Complexité algorithmique

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 $x > T(x)$.

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 $\limsup\limits_{x\to\infty} \dfrac{|g(x)|}{f(x)} = 0$.

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

3. Soit f et g des fonctions positives. Montrer que g est dans $O(f)$ si et seulement si $O(g) \leq O(f)$.

4. Si $g \in O(f)$, alors $O(g) + O(f) = O(f)$.

5. Pour des fonctions $f_1, \ldots, f_r$ positives, et scalaires réels $\alpha_1, \ldots, \alpha_r$ positifs, montrer les identités

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

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

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 7).

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

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

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

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

Laquelles des fonctions f_1, f_2, g_1, g_2 doivent être positives ?
Si $g_1 \in O(f_1)$ où $g_2 \in O(f_2)$, 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 $f_2 + O(g_2) = O(g_2)$).

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

9. 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 $\cos^2(x)$.

10. Répéter exercice 5 en remplaçant 'O' avec 'o'.

11. Supposer que $f, g : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$.

a. Montrer que $f \sim g$ si et seulement si $o(f) = o(g)$.

b. Montrer que $f \sim g$ si et seulement si $f - g \in o(g)$.

Arithmétique et algorithmique

Résumé de la complexité des opérations élémentaires sur les entiers.

Algorithme	complexité
$m \pm n$	$O(\log(m) + \log(n))$
mn	$O(\log(m)\log(n))$
$n \bmod m$	$O(\log(m)\log(n/m))$

Remarque. La complexité $O(\log(n)\log(n/m))$ pour la réduction modulaire $n \bmod m$ est donnée dans Demazure [1, p.30], mais voir les exercices **7** et **8** ci-dessus.

Exercices

1. Soit $m = 100110111_2$ et $n = 10111_2$ (en représentation binaire). Trouver $m + n$ en précisant les étapes pour leur addition.

2. Répéter l'exercice précédent avec $m = 2^{16} - 1$ et $n = 1$ (cf. **3.b.** ci-dessous).

3. Exprimer la complexité de l'addition de m et n en fonction de la taille x des données (avec la notation $O(f)$), en particulier, démontrer que cette complexité est $O(x)$.

a. Plus précisement, décrire un algorithme et compter le nombre d'opérations binaires en termes de $\lfloor \log_2(m) \rfloor + 1$ et $\lfloor \log_2(n) \rfloor + 1$, pour le problème suivant :
Entrées : m et n entiers ($x = \lfloor \log_2(m) \rfloor + \lfloor \log_2(n) \rfloor + 2$)
Sortie : $m + n$

b. La syntaxe $m \mathrel{+}= n$ s'utilise pour un algorithme qui remplace m avec la valeur $m+n$. Supposer que m, n, et $m+n$ s'écrivent en représentation binaire. Est-il possible de décrire un algorithme pour $m \mathrel{+}= n$ avec complexité $O(\log(n))$?

4. Pour des polynômes f et g dans $\mathbb{F}_2[x]$, montrer qu'il existe un algorithme pour le calcul $f \mathrel{+}= g$ dans la classe $O(\deg(g))$.

5. Compter le nombre d'opérations pour la multiplication en utilisant la formule

$$\left(m = \sum_{i=0}^{k} m_i 2^i,\ n = \sum_{j=0}^{\ell} n_j 2^j\right) \longmapsto mn = \sum_{i=0}^{k+\ell} \left(\sum_{j=0}^{i} m_{i-j} n_j\right) 2^i$$

en termes de $k = \lfloor \log_2(m) \rfloor + 1$ et $\ell = \lfloor \log_2(n) \rfloor + 1$.

6. Démontrer que la complexité de la multiplication de deux nombres n, m de tailles dans $O(x)$, est dans $O(x^2)$, en utilisant la multiplication naïve avec retenues.

7. Décrire un algorithme pour la reduction $(n, m) \mapsto n \bmod m$ avec complexité

$$O(\log(m)\log(n/m)).$$

Indication. Étant donnés $n = n_r \dots n_1 n_0$ et $m = m_s \dots m_1 m_0$, la première étape est la soustraction $n \mathrel{-}= 2^{r-s} m$:

n_r	n_{r-1}	$\dots$	n_{r-s}	n_{r-s-1}	$\dots$	n_0
m_s	m_{s-1}	$\dots$	m_0	0	$\dots$	0
0	$*$	$\dots$	$*$	n_{r-s-1}	$\dots$	n_0

ou $n \mathrel{-}= 2^{r-s-1} m$:

n_r	n_{r-1}	$\dots$	n_{r-s-1}	n_{r-s-2}	$\dots$	n_0
0	m_s	$\dots$	m_0	0	$\dots$	0
0	$*$	$\dots$	$*$	n_{r-s-2}		n_0

Le nombre de coordonnées changées est au maximum $s+2$, et le résultat a au moins un bit de moins. En itérant, montrer qu'on obtient, après $O(\log(n/m))$ étapes, la complexité $O(\log(m)\log(n/m))$.

8. Décrire un algorithme pour la fonction $n \mapsto n \bmod 2$ et comparer sa complexité avec l'algorithme de l'exercice précédent pour n grand.

Exponentiation rapide

Un algorithme naïf pour l'exponentiation $a \mapsto a^k \bmod n$ utilise k multiplications dans $\mathbb{Z}/n\mathbb{Z}$. Un algorithme rapide permet de le faire avec $O(\log(k))$ multiplication.

Exercices

1. Montrer que les deux algorithmes suivants calculent le même résultat avec la même complexité. Décrire les résultats intérmediares dans les deux cas.

```
def exp_mod_L(a,k,n):
    # Données d'entree:
    # entiers a, k, n > 0
    # Sortie: a^k mod n
    b = 1
    k_seq = k.bits()
    for t in k_seq:
        if t == 1:
            b = (b * a) % n
        a = (a**2) % n
    return b
```

`exp_mod_L`$(3, 101, 101) == 3$

```
def exp_mod_R(a,k,n):
    # Données d'entree:
    # entiers a, k, n > 0
    # Sortie: a^k mod n
    b = 1
    k_seq = k.bits(); k_seq.reverse()
    for t in k_seq:
        b = (b**2) % n
        if t == 1:
            b = (b * a) % n
    return b
```

`exp_mod_R`$(3, 101, 101) == 3$

Algorithme pgcd – version soustractive

La version soustractive est effective et suffisante pour la démonstration de l'unicité du pgcd, mais elle n'est pas efficace.

```
def pgcd(m,n):
    # Entrees: entiers m, n > 0
    # Sorties: pgcd(m,n)
    # Version soustractive
    while n > 0:
        while m >= n:
            m = m - n
        (m,n) = (n,m)
    return m
```

Algorithme pgcd – version classique

La version "classique" de cet algorithme utilise un algorithme plus efficace pour la division euclidienne.

```
def pgcd(m,n):
    # Entrees: entiers m, n > 0
    # Sorties: pgcd(m,n)
    # Version classique
    while n > 0:
        (m,n) = (n,m%n)
    return m
```

Algorithme pgcd – version binaire

Les ordinateurs ont souvent des implémentations très efficaces des opérations de décalages binaires $n \mapsto \lfloor n/2^k \rfloor$ (`n >> k` ou `shift(n,k)`) ou ($n \mapsto 2^k n$ (`n << k` ou `shift(n,-k)`). L'algorithme `pgcd` classique considère les plus grands bits. En utilisant les décalages binaires l'algorithme binaire suivant opère sur les plus petits bits de m et n.

```
def pgcd(m,n):
    # Entrees: entiers m, n > 0
    # Sorties: pgcd(m,n)
    # Version binaire
    if m == 0: return n
    if n == 0: return m
    i = 0
    while m%2==0:
        m = m >> 1; i += 1
    j = 0
    while n%2==0:
        n = n >> 1; j += 1
    k = min(i,j)
    # m et n sont impairs
    if m < n:
        n = (n-m) >> 1
    else:
        m = (m-n) >> 1
    return pgcd(m,n)<<k
```

Exercices

1. Pour $m = 7$ et $n = 17$, déterminer la suite des valeurs (m, n) dans la version soustractive et la version classique de `pgcd(m,n)`.
2. Écrire les représentations binaires de la suite des valeurs (m, n) de l'exercice précédent.
3. Pour $m = 16 \cdot 7$ et $n = 8 \cdot 17$, déterminer la suite des valeurs (m, n) dans la version classique de `pgcd(m,n)`.
4. Supposer que la complexité de l'algorithme pour la division euclidienne (`m%n`) est dans $O(\log(n)\log(m/n))$. Comparer la complexité des algorithmes soustractif et classique pour le `pgcd`.
5. Pour $m = 7$ et $n = 17$, déterminer les répresentations binaires de m et n et calculer la suite des valeurs de (m, n) dans la version binaire de `pgcd(m,n)`.
6. Répéter l'exercice précédent avec $m = 16 \cdot 7$ et $n = 8 \cdot 17$.
7. Si l'algorithme pour décalage binaire (`n >> k` ou `shift(n,k)`) a complexité dans $O(\log(n) - k)$, déterminer la complexité de la version binaire de l'algorithme `pgcd`.

Algorithme d'Euclide étendu

Le `pgcd` étendu de deux entiers peut être calculé avec la fonction suivante (où la sortie de `quo_rem`(m,n) est le tuple $(q,r) = (\lfloor m/n \rfloor, m \bmod n)$ tel que $m = r + qn$) :

```
def pgcde(m,n):
    # Entrees: entiers m, n > 0
    # Sorties (r,u,v) tel que
    # r = pgcd(m,n) et que
    # r = u m + v n.
    if m < n:
        (r0, u0, v0) = (n, 0, 1)
        (r1, u1, v1) = (m, 1, 0)
    else:
        (r0, u0, v0) = (m, 1, 0)
        (r1, u1, v1) = (n, 0, 1)
    while r1 > 0:
        (q1, r2) = r0.quo_rem(r1)
        (u2, v2) = (u0 - u1*q1, v0 - v1*q1)
        (r0, u0, v0) = (r1, u1, v1)
        (r1, u1, v1) = (r2, u2, v2)
    return (r0, u0, v0)
```

Exercices

1. Dans l'algorithme `pgcde`, vérifier que $r_i = u_i m + v_i n$, pour chaque $i = 0$ et 1, à chaque itération de la boucle `while` et que cette identité reste valide. En particulier, vérifier que dans chaque itération de la boucle `while`, on a

$$\begin{pmatrix} r_0 & u_0 & v_0 \\ r_1 & u_1 & v_1 \end{pmatrix} \longleftarrow \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix} \begin{pmatrix} r_0 & u_0 & v_0 \\ r_1 & u_1 & v_1 \end{pmatrix} = \begin{pmatrix} r_1 & u_1 & v_1 \\ r_2 & u_2 & v_2 \end{pmatrix},$$

et

$$\begin{pmatrix} r_0 & u_0 & v_0 \\ r_1 & u_1 & v_1 \end{pmatrix} \begin{bmatrix} -1 \\ m \\ n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

2. Déterminer la suite des tuples (r_0, u_0, v_0) dans le calcul du `pgcde`$(28, 34)$.

3. Soit n et a des entier premiers entre eux. Montrer comment utiliser le `pgcde` pour trouver l'inverse de la classe de a dans $\mathbb{Z}/n\mathbb{Z}$.

Indication. Utiliser la reduction de l'expression $1 = ua + vn$ modulo n.

4. Trouver l'inverse de 7 modulo 17.

Les théorèmes chinois et de Fermat

Théorème (Théorème chinois.)**.** *Soient p et q des entiers premiers entre eux et n leur produit. Pour tout couple (x_1, x_2) il existe un entier x, uniquement determiné modulo n, tel que $x \equiv x_1 \bmod p$ et $x \equiv x_2 \bmod q$.*

Autrement dit, l'application

$$\mathbb{Z}/n\mathbb{Z} \longrightarrow \mathbb{Z}/p\mathbb{Z} \times \mathbb{Z}/q\mathbb{Z}$$

donnée par $x \mapsto (x \bmod p, x \bmod q)$ est une bijection. Comme elle est un homomorphisme d'anneaux,

— elle est un isomorphisme d'anneaux ;
— elle est un isomorphisme de groupes additifs ;
— elle induit un isomorphisme de groupes multiplicatifs $\mathbb{Z}/n\mathbb{Z}^* \longrightarrow \mathbb{Z}/p\mathbb{Z}^* \times \mathbb{Z}/q\mathbb{Z}^*$.

L'algorithme pour l'inverse de l'homomorphisme $\mathbb{Z}/n\mathbb{Z} \to \mathbb{Z}/p\mathbb{Z} \times \mathbb{Z}/q\mathbb{Z}$ est un conséquence de l'algorithme `pgcde`.

Théorème (Algorithme CRT)**.** *Soient p et q des entiers premier entre eux, n leur produit, et u et v des entiers tel que $1 = up+vq$. Pour tout couple (x_1, x_2) l'entier $x = x_1+up(x_2-x_1)$ satisfait $x \equiv x_1 \bmod p$ et $x \equiv x_2 \bmod q$.*

Théorème (Fermat)**.** *Soit p un nombre premier. On a $a^p \equiv a \bmod p$ pour tout entier a, et $a^{p-1} \equiv 1 \bmod p$ pour tout entier a premier à p.*

Dans un language plus moderne, $\mathbb{Z}/p\mathbb{Z}^*$ est un groupe cyclique d'ordre $p-1$.

Notation. Si p est premier, on écrit $\mathbb{F}_p$ pour le corps $\mathbb{Z}/p\mathbb{Z}$.

Exercices

1. Si $1 = up + vq$, montrer que $x_1 + up(x_2 - x_1) = x_2 + vq(x_1 - x_2)$, et donner une demonstration de la version algorithmique du théoréme chinois.

2. Trouver x tel que $(x \bmod 3, x \bmod 5, x \bmod 7) = (1, 2, 3)$.

3. Soit $\varphi(n)$ le cardinal de $\mathbb{Z}/n\mathbb{Z}^*$. Pour un entier $n = pq$, avec p et q premier entre eux, montrer que $\varphi(n) = \varphi(p)\varphi(q)$.

4. Dans l'anneau $\mathbb{F}_p[x]$, démontrer que

$$\prod_{a=0}^{p-1}(x - a) = x^p - x.$$

5. Trouver la factorisation de $x^4 - x$ dans $\mathbb{F}_2[x]$.

6. Soit K un corps fini à $q\,(= p^r)$ éléments. Démontrer que

a. $\alpha^{q-1} = 1$ pour tout $\alpha \in K^*$, et

b. dans $K[x]$, on a l'identité $\displaystyle\prod_{\alpha \in K}(x - \alpha) = x^q - x$.

Références

[1] M. Demazure, *Cours d'algébre, primalité, divisibilité, codes*, Cassini, 1997.

[2] M. Hindry, *Arithmétique*, Calvage & Mounet, 2008.

[3] V. Shoup, *A Computational Introduction to Number Theory and Algebra*, Cambridge University Press, 2005.

[4] R. P. Brent, P. Zimmermann, *Modern Computer Arithmetic*, Cambridge University Press, 2010.