ALGORITMOS EN ÁLGEBRA LINEAL

Notas de curso (UBA - 2do cuatrimestre de 2005)

http://atlas.mat.ub.es/personals/sombra/curso.html

Michelle Schatzman

Martín Sombra

Institut Camille Jordan (Mathématiques), Université de Lyon 1 ; 43 Bd. du 11 Novembre 1918, 69622 Villeurbanne Cedex, Francia

Universitat de Barcelona, Departament d'Àlgebra i Geometria ; Gran Via 585, 08007 Barcelona, España

CHAPITRE 2

La transformée de Fourier rapide

1. Complexité d'algorithmes

Dans une première approximation, la complexité d'un algorithme est une formalisation de la notion de vitesse d'exécution. Un algorithme est un procédé effectif admettant comme *entrée* (en anglais : *input*) des mots finis (par exemple une suite finie de 0 et de 1) qui ou bien finit au bout d'un certain temps d'exécution donnant comme résultat une *sortie* (en anglais : *output*), ou bien ne finit pas. Ceci n'est nullement une définition précise d'algorithme, c'est seulement l'idée mais ça suffit à nos besoins.

Pour un mot a, on notera

$$\tau(a) \in \mathbb{N}$$

sa *taille* ou *longueur*. Bien entendu, cette notion présuppose un alphabet ou codification de base. La *complexité* $c_A(E)$ d'un algorithme A sur une entrée E est le nombre d'opérations réalisées par A sur cette entrée. La *complexité* de A tout court est une fonction

$$\mathcal{C}_A : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$$

telle que pour $\tau \in \mathbb{N}$, $\mathcal{C}_A(\tau)$ est le nombre maximal d'opérations réalisées par A sur une entrée de taille τ.

Considérons comme exemple la somme en binaire de deux entiers $a, b \in \mathbb{N}$ suivant l'algorithme qu'on nous a appris à l'école. La taille de l'entrée est la somme de la longueur de la représentation en binaire de a et de b :

$$\tau(a, b) = \tau(a) + \tau(b) = \lfloor \log_2 a \rfloor + \lfloor \log_2 b \rfloor + 2.$$

Soit $c := a + b$ et $s := \tau(a)$, $t := \tau(b)$ et $u := \tau(c) \leq \max(s, t) + 1$ et écrivons les représentations binaires

$$a = \overline{a_s a_{s-1} \cdots a_1 a_0}, \quad b = \overline{b_t b_{t-1} \cdots b_1 b_0}, \quad c = \overline{c_u c_{u-1} \ldots c_1 c_0};$$

schématiquement l'algorithme est décrit par la figure

$$\begin{array}{cccccccc} & & r_{s+1} & r_s & r_{s-1} & \cdots & r_1 & \\ & & & a_s & a_{s-1} & \cdots & a_1 & a_0 \\ + & b_t & b_{t-1} & \cdots & \cdots & \cdots & b_1 & b_0 \\ \hline \\ c_u & c_{u-1} & \cdots & \cdots & \cdots & \cdots & c_1 & c_0 \end{array}$$

où r_j est la j-ème *retenue*. En pseudocode :

Algorithme Σ :

Entrée : $a_s, \ldots, a_0; b_t, \ldots, b_0 \in \{0, 1\}$ **;**
Sortie : $c_u, \ldots, c_0 \in \{0, 1\}$**.**

For $k = 0$ **to** $v := \max(s, t)$ **do**
$r_{k+1} \cdot 2 + c_k \leftarrow a_k + b_k + r_k$ **od ;**
if $r_{v+1} = 1$ **then** $c_{v+1} \leftarrow 1$ **;**
end.

La taille de la sortie est $\leq \tau(a,b)$, donc il y a des chances d'avoir un algorithme linéaire : la taille de la sortie est une minoration triviale (mais souvent significative) de la complexité d'un algorithme. Pour le calcul de la complexité on a

(1) $3(v+1)+1$ lectures en mémoire ;
(2) $2(v+1)$ opérations binaires (sommes dans $\mathbb{Z}$ des 0 et des 1) ;
(3) 1 décision ($r_{v+1} =^{?} 1$) ;
(4) $2(v+1)+1$ écritures en mémoire.

Il est difficile de comparer le coût relatif de chacune de ces opérations. Comment les pondérer ? La pratique de l'analyse numérique et du calcul symbolique est de ne compter que les opérations binaires ($+,-,\cdot$ dans $\mathbb{Z}$ des 0 et des 1) c'est-à-dire le point (2) ci-dessus, dans l'hypothèse (ou l'espoir !) que ceci soit la partie la plus lourde dans l'algorithme, et par conséquent son coût domine celui des autres opérations. Avec cette convention $c_\Sigma(a,b) \leq 2(v+1) = 2\max(\tau(a),\tau(v)) \leq 2\tau(a,b)$ donc

$$\mathcal{C}_\Sigma(\tau) \leq 2\tau.$$

Cette convention nous permet de généraliser la notion d'algorithme. Soit $\mathbb{F}$ une structure algébrique quelconque (groupe, anneau, corps) et faisons comme si les opérations de $\mathbb{F}$ sont effectives, ce qui n'est pas toujours le cas (par exemple si $\mathbb{F} = \mathbb{R}$). On considérera des pseudo-algorithmes de type algébrique, sur des machines capables de manipuler des éléments de $\mathbb{F}$ (faire des opérations arithmétiques, les stocker en mémoire, etc). La *complexité*

$$\mathcal{C}_A(\mathbb{F};\tau)$$

de A relative à $\mathbb{F}$ sera définie comme le nombre maximal d'opérations arithmétiques de $\mathbb{F}$ (sommes, différences, multiplications et divisions dans le cas d'un corps) réalisées par A sur une entrée de taille τ.

Ces pseudo-algorithmes sont parfois appelés *machines BSS sur* $\mathbb{F}$, à cause du travail de L. Blum, M. Shub et S. Smale sur le sujet [**2**]. On peut vérifier que les algorithmes ou machines de Turing standard correspondent à des algorithmes sur $\mathbb{F}_2 = \{0,1\}$. Notons que si $\mathbb{F}$ es effectif (par exemple si $\mathbb{F} = \mathbb{Q}$), le pseudo-algorithme A est un véritable algorithme, et sa complexité totale doit tenir compte de la complexité $\mathcal{C}_A(\mathbb{F};\tau)$ relative à $\mathbb{F}$ *et* du coût binaire de chacune des opérations effectuées.

DÉFINITION 1.1. Un algorithme A est *linéaire/polynomial/exponentiel* s'il existe $\nu > 0$ tel que

$$\mathcal{C}_A(\tau) = O(\tau) \quad / \quad O(\tau^\nu) \quad / \quad O(\exp(\tau^\nu))$$

respectivement. La classe composée de tous les algorithmes linéaires/polynomiaux/exponentiels est notée par L/P/Exp respectivement.

On a

$$\mathrm{L} \subset \mathrm{P} \subset \mathrm{Exp}.$$

Grosso modo, cette classification est censée classifier les algorithmes en optimaux/acceptables/intractables. On a $\Sigma \in \mathrm{L}$; comme on vient de signaler ceci est le mieux à quoi on peut s'attendre. Mais en fait, on voudra toujours avoir des algorithmes linéaires (ou quasi-linéaires) : chaque fois que la puissance des ordinateurs est améliorée, la portée d'un algorithme linéaire est améliorée du même facteur. Un algorithme quadratique ou d'ordre supérieur n'est pas tellement sensible à l'amélioration des technologies (sa portée s'accroît comme la racine carrée de la puissance de l'ordinateur).

L'algorithme *de multiplication longue* est l'exemple typique d'algorithme quadratique : schématiquement

$$\begin{array}{cccccccccc}
 & & & & a_s & a_{s-1} & \cdots & a_1 & a_0 \\
\times & & b_t & b_{t-1} & \cdots & \cdots & & b_1 & b_0 \\
\hline
 & & & & b_0\cdot(a_s & a_{s-1} & \cdots & a_1 & a_0) \\
 & & & b_1\cdot(a_s & a_{s-1} & \cdots & a_1 & a_0) & \\
 & & \cdots & \cdots & \cdots & & & & \\
b_t\cdot(a_s & a_{s-1} & \cdots & a_1 & a_0) & & & & \\
\hline
d_v & d_{v-1} & \cdots & \cdots & \cdots & \cdots & \cdots & d_1 & d_0
\end{array}$$

donc on a au moins $(s+1)(t+1)$ multiplications binaires ; la complexité totale est estimée en $O(\tau^2)$.

Comme dernière illustration, considérons la factorisation des entiers, pour laquelle on ne connaît pas d'algorithme polynomial : pour $a \in \mathbb{N}$ donné, il s'agit de trouver les nombres premiers $p_1, \ldots, p_k$ et les exposants $e_1, \ldots, e_k \in \mathbb{N}$ tels que

$$a = p_1^{e_1} \cdots p_k^{e_k},$$

ou ce qui est équivalent, de trouver un diviseur premier p de a. L'algorithme naïf consiste à considérer successivement les candidats à diviseur $d \geq 2$ et de tester si $d|a$. Si oui, on déclare $p \leftarrow d$ le nombre premier en question. Si ce n'est pas le cas et si $d < \lfloor\sqrt{a}\rfloor$, on continue avec $d+1$ et ainsi de suite. S'il n'existe pas $d < \lfloor\sqrt{a}\rfloor$ tel que $d|a$ on déclare a premier. Cet algorithme a une complexité de type

$$a^{1/2},$$

or cette quantité est *exponentielle* en la taille $\lfloor \log_2 a \rfloor + 1$ de l'entrée ; on a

$$\mathcal{C}(\tau) = O(e^{\tau/2}).$$

Par contre, on peut *deviner* un nombre premier p tel que $p|a$ en temps polynomial, par un algorithme non-déterministe.

DÉFINITION 1.2. Un algorithme est *polynomial non-déterministe* si toute instance positive peut se vérifier en temps polynomial. La classe des algorithmes polynomiaux non-déterministes est notée par NP.

Évidemment P $\subset$ NP. La célèbre conjecture P $\neq$ NP est l'enjeu d'un des prix de US\$ 10^6 de la Fondation Clay aux États Unis.

EXERCICE 2.1. ◁ Soient $a, b \in \mathbb{N}$ deux entiers de taille $\tau(a), \tau(b) \leq \tau$.

(1) Estimer en $O(\tau^2)$ la complexité de $a \cdot b$ par l'algorithme de multiplication longue.

(2) Estimer en $O(\tau^2)$ la complexité de la division avec reste

$$a = q\,b + r \quad , \qquad 0 \leq r \leq b-1.$$

En déduire un algorithme pour le calcul du pgcd(a, b) avec complexité binaire $O(\tau^3)$.

▷

EXERCICE 2.2. ◁ Soit $n \geq 1$ et considérons

$$\mathbb{Z}_n := \mathbb{Z}/(n\mathbb{Z}) = \{\overline{0}, \overline{1}, \ldots, \overline{n-1}\}$$

l'anneau de congruences modulo n. Montrer que les versions "modulo n" des algorithmes classiques pour l'addition et la multiplication ont une complexité $O(\log_2(n))$ y $O(\log^2(n))$ respectivement. ▷

2. Transformée de Fourier discrète (TFD) et rapide (TFR)

La transformée de Fourier rapide est un algorithme pour le calcul de la transformée de Fourier discrète d'ordre N en temps $O(N\log(N))$. Cet algorithme remarquable fut proposé par J.W. Cooley et J.W. Tukey in 1965 [**3**] mais fut apparemment découvert avant par Runge et König en 1924 et semble-t-il aussi connu par Gauss. Les références pour cette section sont [**4**] et [**1**].

2.1. La transformée de Fourier discrète. Les coefficients de Fourier d'une fonction continue $f : \mathbb{R} \to \mathbb{R}$ de période 1 sont

$$\widehat{f}(k) = \int_0^1 f(x)e^{-2i\pi kx}\,\mathrm{d}x \quad , \quad k \in \mathbb{Z}.$$

L'inversion est donnée par la série de Fourier

$$f(x) = \sum_{k\in\mathbb{Z}} \widehat{f}(k)e^{2i\pi kx}$$

valable pour f suffisamment régulière (par exemple si $f \in \mathcal{C}^2(\mathbb{R}/\mathbb{Z})$). Soit $N \in \mathbb{N}$ et posons

$$\Omega_N = \left\{ \frac{j}{N} : j = 0, \dots, N-1 \right\} \subset [0,1];$$

la discrétisation de l'intégrale est

$$\widehat{f}_N(k) = \frac{1}{N} \sum_{x\in\Omega_N} f(x)e^{-2i\pi kx} \quad , \quad k \in \mathbb{Z}.$$

Notons que $k \mapsto U_k$ est périodique de période N car

$$\widehat{f}_N(k) = \frac{1}{N} \sum_{x\in\Omega_N} f(x)e^{-2i\pi(k+N)x} = \frac{1}{N} \sum_{x\in\Omega_N} f(x)e^{-2i\pi kx} = \widehat{f}_N(k),$$

donc cette formule produit au plus N nombres complexes différents. La *transformée de Fourier discrète (TFD)* (en anglais : *discrete Fourier transform (DFT)*) est l'opérateur $F_N : \mathbb{C}^N \to \mathbb{C}^N$ tel que pour $u = (u_1, \dots, u_N) \in \mathbb{C}^n$

$$U_k = (F_N u)_k = \sum_{j=0}^{N-1} u_j e^{-2i\pi kj/N} \quad , \quad 0 \le k \le N-1.$$

Posons $\omega := e^{-2i\pi/N}$, la matrice de la TFD est

$$F_N = [\omega_{i,j}]_{0\le i,j\le N-1} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{bmatrix}.$$

Cette matrice est symétrique (mais pas hermitienne !), et unitaire à un facteur scalaire près :

LEMME 2.1. *Pour tout $N \in \mathbb{N}$,*

$$F_N^* F_N = \overline{F}_N F_N = N\mathbf{1}_N.$$

DÉMONSTRATION.

$$(F_N)_{i,k} = \sum_{j=0}^{N-1} \omega^{-ji}\omega^{jk} = \sum_{j=0}^{N-1} \omega^{(k-i)j}.$$

Si $i = k$ alors

$$(F_N)_{i,k} = \sum_{j=0}^{N-1} 1 = N;$$

sinon $\omega^{k-i} \neq 1$ et

$$(F_N)_{i,k} = \sum_{j=0}^{N-1} \omega^{(k-i)j} = (1-\omega^{k-i})^{-1}(1-(\omega^{k-i})^N) = 0.$$

La seconde identité se suit de la première du fait que F_N est symétrique. □

La formule d'inversion est donc

$$f(x) = \sum_{k=0}^{N-1} \widehat{f}_N(k) e^{2i\pi kx} \quad , \quad x \in \Omega_N,$$

en analogie avec le cas continu.

2.2. Algorithme TFR. La TFD est une multiplication matrice-vecteur $F_N u$, donc *a priori* nécessite de N^2 multiplications complexes. La *transformée de Fourier rapide (TFR)* (en anglais : *fast Fourier transform (FFT)*) a pour but de calculer la TFD en complexité $O(N\log(N))$. C'est notre premier exemple de matrice structurée pour laquelle on sait accélérer les calculs.

Présentons l'idée pour le cas $N = 4$. Écrivons les matrices de F_2 et de F_4 explicitement

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad , \quad F_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}.$$

Donc

$$\begin{aligned} V_0 &= v_0 + v_1, \\ V_1 &= v_0 - v_1; \end{aligned}$$

et

$$\begin{aligned} U_0 &= u_0 + u_1 + u_2 + u_3, \\ U_1 &= u_0 - iu_1 - u_2 + iu_3, \\ U_2 &= u_0 - u_1 + u_2 - u_3, \\ U_3 &= u_0 + iu_1 - u_2 - iu_3. \end{aligned}$$

Si l'on réorganise les lignes de F_4 on obtient

$$\begin{aligned} \begin{bmatrix} U_0 \\ U_2 \end{bmatrix} &= \begin{bmatrix} u_0 + u_1 + u_2 + u_3 \\ u_0 - u_1 + u_2 - u_3 \end{bmatrix} = F_2 \begin{bmatrix} u_0 + u_2 \\ u_1 + u_3 \end{bmatrix}, \\ \begin{bmatrix} U_1 \\ U_3 \end{bmatrix} &= \begin{bmatrix} u_0 - iu_1 - u_2 + iu_3 \\ u_0 + iu_1 - u_2 - iu_3 \end{bmatrix} = F_2 \begin{bmatrix} 1 & \\ & -i \end{bmatrix} \begin{bmatrix} u_0 - u_2 \\ u_1 - u_3 \end{bmatrix}. \end{aligned}$$

On a trouvé des auto-similarités nous permettant factoriser F_4 en termes de deux copies de F_2 et d'une matrice diagonale.

Présentons les détails de la TFR pour le cas général. Supposons $N = 2M$ pair, alors pour $k = 2\ell$ pair aussi on a

$$\begin{aligned} U_k &= \sum_{j=0}^{N-1} \omega^{(2\ell)j} u_j \\ &= \sum_{j=0}^{M-1} (\omega^2)^{\ell j} u_j + \omega^{2\ell M} \sum_{j=0}^{M-1} (\omega^2)^{\ell j} u_{M+j} \\ &= \sum_{j=0}^{M-1} (\omega^2)^{\ell j} (u_j + u_{M+j}). \end{aligned}$$

Pour $k = 2\ell + 1$ impair

$$\begin{aligned} U_k &= \sum_{j=0}^{N-1} \omega^{(2\ell+1)j} u_j \\ &= \sum_{j=0}^{M-1} (\omega^2)^{\ell j} (\omega^j u_j) + \omega^{(2\ell+1)M} \sum_{j=0}^{M-1} (\omega^2)^{\ell j} (\omega^j u_{M+j}) \\ &= \sum_{j=0}^{M-1} (\omega^2)^{\ell j} (\omega^j (u_j - u_{M+j})). \end{aligned}$$

Notons

$$u_I := \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{M-1} \end{bmatrix} \quad , \quad u_{II} := \begin{bmatrix} u_M \\ u_{M+1} \\ \vdots \\ u_{N-1} \end{bmatrix},$$

et

$$U_{\text{pair}} = \begin{bmatrix} U_0 \\ U_2 \\ \vdots \\ U_{N-1} \end{bmatrix} \quad , \quad U_{\text{impair}} = \begin{bmatrix} U_1 \\ U_3 \\ \vdots \\ U_{N-1} \end{bmatrix}.$$

Posons $D_M = \text{diag}(\omega^j : 0 \leq j \leq M-1)$; alors

$$\begin{bmatrix} U_{\text{pair}} \\ U_{\text{impair}} \end{bmatrix} = \begin{bmatrix} F_M & \\ & F_M \end{bmatrix} \cdot \begin{bmatrix} \mathbf{1}_M & \\ & D_M \end{bmatrix} \cdot \begin{bmatrix} \mathbf{1}_M & \mathbf{1}_M \\ \mathbf{1}_M & -\mathbf{1}_M \end{bmatrix} \cdot \begin{bmatrix} u_I \\ u_{II} \end{bmatrix}. \tag{1}$$

La permutation σ_N des lignes donnant

$$\begin{bmatrix} U_{\text{pair}} \\ U_{\text{impair}} \end{bmatrix} \mapsto \begin{bmatrix} U_I \\ U_{II} \end{bmatrix}$$

est définie par

$$\sigma_N(k) = \begin{cases} 2k & \text{si } 0 \leq k \leq M-1, \\ 2(k-M)+1 & \text{si } M \leq k \leq 2M-1; \end{cases}$$

on a donc la factorisation

$$F_N = P_{\sigma_N} \cdot \begin{bmatrix} F_M & \\ & F_M \end{bmatrix} \cdot \begin{bmatrix} \mathbf{1}_M & \\ & D_M \end{bmatrix} \cdot \begin{bmatrix} \mathbf{1}_M & \mathbf{1}_M \\ \mathbf{1}_M & -\mathbf{1}_M \end{bmatrix}.$$

Ainsi $F_N u$ peut se calculer avec deux TFD d'ordre $N/2$. Si N est une puissance de 2, on peut continuer jusqu'à se réduire au cas trivial $F_1 = 1$.

On peut représenter graphiquement la TFR par un diagramme de papillons et flèches horizontales, *voir* le cas $N = 8$ (avec $\omega = e^{-i/4}$) ci-dessous. Les papillons représentent l'application

$$\begin{bmatrix} u_I \\ u_{II} \end{bmatrix} \mapsto \begin{bmatrix} u_I + u_{II} \\ u_I - u_{II} \end{bmatrix}.$$

Les flèches horizontales représentent le transfert d'un coefficient, éventuellement multiplié par un scalaire indiqué sur la flèche. La permutation à la fin est

$$(\sigma_8 \circ (\sigma_4, \sigma_4))^{-1}.$$

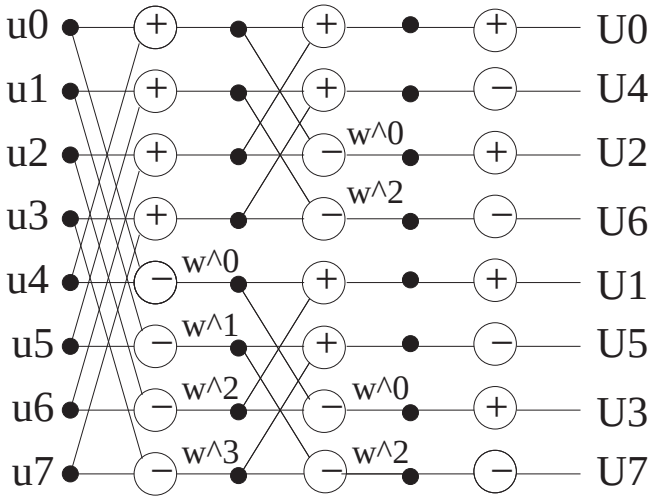

FIG. 1. Papillons TFR

THÉORÈME 2.2. *Soit $N = 2^n$, alors la TFR évalue $F_N u$ en au plus $1,5N\log_2(N)$ opérations de $\mathbb{C}$ (additions/soustractions et multiplications par ω^j avec $0 \leq j \leq N-1$).*

DÉMONSTRATION. Notons

$$t_n := \mathcal{C}_{TFR}(\mathbb{C}; 2^n)$$

la complexité de la TFR en dimension N. Pour obtenir $u_I + u_{II}$ et $u_I - u_{II}$ on fait N additions/soustractions et pour la multiplication $D_M(u_I - u_{II})$ on fait $M = N/2$ multiplications de $\mathbb{C}$; puis pour les TFR en dimension M on fait $2t_{n-1}$ opérations de $\mathbb{C}$. On a l'inégalité

$$t_n \leq 2t_{n-1} + \frac{3}{2}2^n.$$

Par induction et le fait que $t_0 = 0$ on trouve

$$t_n \leq \frac{3}{2}n2^n = 1,5N\log_2(N).$$

□

2.3. Convolution de vecteurs. Il y a une relation étroite entre la TFD et l'évaluation et l'interpolation de polynômes. Pour $u = (u_0, \dots, u_{N-1}) \in \mathbb{C}^N$ considérons le polynôme

$$P_u = \sum_{j=0}^{N-1} u_j x^j.$$

Alors

$$F_N u = \begin{bmatrix} P_u(1) \\ P_u(\omega) \\ \vdots \\ P_u(\omega^{N-1}) \end{bmatrix}$$

est l'évaluation de P_u en les puissances successives de ω, tandis que

$$\frac{1}{N} F_N^* \begin{bmatrix} v_0 \\ v_1 \\ \vdots \\ v_{N-1} \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_{N-1} \end{bmatrix}$$

donne les coefficients de l'unique polynôme P de degré au plus $N-1$ tel que $P(\omega^j) = v_j$. Donc on identifie la TFD avec l'application évaluation

$$F_N : \mathbb{C}[x]_{\leq N-1} \to \mathbb{C},$$

et son inverse $\frac{1}{N} F_N^*$ avec l'interpolation.

Pour des vecteurs $u, v \in \mathbb{C}^N$, la *convolution* est définie comme

$$u * v = \left[\sum_{j=0}^{N-1} u_j v_{k-j} : 0 \leq k \leq 2N-1 \right]^T \in \mathbb{C}^{2N}.$$

On a

$$P_{u*v} = P_u \cdot P_v.$$

Une des principales propriétés de la TFD est qu'elle traduit la convolution en multiplication :

THÉORÈME 2.3. *Soient $u, v \in \mathbb{C}^N$, alors*

$$u * v = \frac{1}{2N} F_{2N}^* (F_{2N} u \cdot F_{2N} v).$$

DÉMONSTRATION. On a $P_{u*v}(\omega^j) = P_u(\omega^j) \cdot P_v(\omega^j)$ donc

$$F_{2N}(u * v) = F_{2N} u \cdot F_{2N} v;$$

le résultat s'obtient en appliquant l'inverse de F_{2N} à cette identité. □

Joint à la TFR, ceci entrane un algorithme rapide pour multiplier sur $\mathbb{C}$ deux polynômes

$$f = f_0 + \cdots + f_{N-1} x^{N-1} \quad , \quad g = g_0 + \cdots + g_{N-1} x^{N-1} \in \mathbb{C}[x].$$

COROLLAIRE 2.4. *Soient $f, g \in \mathbb{C}[x]$ deux polynômes de degré au plus $N-1$ (N étant une puissance de 2), alors $f \cdot g$ peut se calculer en $4,5N \log_2(N) + O(N)$ opérations de $\mathbb{C}$.*

DÉMONSTRATION. Suivant le théorème de convolution, h se calcule avec 3 TFD en dimension $2N$, N produits et une division, la complexité est

$$3 \times 1,5(2N) \log_2(2N) + N + 1 = \frac{9}{2} N \log_2(N) + O(N).$$

□

Notons que l'algorithme naïf

$$h = f \cdot g = \sum_{k=0}^{2N-1} \left(\sum_{j=0}^{N-1} f_j g_{k-j} \right) x^k$$

comprend $2N^2 + O(N)$ opérations de $\mathbb{C}$.

2.4. La TFR en binaire. Les constructions précédentes se généralisent à un anneau commutatif A quelconque à la place de $\mathbb{C}$, muni d'une racine principale d'ordre N de 1, qui jouera le rôle de ω.

Définition 2.5. Soit A un anneau commutatif et $N \in \mathbb{N}$. Un élément $\omega \in A$ est une racine N-ème *principale* de l'unité si

(1) $\omega^N = 1$;

(2) $\omega^M - 1$ n'est pas un diviseur de 0, pour $1 \leq k \leq N-1$.

Notons que dans le cas d'un *domaine* A, cette définition coïncide avec celle de racine N-ème *primitive* de l'unité. La condition (2) est nécessaire pour que le lemme 2.1 reste valable (et avec la même démonstration) dans ce cadre plus général.

On appliquera cette idée aux anneaux de congruences de type

$$A_N = \mathbb{Z}_{2^N+1} := \mathbb{Z}/(2^N+1)\mathbb{Z}$$

pour $N = 2^n$. Le lemme suivant nous garantit l'existence de racines principales :

Lemme 2.6. *Soit $N = 2^n$, alors $2 \in \mathbb{Z}_{2^N+1}$ est une racine $2N$-ème principale de l'unité.*

Démonstration. On a

$$2^{2N} \equiv (-1)^2 = 1 \pmod{2^N+1}$$

donc la première condition est vérifiée. Pour chaque $1 \leq M \leq 2N-1$, la condition que $2^M - 1$ ne soit pas un diviseur de 0 dans $\mathbb{Z}_{2^N+1}$ est équivalente à ce que

$$\gcd(2^M - 1, 2^N + 1) = 1. \tag{2}$$

Maintenant pour tout $a \geq 2$ et $k, \ell \geq 1$ on a

$$\gcd(a^k - 1, a^\ell - 1) = a^{\gcd(k,\ell)} - 1;$$

en appliquant cela à $a = 2$, $k = M$ et $\ell = 2N$ on obtient

$$\gcd(2^M - 1, 2^N + 1) | \gcd(2^M - 1, 2^{2N} - 1) = 2^{\gcd(M,2N)} - 1 | 2^N - 1$$

car $\gcd(M, 2N) | N$ puisque $M < 2N$ et que $2N$ est une puissance de 2. Donc

$$\gcd(2^M - 1, 2^N + 1) | 2 = (2^N + 1) - (2^N - 1)$$

et on en déduit $\gcd(2^M - 1, 2^N + 1) = 1$ car $2^M - 1$ est impair. □

Ceci nous permet de construire la TFD sur $\mathbb{Z}_{2^{N/2}+1}$ pour $\omega = 2$ racine principale d'ordre $N = 2^n$; par la suite on estime la complexité binaire de la TFR correspondante.

Théorème 2.7. *Soit $N = 2^n$, alors pour $u \in \mathbb{Z}_{2^{N/2}+1}$ la TFR évalue $F_N u$ en au plus $1,5N^2 \log_2(N) + O(N \log_2(N))$ opérations binaires.*

DÉMONSTRATION. La TFR calcule $F_N u$ en $1,5N\log(N)$ opérations de $\mathbb{Z}_{2^{N/2}+1}$ de type addition/soustraction et multiplication par 2^k pour $0 \leq k \leq N-1$.

Chaque addition/soustraction coûte $2(N/2+O(1)) = N+O(1)$ opérations binaires (exercice (2.2). En outre, tout élément $a \in \mathbb{Z}_{2^{N/2}+1}$ est représenté par au plus N bits et peut donc s'écrire comme

$$a = \sum_{j=0}^{N/2} a_j 2^j$$

avec $a_j = 0$ ou 1. Notons $r_{N/2}(j+k)$ le reste de diviser $j+k$ par $N/2$, alors

$$2^k \cdot a = \sum_{j=0}^{N/2} \pm a_j 2^{r_{N/2}(j+k)}$$

est un déplacement suivi d'un certain changement de signe. On peut donc écrire $2^k \cdot a$ comme différence de deux éléments de $\mathbb{Z}_{2^{N/2}+1}$, ce qui peut se calculer aussi en complexité $N+O(1)$. La complexité totale est

$$(1,5N\log(N))(N+O(1)) = 1,5N^2\log_2(N) + O(N\log_2(N)).$$

□

La même estimation s'applique au calcul de l'inverse $N^{-1}\overline{F}_N$. La matrice $\overline{F}_N$ est la TFD associée à la racine principale

$$2^{-1} = -2^{N/2} = 2^{N-1} \in \mathbb{Z}_{2^{N/2}+1};$$

le coût du calcul de $v = \overline{F}_N u$ est le même que celui de la TFR, en rajoutant les multiplications

$$N^{-1}v_i = 2^{(N-1)n}v_i \quad , \quad 1 \leq i \leq N$$

qui coûtent $O(N)$ chacune ; la complexité totale reste en $1,5N^2\log_2(N)+O(N\log_2(N))$ opérations binaires.

2.5. L'algorithme de Schönhage-Strassen pour la multiplication d'entiers. Maintenant on se tourne vers une application importante de la TFR : la multiplication rapide d'entiers. Soient

$$a, b \in \mathbb{N}$$

deux entiers de longueur binaire $\leq N = 2^n$. L'algorithme de multiplication longue calcule le produit

$$c = a \cdot b$$

en $O(N^2)$ opérations binaires ; on montrera que ceci peut se faire en complexité quasi-linéaire

$$O(N\log^2(N)\log(\log(N))).$$

L'algorithme qu'on présentera est une légère simplification de celui dû à Schönhage et Strassen [**5**] qui fait cette tâche en complexité $O(N\log(N)\log(\log(N)))$.

Classiquement on note $M(N)$ la complexité de multiplier deux entiers de longueur au plus N. Le théorème à démontrer est donc

THÉORÈME 2.8. $M(N) = O(N\log^2(N)\log(\log(N)))$.

Soient

$$t, \ell \in \mathbb{N}$$

des puissances de 2 telles que $t\ell = N$ et écrivons a et b en base 2^ℓ :

$$a = \sum_{j=0}^{t-1} a_j (2^\ell)^j \quad , \quad b = \sum_{j=0}^{t} b_j (2^\ell)^j$$

avec $0 \leq a_j, b_j \leq 2^\ell - 1$. Considérons les polynômes

$$P_a := \sum_{j=0}^{t-1} a_j x^j \quad , \quad P_b = \sum_{j=0}^{t} b_j x^j \in \mathbb{Z}[x].$$

Si on arrive à calculer rapidement le produit $P_a \cdot P_b$, on peut calculer le produit $a \cdot b$ comme

$$a \cdot b = (P_a \cdot P_b)(2^\ell).$$

Ceci se fait rapidement puisque l'évaluation en 2^ℓ consiste à faire une série de shifts et d'additions, ce qui prend $O(N \log(N))$ opérations binaires.

Intuitivement, on voudrait calculer la convolution $P_a \cdot P_b$ *via* la TFR, mais il y a un point délicat parce que la convolution repose elle même sur des multiplications d'entiers. Seulement par un choix convenable des paramètres t, ℓ la taille de ces entiers diminue et nous permet de monter une récurrence. Posons

$$T(N)$$

la complexité de multiplier deux entiers de longueur au plus N modulo $2^N + 1$. Trivialement

$$M(N) \leq T(2N)$$

puisque si $\tau(a), \tau(b) \leq N$ alors $0 \leq a \cdot b \leq 2^{2N}$ coïncide avec son reste modulo $2^{2N} + 1$. Soit

$$Q = P_a \cdot P_b = \sum_{j=0}^{2t-2} c_j x^j.$$

avec $c_j = \sum_{i=0}^{t-1} a_i b_{j-i}$ donc

$$0 \leq c_j \leq t 2^{2\ell}.$$

Pour calculer $c_j \pmod{2^N + 1}$ il suffit donc de le faire modulo $t(2^{2\ell} + 1)$, ou ce qui est équivalent grâce au théorème chinois, modulo t et modulo $2^{2\ell} + 1$ séparément.

Le calcul de Q modulo t se fait par l'algorithme classique, tandis que le calcul modulo $2^{2\ell} + 1$ se fait avec la TFR. Remarquons que 2 est une racine 4ℓ-ème de l'unité modulo $2^{2\ell} + 1$, donc le calcul de Q *via* la TFR en dimension $2t$ sera possible si

$$4\ell \geq \deg(Q) + 1 = 2t.$$

On prends donc le ℓ minimal satisfaisant aussi $t\ell = N$, donc

$$\ell := 2^{\lfloor n/2 \rfloor} \quad , \quad t := 2^{[n/2]}.$$

La racine $2t$-ème principale de l'unité utilisée est $\omega = 4$ pour n pair et $\omega = 2$ pour n impair. Voici l'algorithme complet :

Algorithme : multiplication d'entiers de Schönhage-Strassen.
Entrée : $a, b \in \mathbb{N}$ **entiers de longueur bit** $\leq N = 2^n$ **;**
Sortie : $a \cdot b$ **modulo** $2^N + 1$**.**

(1) $\ell \leftarrow 2^{\lfloor n/2 \rfloor}$**,** $t \leftarrow N/\ell$ **;**

(2) Écrivons

$$a = \sum_{j=0}^{t-1} a_j (2^\ell)^j \quad , \quad b = \sum_{j=0}^{t} b_j (2^\ell)^j$$

avec $0 \leq a_j, b_j \leq 2^\ell - 1$**, alors**

$$P_a \leftarrow \sum_{j=0}^{t-1} a_j x^j, P_b \leftarrow \sum_{j=0}^{t} b_j x^j;$$

(3) calcul de

$$R \leftarrow P_a \cdot P_b \pmod{t}$$

avec l'algorithme de multiplication longue ;

(4) calcul de

$$S \leftarrow P_a \cdot P_b \pmod{2^{2\ell}+1}$$

avec la TFR d'ordre $2t$, et multiplication d'entiers modulo $2^{2\ell}+1$ en utilisant cet algorithme récursivement ;

(5) $Q \leftarrow P_a \cdot P_b = (2^{2\ell}+1)(R - S \pmod{t}) + R$ **;**

(6) $a \cdot b \leftarrow Q(2^\ell) \pmod{2^N+1}$**.**

fin.

Estimons la complexité de cet algorithme. Le pas (3) est la multiplication de deux polynômes de degré $\leq t-1$ modulo t ; ceci demande $O(t^2)$ opérations modulo t, soit

$$O(t^2 \log^2(t)) = O(N \log^2(N))$$

opérations binaires. La TFR du pas (4) se fait en

$$O(\ell^2 \log(\ell)) = O(N \log^2(N))$$

opérations binaires. Ensuite il faut multiplier deux à deux modulo $2^{2\ell}+1$ les $2t$ coefficients de la TFD, ce qui demande

$$2tT(2\ell)$$

opérations binaires. La complexité du pas (5) est $O(N)$ et peut donc se négliger. Au total on obtient la récurrence

$$T(N) \leq 2tT(2\ell) + O(N \log^2(N)). \tag{3}$$

PROPOSITION 2.9. $T(N) = O(N \log^2(N) \log(\log(N)))$.

DÉMONSTRATION. On fait récurrence sur N. Posons $\widehat{T}(N) := T(N)/N$, alors la récurrence (3) se récrit en

$$\widehat{T}(N) \leq \frac{2tT(2\ell)}{N} + c_1 \log^2(N) = 4\widehat{T}(2\ell) + c_1 \log^2(N) \leq 4\widehat{T}(\sqrt{4N}) + c_1 \log^2(N)$$

pour un certain $c_1 > 0$. Soit N_0 tel que pour $N \geq N_0$ on ait $\sqrt{4N} \leq N^{2/3}$, alors l'hypothèse inductive $\widehat{T}(2\ell) \leq c_2 \log^2(2\ell) \log(\log(2\ell))$ entraîne

$$\begin{aligned}
\widehat{T}(N) &\leq 4c_2 \log^2(\sqrt{4N}) \log(\log(\sqrt{4N})) + c_1 \log^2(N) \\
&= c_2 \log^2(4N) \log(\frac{2}{3} \log(N)) + c_1 \log^2(N) \\
&\leq c_2 \log^2(N) \left(\frac{\log^2(4N)}{\log^2(N)} (\log(\log(N)) - \log(\frac{3}{2})) + \frac{c_1}{c_2} \right) \\
&\leq c_2 \log^2(N) \log(\log(N)
\end{aligned}$$

pour c_2 suffisamment grand et $N \gg 0$. □

REMARQUE 2.10. La complexité du pas (3) s'améliore en

$$O(t^{1,59} \log^2(t)) = O(N^{0,8} \log^2(N))$$

en utilisant par exemple la multiplication de polynômes de Karatsuba à la place de la multiplication longue. Dans le pas (4) on peut utiliser une"wrapped convolution" [**1**] à la place de la convolution habituelle.

Ces deux changements permettent d'aboutir à une complexité $O(N\log(N)\log(\log(N)))$ comme dans la version originale de l'algorithme, *voir* [**1**] pour les détails.

Les graphiques suivants illustrent la performance *pratique* des algorithmes de multiplication longue, de Karatsuba, et de Schönhage-Strassen. Ces expériences furent conduites par Emiliano Kargieman en utilisant la librairie GMP de précision multiple (`http://www.sox.com/gmp`).

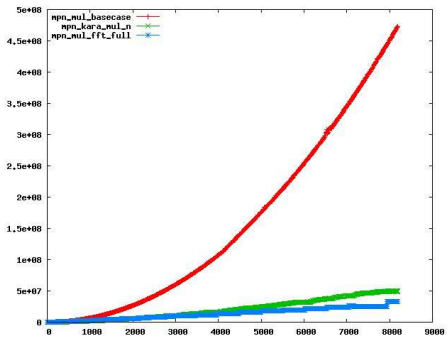

FIG. 2. Comparaison des trois méthodes (de 8 à 8192 bits en incréments de 8 bits)

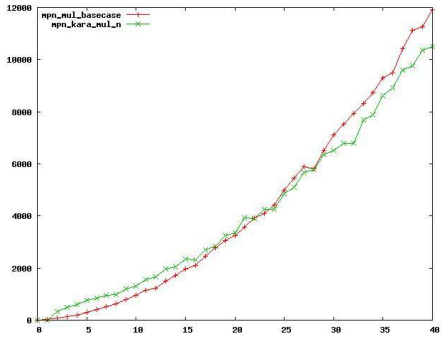

FIG. 3. Quand Karatsuba gagne contre la multiplication longue (de 0 à 40 bits en incréments de 1 bit)

On voit que Karatsuba commence à gagner contre la multiplication longue pour des nombres de 23 bits. La TFR dépasse la multiplication longue pour des nombres d'à partir 340 bits approximativement, mais pour qu'elle gagne contre toutes les autres méthodes il faut que les nombres soient d'au moins 2300 bits ! Si l'on suppose que les complexités $2N^2$ et $cN\log_2(N)\log_2(log_2(N))$ de la multiplication longue et de l'algorithme de Schönhage-Strassen reflètent le temps d'exécution réel, on conclut que la constante est au moins

$$c \geq \frac{2 \cdot 340^2}{340\log(340)\log(\log(340))} = 66,17.$$

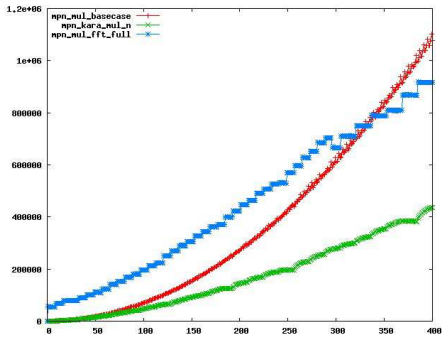

FIG. 4. Quand la TFR gagne à la multiplication longue (de 0 à 400 bits en incréments de 1 bit)

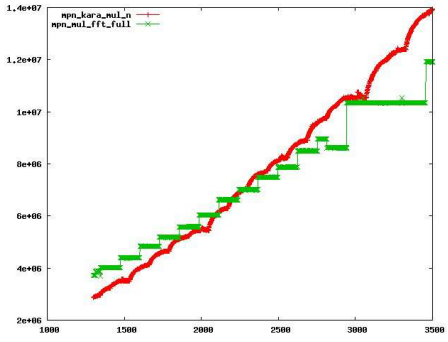

FIG. 5. Quand la TFR gagne contre Karatsuba (de 1300 à 3500 bits en incréments de 1 bit)

Bibliographie

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam, 1975. Second printing, Addison-Wesley Series in Computer Science and Information Processing.

[2] Lenore Blum, Felipe Cucker, Michael Shub, and Steve Smale. *Complexity and real computation*. Springer-Verlag, New York, 1998. With a foreword by Richard M. Karp.

[3] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19 :297–301, 1965.

[4] Michelle Schatzman. *Analyse numérique*. InterEditions, Paris, 1991. Cours et exercices pour la licence. [Course and exercises for the bachelor's degree].

[5] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing (Arch. Elektron. Rechnen)*, 7 :281–292, 1971.