INFO607, L3 Informatique, Algorithmique II
Leçon 1, Complexité des algorithmes, notations $\mathcal{O}$, Θ, et Ω et analyse en pire cas

Jacques-Olivier Lachaud
LAMA, Université Savoie Mont blanc
https://www.lama.univ-savoie.fr/wiki
(suivre INFO607)

14 février 2022

1 Complexité des algorithmes (Rappels)

Il y a souvent deux buts contradictoires lorsque l'on cherche à mettre au point un algorithme pour résoudre un problème donné :

1. L'algorithme doit être facile à comprendre, coder, maintenir, mais aussi facile à vérifier.
2. L'algorithme doit utiliser efficacement les ressources de l'ordinateur, c'est-à-dire s'exécuter rapidement mais aussi prendre une place raisonnable en mémoire.

Si un algorithme doit être utilisé très souvent, il est alors intéressant de mettre en œuvre une solution complexe mais efficace en temps et/ou en espace mémoire. Il est alors utile de pouvoir comparer objectivement les complexités relatives.

1.1 Mesure du temps d'exécution d'un programme

Le temps d'exécution d'un programme dépend :

1. des données en entrée,
2. de la qualité du code généré par le compilateur,
3. de la nature et de la rapidité des instructions de la machine d'exécution du programme,
4. de l'algorithme utilisé pour résoudre le problème.

D'après le premier point, il est clair que le temps d'exécution n'est pas juste une valeur, mais une fonction des données. Très souvent, la valeur des données n'est pas significative, mais seul compte le nombre de données, mettons n. Le temps d'exécution d'un programme sera donc une fonction $T(n)$, qui est le temps d'exécution de ce programme pour n données en entrée. Par exemple, il est clair qu'un programme de tri sera de plus en plus lent si on augmente le nombre de données à trier. Attention, dans certains (rares) cas, la valeur de n peut grandement changer le temps d'exécution (e.g. suite de Syracuse) : il faudra donc rester méfiant.

Maintenant l'unité de temps de $T(n)$ ne peut être précisée du fait des points (2) et (3). L'unité ne sera donc que relative. Un même programme P aura peut-être un temps d'exécution $T_1(n) = c_1 n^2$ sur une machine $M1$ et temps d'exécution $T_2(n) = c_2 n^2$ sur une machine $M2$. Si les constantes c_1 et c_2 peuvent être distinctes (et très variables), il est en revanche peu probable que la partie n^2 du temps d'exécution se transforme d'une machine à une autre. En effet, un processeur peut être cadencé plus rapidement qu'un autre, mais globalement, s'il doit faire K opérations, cela lui prendra un temps proportionnel à K.

On dira donc souvent que le programme P s'exécute en un temps proportionnel à n^2, et non s'exécute en $c_1 n^2$ sur la machine $M1$, car cela ne présente pas toujours un intérêt majeur.

Parfois, le temps d'exécution d'un programme peut être rapide sur n données mais lent sur n autres données. Un exemple typique est le tri insertion avec des données déjà triées, qui est rapide, mais qui est lent sur la plupart des autres données. Dans ces cas-là, $T(n)$ désignera le temps d'exécution dans le *pire cas*, car c'est celui qui est problématique.

Une autre façon est de définir le temps d'exécution *moyen* $\hat{T}(n)$, qui est la moyenne des temps d'exécution de toutes les données de taille n. Si cette mesure peut paraître plus utile ou objective, il faut néanmoins garder à l'esprit que les ensembles de n données sont rarement équiprobables dans les applications réelles. Dans le cas du tri, on a souvent des données quasi-triées en entrée, du fait des processus de saisie ou d'acquisition. Néanmoins, on montrera dans certains cas comment calculer $\hat{T}(n)$, et sous quelles hypothèses ce temps est valide.

Exemples :

1. L'algorithme de calcul du plus grand élément d'un tableau à n éléments nécessite de regarder toutes les cases du tableau une fois exactement. Le temps d'exécution dans le pire cas est donc proportionnel à n. Comme dans le meilleur cas il est aussi proportionnel à n, il est clair que le temps d'exécution moyen est proportionnel à n lui-aussi.
2. Un algorithme de recherche dichotomique dans un tableau trié est beaucoup plus rapide. On montre que son temps d'exécution est proportionnel à $\log_2 n$, dans le pire cas et dans le cas moyen aussi.
3. L'algorithme de tri insertion a un temps d'exécution dans le pire cas proportionnel à n^2, mais son temps d'exécution moyen est moins clair. Si on suppose que tous les ordres sont équiprobables, on peut montrer que le temps d'exécution moyen est aussi proportionnel à n^2 (avec une constante inférieure).

1.2 Notations O, Θ, Ω

Lorsque l'on veut comparer les vitesses d'accroissement de fonction sans se préoccuper des constantes mises en jeu, il est pratique d'utiliser une notation concise pour exprimer la notion de proportionnalité, où le fait qu'une fonction grandit plus vite ou moins vite qu'une autre "à l'infini". On dispose pour cela de trois notations classiques : O = "grand O", Θ = "Téta", Ω = "grand Oméga".

Dans la suite T et f sont deux fonctions de n.

- $T(n) = O(f(n))$ ssi il existe deux constantes c et n_0 telles que $\forall n \geq n_0, T(n) \leq cf(n)$. Cette notation indique que "T ne croît pas plus vite que f".
  De manière équivalente, la limite de $\frac{T(n)}{f(n)}$ lorsque n tend vers l'infini est borné par une constante.
- $T(n) = \Omega(f(n))$ ssi il existe deux constantes c et n_0 telles que $\forall n \geq n_0, T(n) \geq cf(n)$.[1] Cette notation indique que "T ne croît pas moins vite que f".
- $T(n) = \Theta(f(n))$ ssi il existe trois constantes c_1, c_2 et n_0 telles que $\forall n \geq n_0, c_1 f(n) \leq T(n) \leq c_2 f(n)$. Cette notation indique que "T et f croissent aussi vite".

Une notation importante est $O(1)$ qui exprime la croissance de toute fonction constante. Ainsi, on dira qu'un ensemble d'instructions dont le temps d'exécution ne dépend pas de la taille des données en entrée et est borné par une constante a une complexité $O(1)$.

Voilà une liste de règles utiles avec ces notations :

Facteurs constants inutiles $\alpha f(n) = \Theta(f(n))$ pour constante $\alpha > 0$. On peut donc ignorer les facteurs constants dans toutes les expressions.

Complexité constante $f(n) = \mathcal{O}(1)$ indique une fonction bornée par une constante.

Transitivité de $\mathcal{O}$ $f(n) = \mathcal{O}(g(n))$, et $g(n) = \mathcal{O}(h(n))$, implique $f(n) = \mathcal{O}(h(n))$

Transitivité de Ω $f(n) = \Omega(g(n))$, et $g(n) = \Omega(h(n))$, implique $f(n) = \Omega(h(n))$

Transitivité de Θ $f(n) = \Theta(g(n))$, et $g(n) = \Theta(h(n))$, implique $f(n) = \Theta(h(n))$

Règle des sommes $f(n) + g(n) = \mathcal{O}(\max(f(n), g(n))$

Règle des sommes $f(n) + g(n) = \Omega(\min(f(n), g(n))$

Règle des produits $f(n)\mathcal{O}(g(n)) = \mathcal{O}(f(n)g(n))$, vrai pour Θ et Ω aussi.

Polynômes $n^a = \mathcal{O}(n^b)$ lorsque $0 \leq a \leq b$.

Polynômes $a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k = \Theta(n^k)$ lorsque $a_k > 0$ (seul le monôme de plus grand degré compte dans la complexité).

Equ. logarithmes $\log_b n = \Theta(\ln n)$, on écrira donc souvent log sans préciser.

Logarithmes $\log n = \Omega(1)$ et $\log n = \mathcal{O}(n^\epsilon)$, pour tout $\epsilon > 0$ (log est dominé par les polynômes non constants)

Exponentielles $a^n = \mathcal{O}(b^n)$, pour $0 < a \leq b$

Exponentielles $e^n = \Omega(n^k)$ pour n'importe quel $k > 0$ (exp domine les polynômes).

Exemples :

- Il est clair que $n = O(n)$, $n = \Omega(n)$, et $n = \Theta(n)$.
- Plus généralement, $f(n) = O(\alpha f(n))$, $f(n) = \Omega(\alpha f(n))$ et $f(n) = \Theta(\alpha f(n))$.
- On a aussi que $n = O(n^2)$, $n^2 = O(n^3)$ et plus généralement $n^a = O(n^b)$ ssi $0 \leq a \leq b$.
- On a bien sûr $n = O(n \log n)$ et $n \log n = O(n^2)$

Exercice : (Notations O, Θ, Ω)

1. montrez (a la mano) que $4n = O(n^2/2)$
2. montrez (a la mano) que $n^2 + 5n = O(n^2)$
3. montrez que $T(n) = \Theta(f(n))$ ssi $T(n) = O(f(n))$ et $T(n) = \Omega(f(n))$.
4. montrez que $T(n) = O(f(n))$ ssi $f(n) = \Omega(T(n))$.
5. (transitivité) montrez que si $T(n) = O(f(n))$ et $f(n) = O(g(n))$ alors $T(n) = O(g(n))$.
6. (transitivité) montrez que si $T(n) = \Theta(f(n))$ et $f(n) = \Theta(g(n))$ alors $T(n) = \Theta(g(n))$.

On dispose de règles d'addition et de multiplication de ces notations, notamment :

1. Une définition non symétrique parfois utilisée est de dire qu'il existe une infinité de $n \geq n_0$ pour lesquels $T(n) \geq cf(n)$, mais pas forcément tous.

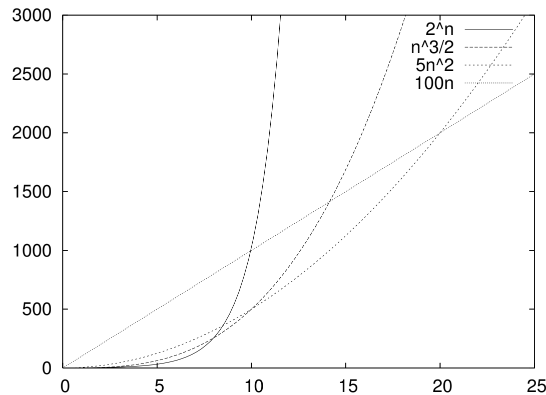

FIGURE 1 – Temps d'exécution de quatre programmes différents, de temps d'exécution respectifs 2^n, $n^3/2$, $5n^2$, $100n$. L'unité de temps est sans importance, mettons des secondes.

Addition de O. Si $T_1(n) = O(f(n))$ et $T_2(n) = O(g(n))$, alors $T_1(n)+T_2(n) = O(\max(f(n), g(n)))$. C'est notamment utile lorsque vous avez mesuré la complexité de deux parties successives de votre programme et que vous cherchez à déterminer la complexité du programme tout entier. Il s'agit bien de l'addition de deux temps.

Produit de O. Si $T_1(n) = O(f(n))$ et $T_2(n) = O(g(n))$, alors $T_1(n)T_2(n) = O(f(n)g(n))$. Cela montre par exemple que $O(n^2/2) = O(n^2)$. La règle des produits est utilisée pour mesurer le temps d'exécution de programme contenant des boucles ou des appels répétitifs à un même sous-programme de complexité connue.

Quelques questions :

1. Comment montrer que $\log n = \mathcal{O}(n)$?
2. Utilisez la règle des produits : "$f(n) = \mathcal{O}(g(n))$ implique $h(n)f(n) = \mathcal{O}(h(n)g(n))$", pour déduire que $n \log n = \mathcal{O}(n^2)$.
3. Est-ce que $\log n = \Theta(n)$?
4. Comment montrer aussi que $n + \log n = \mathcal{O}(n)$?
5. Est-ce que $n + \log n = \Theta(n)$?
6. Comment montrer que $\log n = \mathcal{O}(n^a)$ pour $a > 0$?

1.3 Complexité et temps d'exécution asymptotique

Il n'est donc pas facile de comparer les efficacités respectives d'algorithmes, sachant que leur vitesse d'exécution dépend de beaucoup de paramètres, dont la machine. On va voir néanmoins que l'on dispose d'un moyen pour le faire qui est assez objectif.

Supposons par exemple que l'on dispose de quatre programmes $(P_i)_{i=1..4}$ qui résolvent le même problème. Chaque programme P_i s'exécute sur une machine M_i. On note $T_i(n)$ leurs temps d'exécution respectifs, que l'on peut observer sur la Figure 1.

Lequel est le meilleur ? Cela dépend de la taille des données à traiter et du temps que l'on peut y consacrer. Si on suppose que l'on ne dispose que de 10^3 secondes, ces quatre programmes/machines sont quasiment aussi efficaces les uns que les autres. Si maintenant on dispose de 10^4 secondes, on s'aperçoit que c'est le programme/machine avec le taux d'accroissement le plus faible qui devient vite le plus efficace. Ainsi, pour un algorithme en $O(n)$, le gain réalisé est identique au temps rajouté, ce qui n'est pas le cas pour les autres.

$T(n)$	Taille max pour 10^3 s	Taille max pour 10^4 s	Gain
$100n$	10	100	10
$5n^2$	14	45	3,2
$n^3/2$	12	27	2,3
2^n	10	13	1,3

Un façon complètement symétrique de voir les choses est de supposer que l'on garde les mêmes programmes compilés de la même façon, mais qu'on puisse cadencer les processeurs dix fois plus vite. Le gain observé pour le même temps sera alors complètement similaire au fait de se donner dix fois plus de temps.

On en conclut que lorsqu'on veut traiter des données de plus en plus grandes, il est intéressant de comparer les temps d'exécution en terme d'accroissement O, c'est-à-dire de manière *asymptotique*, en négligeant les constantes qui ne sont pertinentes que pour des petites données. La *complexité en temps* d'un programme est donc son temps d'exécution mesuré en terme d'accroissement de la taille des données en entrée.

Exemples :

1. Sur l'exemple précédent, la meilleure complexité est celle du programme de temps $100n$, même si ce n'est pas le programme le plus efficace pour de petites valeurs de n.
2. Dans certains cas, la constante est importante. Il existe un problème d'optimisation classique (programmation linéaire) dont l'algorithme classique dit du simplexe est efficace en pratique, mais peut avoir une complexité exponentielle dans certains cas. Il existe un algorithme de complexité polynomiale qui résoud le même problème, mais la constante est très importante et sur toutes les données que l'on peut traiter le rend inutilisable.

1.4 Calcul de la complexité d'un algorithme

On peut maintenant déterminer (à des constantes près) la complexité d'un algorithme donné. Attention, on est souvent obligé de donner une complexité dans le pire cas, notamment lorsque le programme a des morceaux d'instructions qui sont conditionnés.

Les règles sont les suivantes :

— Le temps d'exécution de chaque affectation, lecture, écriture en mémoire est supposé être constant ou en $O(1)$.

— De même, on suppose souvent (mais pas toujours) que le temps d'exécution de l'addition, soustraction, multiplication, division est constant (i.e. $\mathcal{O}(1)$). Cela est faux en général, mais assez vrai lorsqu'on limite la taille des données à des valeurs codées sur moins de 32 ou 64 bits.

— Si $T_1(n)$ et $T_2(n)$ sont les temps d'exécution de deux fragments de programme, le temps d'exécution de leur succession est $T_1(n) + T_2(n)$. Si $T_1(n) = O(f(n))$ et $T_2(n) = O(g(n))$ alors la règle des sommes donne $T(n) = O(\max(f(n), g(n)))$.
En particulier, une succession finie d'instructions élémentaires prend $O(\max(1, 1, \ldots, 1))$, soit $O(1)$.

— Le temps d'exécution d'un "Si" est le temps d'exécution de la condition (souvent $O(1)$) plus le temps d'exécution le plus large entre la partie "alors" et la partie "sinon". On note que le temps d'exécution devient un temps dans le pire cas.
On peut utiliser la notation Ω pour le meilleur cas.

— Le temps d'exécution d'une boucle est la somme de tous les temps d'exécution du bloc interne plus les temps d'exécution de la condition de terminaison. Si le nombre d'itération maximal $O(f(n))$ est connu et que le temps d'exécution du bloc interne est borné par $O(g(n))$, alors le temps d'exécution de la boucle est $O(f(n)g(n))$ (règle des produits).

— Pour les appels de fonction/procédure, il faut bien sûr sommer leur temps d'exécution. Si l'appel est récursif, il est en général sur une partie plus petite des données. On obtient donc une relation de récurrence sur les temps d'exécution, et il existe des techniques classiques pour trouver la forme close qui correspond à la récurrence.
Nb : exemple de calcul de la factorielle : $T(n) = c + T(n-1)$. On en déduit $T(n) = cn = O(n)$.

1.5 Complexités usuelles des algorithmes

Les algorithmes peuvent être classées par ordre croissant ainsi :

Temps constant $\mathcal{O}(1)$ En gros, les instructions élémentaires.

Temps logarithmique $\mathcal{O}(\log n)$ Typiquement, la recherche dichotomique, le calcul du plus grand commun diviseur (pour des entiers pas trop grands), l'insertion ou la suppression dans un tas.

Temps linéaire $\mathcal{O}(n)$ Maximum, minimum, moyenne, inversion d'un tableau, ... Notez que tout algorithme "utile" qui prend en entrée n données quelconques doit au moins les lire une fois pour les prendre en compte : c'est donc la complexité minimale de la plupart des algorithmes qui travaillent sur un nombre variable de données.

Temps quasi-linéaire $\mathcal{O}(n \log n)$ Typiquement les meilleurs algorithmes de tri (tri fusion, tri par tas, tri par monotonie), algorithmes de calcul d'enveloppe convexe.

Temps quadratique $\mathcal{O}(n^2)$ Somme de deux matrices $n \times n$, tri à bulle/sélection/insertion, calcul efficace des coefficients binomiaux.

Temps cubique $\mathcal{O}(n^3)$ Résolution de systèmes linéaires (pivot de Gauss, décomposition de Cholesky, etc), algorithme Hongrois des mariages parfaits

Temps polynomial $\mathcal{O}(n^a)$ ou classe P. Inclut les algorithmes précédents et plus généralement tous les algorithmes de temps d'exécution bornés par un polynôme en n. Par exemple, l'optimisation par programmation linéaire en dimension finie, max-flow sur les graphes, produit de matrices.

Au delà Pour beaucoup de problèmes, on ne sait pas s'il existe un algorithme en temps polynomial pour le résoudre. On a notamment la classe NP des algorithmes où on peut vérifier en temps polynomial qu'une solution est correcte (par exemple problème de décision du voyageur de commerce), mais on ne sait pas écrire d'algorithme pour trouver une solution en temps polynomial. C'est le fameux "P $\neq$ NP ?".

Voyageur de commerce, 3-SAT, maximum independent sets, cycle hamiltonien.

On note que l'on a bien montré la complexité en pire cas d'un algorithme en $O(f(n))$ lorsqu'on peut exhiber un exemple d'exécution où le temps d'exécution atteint bien asymptotiquement ce $f(n)$. C'est la même chose pour le meilleur cas. Ainsi $O(n^3)$ est une borne supérieure de la complexité dans le pire cas du tri à bulle, mais n'est jamais atteinte. De même $\Omega(n)$ est une borne inférieure de la complexité dans le meilleur cas de ce même algorithme, mais n'est jamais atteinte non plus.

Quelques questions :

- Montrez que la complexité d'un algorithme de sommation conditionnelle est $O(n)$. Exemple la moyenne des notes différentes de 0.
- Montrez que le tri à bulle est un $O(n^2)$.
- Montrez que le pire cas de quicksort est un $O(n^2)$.
- Quelle est la complexité de la recherche dichotomique ?
- Quelle est la complexité des calculs récursif/itératif de la factorielle ?
- Quelle est la complexité de l'algorithme du sac-à-dos ?
- Quelle est la complexité de calcul de la version récursive du binomial C_n^p ? (Remarquez que la somme des binomiaux fait 2^n).
- Quelle est la complexité du calcul de l'enveloppe convexe par l'algorithme de Graham ? Par Melkman ?
- Quelle est la complexité des algorithmes Insérer et SupprimerMin des tas ? En déduire la complexité du tri par tas ?