Complexité

La complexité est un concept du coût des algorithmes, qui permet de comparer l'efficacité des algorithmes.

Un algorithme est une suite finie d'étapes (instructions qui peuvent être traduites dans le language d'un ordinateur) permettant de résoudre un problème (par exemple pour le calcul d'une fonction). La complexité d'un algorithme est une mesure du *temps de calcul* (ou *temps* tout court) en fonction de la taille des données. On parle aussi du *coût* de l'algorithme. La notion du temps peut être en nombre d'opérations binaires (changements de bits) ou en fonction d'autres opérations élémentaires (comme les multiplications et les additions). Les opérations élémentaires peuvent, elles-mêmes, avoir un coût en opérations binaires en fonction du choix de l'algorithme pour les résoudre.

Le lien entre le coût d'un algorithme en opérations binaires et le temps CPU ou réel dépend du matériel utilisé. On parle de temps (ou coût) de calcul (binaire) pour avoir une mesure théorique indépendante de sa mise en application dans un ordinateur (mais qui doit préserver une correspondance linéaire en passant entre ordinateurs). On peut aussi distinguer le coût d'un algorithme en temps et en espace [= mémoire] (mais le coût en espace minorise le coût en temps).

La plupart des données d'entrée sont précisées par des entiers (ou des suites finies d'entiers). Pour n dans $\mathbb{N}^*$, la taille de n (en bits) est le nombre de chiffres (en représentation binaire), qui vaut $\lfloor \log_2(x) \rfloor + 1$. Par exemple :

n	n_2	$\lfloor \log_2(n) \rfloor + 1$
1	1_2	1
2	10_2	2
3	11_2	2
4	100_2	3
5	101_2	3
6	110_2	3
7	111_2	3
8	1000_2	4
$2^{128}-1$	$11\ldots1_2$	128

Pour un algorithme on note $T(x)$ le temps maximal de calcul pour toutes données de taille $\leq x$. Par conséquent

- $T(x) \leq T(y)$ si $x \leq y$;
- si y est la taille maximale d'un objet intermédiaire (le coût en espace) et z la taille des données de sortie, alors $\max(y, z) \leq T(x)$;
- il est possible que $x > T(x)$, si on n'utilise pas tous les données d'entrée.

Exemple. Accès dans un tableau.
Entrée : Une liste de n octets et un entier i tel que $0 \leq i < n$.
Sortie : Le ième octet.

Donc les données sont $8n \leq 8n + \lfloor \log_2(i) \rfloor + 1 \leq 8n + \lfloor \log_2(n) \rfloor + 1$, et (s'il ne faut pas réécrire les données) le temps du calcul est borné par la lecture des $\lfloor \log_2(n) \rfloor + 1$ bits de i et les 8 bits du l'ième octet ($= \lfloor \log_2(n) \rfloor + 1 + 8$).

En pratique, comme il y a n octets écrits sur un disque dur ou en mémoire active, on peut supposer que n est borné par le nombre d'atomes dans l'univers (environ $10^{80} \approx 2^{266}$), donc le temps du calcul est borné par une constante ($= 266 + 8$).

Notation grand 'o'

Une mesure standard pour la complexité est donnée par les classes d'équivalence $O(f)$. Soit $f : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$ une fonction, on définit

$$O(f) = \{g : \mathbb{R}_{>0} \to \mathbb{R} \mid \exists\, c > 0 \text{ et } x_0 > 0 \text{ tel que } |g(x)| < cf(x)\ \forall\, x > x_0\}.$$

Si le temps $T(x)$ d'un algorithme est dans $O(f)$, on dit que la complexité d'un algorithme est dans $O(f)$.

Remarque. En analyse, on utilise une notation semblable dans les voisinages d'une élément (réel ou complexe) x_0 :

$$O(f) \text{ quand } x \to x_0 = \{g : \mathbb{R} \to \mathbb{R} \mid \exists\, c > 0 \text{ et } \varepsilon > 0 \text{ tel que } |g(x)| < c|f(x)| \text{ si } |x - x_0| \leq \varepsilon\},$$

qui, pour $x_0 = \infty$, correspond a notre définition.

Ordre partiel sur des classes. Pour f et g deux fonctions $\mathbb{R}_{>0} \to \mathbb{R}_{>0}$, on définit

$$O(f) \leq O(g) \text{ si } O(f) \subseteq O(g).$$

Par conséquent, pour tout réel $a > 0$, entier $k > 1$ et réel $c > 1$ on a :

$$O(1) = O(a) < O(\log(x)) < O(\log^k(x)) < O(x) < O(x^k) < \cdots < O(c^x).$$

Arithmétique des classes. Pour f et g deux fonctions $\mathbb{R}_{>0} \to \mathbb{R}_{>0}$, on définit

$$O(f) + O(g) = \{a + b \mid a \in O(f) \text{ et } b \in O(g)\},$$

$$O(f)O(g) = \{ab \mid a \in O(f) \text{ et } b \in O(g)\}.$$

Pour $f : \mathbb{R}_{>0} \to \mathbb{R}$ et $g : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$, on définit

$$f + O(g) = \{f + h \mid h \in O(g)\},$$

$$fO(g) = \{fh \mid h \in O(g)\}.$$

Ce définitions sont des cas spéciaux des définitions, pour des sous-ensembles S_1 et S_2 d'un groupe abélien G (ci-dessus l'ensemble des fonctions $f : \mathbb{R}_{>0} \to \mathbb{R}$) :

$$S_1 + S_2 = \{x + y \mid x \in S_1, y \in S_2\} \text{ et } S_1S_2 = \{xy \mid x \in S_1, y \in S_2\},$$
$$x + S_1 = \{x + y \mid y \in S_1\} \text{ et } xS_1 = \{xy \mid y \in S_1\}.$$

On définit également pour x_1 et x_2 dans G :

$$(x_1 + S_1) + (x_2 + S_2) = \{x_1 + x_2 + y_1 + y_2 \mid y_1 \in S_1, y_2 \in S_2\}$$
$$(x_1 + S_1)(x_2 + S_2) = \{(x_1 + y_1)(x_2 + y_2) \mid y_1 \in S_1, y_2 \in S_2\}.$$

On trouve très souvent une notation "=" asymétrique, que veut dire "$\in$" ou "$\leq$" selon le contexte.

Notation courante	Notation correcte
$f = O(g)$	$f \in O(g)$
$f = g + O(h)$	$f \in g + O(h)$
$O(f) = O(g)$	$O(f) \leq O(g)$

Cette notation courante implique des expressions asymétriques comme

$$x = O(x^2) \text{ et } O(x) = O(x^2), \text{ mais } x^2 \neq O(x) \text{ et } O(x^2) \neq O(x).$$

Notation petit 'o' et $\sim$

Pour une fonction $f : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$, on définit

$$o(f) = \{g : \mathbb{R}_{>0} \to \mathbb{R} \mid \forall\, \varepsilon > 0,\ \exists\, x_0 > 0 \text{ tel que } |g(x)| < \varepsilon f(x)\ \forall\, x > x_0\}.$$

La notation $f \sim g$ s'utilise pour deux fonctions telles que $\lim\limits_{x\to\infty} \dfrac{f(x)}{g(x)} = 1$.

Exercices

1. **a.** Montrer que g est dans $O(f)$ si et seulement si $\limsup\limits_{x\to\infty} \dfrac{|g(x)|}{f(x)} = c < \infty$.

b. Montrer que $O(f)$ est un espace vectoriel réel.

2. **a.** Montrer que g est dans $o(f)$ si et seulement si $\lim\limits_{x\to\infty} \dfrac{g(x)}{f(x)} = 0$.

b. Montrer que $o(f)$ est un sous-espace vectoriel réel de $O(f)$.

3. Soient f et g des fonctions positives.

a. Montrer que $O(g) \leq O(f)$ si et seulement si g est une fonction positive de $O(f)$.

b. Si $g \in O(f)$ est positive, alors $O(g) + O(f) = O(f)$.

4. Pour des fonctions $f_1, \ldots, f_t$ positives, et scalaires réels $\alpha_1, \ldots, \alpha_t$ positifs, montrer les identités

a. $\sum_{i=1}^t O(f_i) = O(\sum_{i=1}^t \alpha_i f_i)$ et en particulier, $\sum_{i=1}^t O(f_i) = O(\sum_{i=1}^t f_i)$.

b. $\prod_{i=1}^t O(f_i) = O(\prod_{i=1}^t f_i)$.

Conclure que $(O(f_1)+O(g_1))(O(f_2)+O(g_2)) = O((f_1+g_1)(f_2+g_2))$ (*cf.* exercice **6**).

5. Définir la somme $(f_1 + O(g_1)) + (f_2 + O(g_2))$ et démontrer que

$$(f_1 + O(g_1)) + (f_2 + O(g_2)) = (f_1 + f_2) + O(g_1 + g_2).$$

L'ensemble $f_i + O(g_i)$ porte quelle structure ?

6. Définir le produit $(f_1 + O(g_1))(f_2 + O(g_2))$ et démontrer que

$$\begin{aligned}(f_1 + O(g_1))(f_2 + O(g_2)) &\subseteq f_1 f_2 + O(f_1 g_2 + f_2 g_1 + g_1 g_2) \\ &= (f_1 + O(g_1))(f_2 + O(g_2)) + O(g_1 g_2).\end{aligned}$$

Lesquelles des fonctions f_1, f_2, g_1, g_2 doivent être positives ?
Si $g_1 \in O(f_1)$ ou $g_2 \in O(f_2)$, alors

$$(f_1 + O(g_1))(f_2 + O(g_2)) = f_1 f_2 + O(f_1 g_2 + f_2 g_1 + g_1 g_2),$$

et également si $f_1 \in O(g_1)$ où $f_2 \in O(g_2)$ (dans ce dernier cas pour des raisons triviales car $f_1 + O(g_1) = O(g_1)$ ou $f_2 + O(g_2) = O(g_2)$).

7. Trouver une expression pour la classe $\big(x + O(x^{1/2})\big)\big(x + O(\log(x))\big)^2$.

8. Montrer qu'il existe des fonctions f et $g : \mathbb{R}_{>0} \to \mathbb{R}$ telles que $g \notin O(f)$ et $f \notin O(g)$.
Indication. On peut construire des fonctions comme combinaison linéaire avec un terme dominant multiplié par des fonctions périodiques $\sin^2(x)$ et $\cos^2(x)$.

9. Répéter exercice **4** en remplaçant 'O' par 'o'.

10. Supposer que $f, g : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$. Montrer l'équivalence des expressions suivantes

a. $f \sim g$, **b.** $f - g \in o(g)$, **c.** $f/g \in 1 + o(1)$, **d.** $f/g \sim 1$.

Complexité des programmes

On donne un exercice qui sert comme un mini-texte de modélisation. On peut envisager de préparer un exposé court autour de cet exercice, en intégrant des expériences et experimentation en Python/Sage.

Exercice de modélisation. Soit donné les algorithmes ci-dessous, avec donnée d'entrée n.

a.

```
def runtime1(n):
    m = 0
    for i in range(n):
        m += 1
    return m
```

b.

```
def runtime2(n):
    m = 0
    for i in range(n):
        m += i
    return m
```

c.

```
def runtime3(n):
    m = 0
    for i in range(n):
        m *= 2
        m += 1
    return m
```

d.

```
def runtime4(n):
    m = 0
    for i in range(n):
        m *= 2
        m += i
    return m
```

Déterminer, pour chaque programme, une expression en forme de somme pour la valeur de sortie, la taille (en bits) de cette valeur, et la complexité de l'algorithme, en fonction de $\log_2(n)$. Vérifier que cette complexité correspond bien au temps de calcul à l'ordinateur.

Indication. Pour le programme `runtime4`, on peut formuler et démontre l'identité suivante pour l'expression de la somme représentant la valeur de sortie.

Lemme.

$$\sum_{i=0}^{n-1} i\, 2^{n-i-1} = 2^n - n - 1.$$