : $\cancel{2}n^3$, ce qui donne n^3. Autre exemple : $n + \log n$ devient n (car $\log n$ devient négligeable par rapport à n lorsque $n \to \infty$). Par contre, on n'ignore pas les facteurs dominés (attention) ! Par exemple, $n \cdot \log n$ n'est pas simplifiable en n.

Une fois ces simplifications faites, on a les équivalences suivantes :

Notation	$a = o(b)$	$a = O(b)$	$a = \Theta(b)$	$a = \Omega(b)$	$a = \omega(b)$
Signification	$a < b$	$a \leq b$	$a = b$	$a \geq b$	$a > b$

Exemples : $2n^3 + 3n^2 + 4 = \Theta(n^3)$. On aurait aussi pu écrire, de manière moins précise, que ça vaut $\Omega(n^3)$, $o(n^{10})$, ou encore $O(n^3)$ ou $O(n^{10})$, mais pas $o(n^3)$ ni $\Omega(n^4)$, par exemple.

Notez que le symbole "=" n'a pas la même signification qu'habituellement. En particulier, il n'est pas symétrique, par exemple $a = O(b)$ n'implique pas que $b = O(a)$. Pour éviter les confusions, on utilise parfois le symbole $\in$ à la place (avec la même signification).

Adjectifs (rappels)

Constant	$O(1)$	Polynomial	$O(n^c) = n^{O(1)}$
Logarithmique	$O(\log n)$	Quasi-polynomial	$n^{log^{O(1)}n}$
Linéaire	$O(n)$	Exponentiel	$2^{n^{O(1)}}$ ou parfois juste $O(2^n)$
Quasi-linéaire	$O(n \log n)$	Factoriel	$O(n!) = O(n^n)$
Quadratique	$O(n^2)$		

Les mêmes adjectifs sont utilisés pour Θ et Ω (en disant "au moins ..." pour Ω). Pour o et ω, on parlera plutôt de super-X et de sous-X (où X est l'adjectif de votre choix), par exemple, $\omega(n)$ est super-linéaire et $o(n^2)$ est sous-quadratique. Ces adjectifs peuvent s'appliquer aux algorithmes aussi bien qu'aux problèmes qu'ils résolvent. Notez que les bases des logarithmes ne sont pas précisées, car les logs en bases différentes ne diffèrent que par des facteurs constants (c'est bien pratique !).

7.2 Classes génériques pour le temps et l'espace

Il existe de nombreuses classes de complexité, qui correspondent à des ensembles de problèmes dont la résolution requiert un certain niveau de ressource, en général, en temps ou en espace, et parfois d'autre choses (quantité d'aléa, non-déterminisme, *etc.*). Elles s'expriment en fonction de la taille n de l'entrée à traiter.

La plupart de ces classes sont des cas particulier de deux classes génériques. Nous les définissons ici en termes de langages (problèmes de décision), mais leur définition s'étend naturellement aux autres formes de problèmes (problèmes de recherche, d'optimisation, de dénombrement, etc.).

- $\mathsf{TIME}(f(n))$: Ensemble des langages qui peuvent être décidés en temps $O(f(n))$, sans se soucier de l'espace.
- $\mathsf{SPACE}(f(n))$: Ensemble des langages qui peuvent être décidés en espace $O(f(n))$, sans se soucier du temps.

La définition exacte de ces classes requiert de spécifier le modèle de machine utilisé. Par exemple, machine de Turing à une bande ou plusieurs bandes, machine RAM, etc. (La bonne nouvelle est que pour beaucoup d'entre elles, cela n'a pas d'importance.)

Exemple

Soit P_{find} le problème de décider si un mot de longueur n contient un certain caractère. Il suffit de parcourir les caractères du mot un à un et vérifier si le caractère courant est égal à celui que l'on cherche (en s'arrêtant dès qu'on le trouve). Dans le pire des cas, on parcourt tous les caractères avec un nombre constant d'opérations pour chacun, le problème

est donc résoluble en temps $n \cdot O(1) = O(n)$. Qu'en est-il de l'espace ? Hormis la taille de l'entrée, qui n'est pas comptabilisée, et l'écriture d'un bit de sortie (OUI/NON), ce problème n'utilise aucun autre espace, il est donc résoluble en espace constant $O(1)$. Au final, on a donc $P_{find} \in \mathsf{TIME}(n)$ et $P_{find} \in \mathsf{SPACE}(1)$, on peut aussi écrire $P_1 \in \mathsf{TIME}(n) \cap \mathsf{SPACE}(1)$.

Quid de décider si un mot possède au moins deux caractères identiques ? Clairement, c'est dans $\mathsf{TIME}(n^2) \cap \mathsf{SPACE}(1)$. Peut-on mieux faire ? (spoiler : oui !)

Inclusions simples

Si $t_1(n) \leq t_2(n)$, on a bien sûr $\mathsf{TIME}(t_1(n)) \subseteq \mathsf{TIME}(t_2(n))$ et $\mathsf{SPACE}(t_1(n)) \subseteq \mathsf{SPACE}(t_2(n))$. Déterminer si ces inclusions sont strictes lorsque $t_1(n) < t_2(n)$ est moins évident, nous y reviendrons. On peut aussi s'interroger sur les relations entre TIME et SPACE (spoiler : en général, $\mathsf{SPACE}(t(n))$ contient *strictement* plus de problèmes que $\mathsf{TIME}(t(n))$).

7.3 Classes incontournables

Les classes suivantes sont des cas particuliers très utilisés de TIME et SPACE :

- $\mathsf{LOGSPACE} = \mathsf{SPACE}(\log n)$ Langages décidables en espace logarithmique $O(\log n)$.
- $\mathsf{P} = \mathsf{TIME}(n^{O(1)})$ Langages décidables en temps polynomial $n^{O(1)}$.
- $\mathsf{PSPACE} = \mathsf{SPACE}(n^{O(1)})$ Langages décidables en espace polynomial $n^{O(1)}$.
- $\mathsf{EXP} = \mathsf{TIME}(2^{n^{O(1)}})$ Langages décidables en temps exponentiel $2^{n^{O(1)}}$.

Notez que nous n'avons pas défini de classe particulière pour le temps logarithmique. La raison est que pour la grande majorité des problèmes usuels, un algorithme (ou une machine de Turing) doit au moins lire l'intégralité de son entrée, ce qui prend déjà un temps $O(n)$. Il y a des exceptions, comme chercher un élément dans une liste triée que l'on peut adresser directement [1].

La semaine prochaine, nous montrerons que $\mathsf{LOGSPACE} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$. Nous montrerons aussi que $\mathsf{P} \subsetneq \mathsf{EXP}$ (inclusion *stricte*) et $\mathsf{LOGSPACE} \subsetneq \mathsf{PSPACE}$.

7.4 Pourquoi P est la plus importante

La classe la plus étudiée est certainement P, l'ensemble des langages décidables en temps polynomial $n^{O(1)}$. La raison est que cette classe capture, d'un point de vue théorique, les

1. Adresser directement = consulter les éléments à une certaine position sans parcourir les précédents. Ce n'est pas possible pour une machine de Turing, mais ça l'est pour une machine RAM (nos ordinateurs).

problèmes qui peuvent être résolus efficacement (rapidement). Bien sûr, c'est sujet à discussion. On peut s'interroger, par exemple, sur l'efficacité réelle d'un algorithme qui prendrait un temps n^{10}. Pour certaines applications où n est grand, même une complexité de n^2 peut s'avérer trop lent. Cependant, la classe P a de nombreux avantages, parmi lesquels :

- Indépendance de la machine : La classe P est la même pour tous les modèles de machine "raisonnables" (ordinateurs classiques, machine de Turing, machines parallèles, modèles biologiques, *etc.*). La raison est que n'importe lequel de ces modèles peut simuler les autres en temps polynomial.
- Composabilité : Si un algorithme fait appel à un autre algorithme polynomial un nombre polynomial de fois, alors cela résulte en un algorithme polynomial.
- Petits exposants en pratique : L'écrasante majorité des problèmes qui se posent naturellement et qui sont dans P s'avèrent finalement avoir une complexité qui n'excède pas $O(n^2)$ ou $O(n^3)$. On peut donc estimer que pour des tailles de problèmes raisonnables, la classe P capture bel et bien ce qui peut être résolu efficacement.

7.5 Quelques exemples

Tous les problèmes suivants ont une complexité en temps linéaire, quasi-linéaire ou quadratique :

- Trier une liste de n éléments
- Décider si un graphe donné est connexe
- Trouver le plus court chemin d'un point A à un point B
- Calculer la capacité maximale d'un réseau routier
- Multiplier deux matrices
- Décider si un graphe peut être colorié avec 2 couleurs sans que deux sommets voisins aient la même couleur

En revanche, aucun algorithme en temps polynomial n'est connu pour les problèmes suivants :

- Décider si un graphe peut être colorié avec 3 couleurs
- Trouver un itinéraire optimal qui parcourt un ensemble de destinations
- Créer des emplois du temps optimaux
- Factoriser un grand nombre
- Miner des bitcoins

Nous aurons l'occasion de revenir sur la plupart de ces problèmes.