Projet d'algorithmique en langage C

Multiplication de deux polynômes ou de deux entiers par transformée de Fourier rapide sur un corps fini

Réalisé par Alexandre Goyer et Nicholas Rumiz

Encadré par M. Michael Quisquater

Année universitaire 2019-2020

Sommaire

Introduction

Admettons que l'on veuille calculer le produit de 12345 par 6789. La méthode apprise à l'école primaire se présente de la façon suivante. Comptons le nombre d'opérations sur les chiffres.

				1	2	3	4	5
$\times$					6	7	8	9
			1	1	1	1	0	5
$+$			9	8	7	6	0	$\cdot$
$+$		8	6	4	1	5	$\cdot$	$\cdot$
$+$	7	4	0	7	0	$\cdot$	$\cdot$	$\cdot$
$=$	8	3	8	1	0	2	0	5

Le calcul des quatre (4 correspondant au nombre de chiffres de 6789) lignes intermédiaires demande de calculer quatre fois le produit de 12345 par un chiffre (en l'occurence 9, 8, 7 puis 6 ici). Chacun de ces quatre produits nécessite 5 (le nombre de chiffres de 12345) opérations sur les chiffres, voire plus si on compte les retenues. Sans même compter le coût du calcul de la dernière ligne, on observe alors que cette méthode demande au moins 4×5 opérations sur les chiffres.

De manière plus générale, il est aisé de voir que si on veut multiplier deux entiers de n chiffres chacun, cette méthode demande au moins n^2 opérations sur les chiffres. Inversement, on peut montrer qu'il existe une constante $M > 0$ telle que le calcul du produit de deux entiers ayant chacun un nombre de chiffres $\leq n$ avec cette méthode demande moins de $M \times n^2$ opérations sur les chiffres. Pour cette raison, on dit que la complexité de l'algorithme décrit par cette méthode est quadratique, ou bien en $\mathcal{O}(n^2)$.

Pendant longtemps, les mathématiciens ne soupçonnaient pas l'existence d'algorithmes ayant une meilleure complexité. C'est l'apparition des ordinateurs (et notamment les questions de cryptanalyse qui en découlent) qui a mené à la création de tels algorithmes. Le premier à voir le jour est dû au mathématicien russe Anatoli Karatsuba en 1962 dont la complexité est en $\mathcal{O}\left(n^{\log_2(3)}\right)$. Il fut suivi, quelques années plus tard, par l'algorithme Toom-Cook dû aux mathématiciens Andrei Toom (russe) et Stephen Cook (américano-canadien), qui en est un raffinement et dont la complexité est en $\mathcal{O}\left(n^{\log_3(5)}\right)$ (on a $\log_2(3) \simeq 1.58$ et $\log_3(5) \simeq 1.46$).

L'algorithme que l'on présente dans ce rapport fut publié en 1971 par les mathématiciens allemands Arnold Schönhage et Volker Strassen dans leur article intitulé *Schnelle Multiplikation großer Zahlen*, et sa complexité est en $\mathcal{O}\left(n \log(n) \log(\log(n))\right)$. Il resta celui possédant la meilleure complexité jusqu'à ce que le mathématicien suisse Martin Fürer propose un algorithme plus rapide en 2007 puis les mathématiciens Joris van der Hoeven (néerlandais) et David Harvey (américain) qui ont proposé en 2019 un algorithme dont la complexité est en $\mathcal{O}(n \log(n))$.

Cependant, la constante cachée dans le $\mathcal{O}$ joue un rôle important en réalité, ce qui rend inutilisables les algorithmes de Fürer et de van der Hoeven-Harvey. Ainsi, dans la pratique, on utilise l'algorithme naïf (celui décrit au début de l'introduction) pour des entiers de quelques dizaines de chiffres, l'algorithme de Karatsuba pour des entiers allant jusqu'à quelques milliers de chiffres et enfin l'algorithme de Schönhage-Strassen pour les entiers plus grands.

Chapitre 1

Présentation théorique de l'algorithme

A Racines de l'unité

Cette section permet de rappeler quelques résultats concernant les racines de l'unité. C'est grâce à elles que l'on peut mettre en place une transformation de Fourier rapide, ce qui constitue le coeur de l'algorithme de Schönhage-Strassen.

On se place dans un anneau $\mathbb{A}$ unitaire, commutatif et dans lequel 2 est inversible.

Définitions 1.1

Un élément $\alpha \in \mathbb{A}$ est *régulier* si pour tout β, $\alpha\beta = 0$ entraîne $\beta = 0$. Un élément $\omega \in \mathbb{A}$ est une *racine principale n-ième de l'unité* si $\omega^n = 1$ et si $\omega^t - 1$ est régulier pour tout $1 \leq t < n$.

***Remarque* 1.2** Pour montrer que ω est une racine principale n-ième de l'unité, il suffit de vérifier que $\omega^n = 1$ et $\omega^t - 1$ est régulier pour tout $1 \leq t < n$ diviseur de n. En effet, supposons que $1 \leq t < n$ ne divise pas n et notons d le pgcd de t et n. Alors d divise n et le théorème de Bezout fournit $(u, v) \in (\mathbb{N}\backslash\{0\})^2$ tel que $ut - vn = d$. On observe alors que $(\omega^t - 1)\left(1 + \omega^t + \cdots + \omega^{(u-1)t}\right) = \omega^d - 1$ et donc $\omega^t - 1$ est régulier puisque $\omega^d - 1$ l'est.

Lemme 1.3

Soit ω une racine principale n-ième de l'unité. L'inverse ω^{-1} est encore une racine principale n-ième de l'unité. Si $n = pq$, alors ω^p est une racine principale q-ième de l'unité et enfin, pour tout $1 \leq t < n$, on a :

$$\sum_{k=0}^{n-1} \omega^{tk} = 0\,. \tag{1.1}$$

Démonstration. Si $1 \leq t < n$, $(\omega^{-1})^t - 1$ est régulier car $((\omega^{-1})^t - 1)(-\omega^t) = \omega^t - 1$ l'est. De plus, on a bien $(\omega^{-1})^n = 1$ d'où le premier point. Le deuxième point est évident. Enfin, pour le troisième point, il suffit de remarquer que

$$(\omega^t - 1)\sum_{k=0}^{n-1}\omega^{tk} = \omega^{tn} - 1 = (\omega^n)^t - 1 = 0\,.$$

Comme $\omega^t - 1$ est régulier, la somme est bien nulle. □

Lemme 1.4

Soient $\omega \in \mathbb{A}$ et $n \geq 2$ un puissance de deux. On a le résultat suivant.

$$\omega \text{ est une racine principale } n\text{-ième de l'unité} \iff \omega^{\frac{n}{2}} = -1 \tag{1.2}$$

Démonstration. Si ω est une racine principale n-ième de l'unité, on a $(\omega^{\frac{n}{2}} - 1)(\omega^{\frac{n}{2}} + 1) = \omega^n - 1 = 0$ donc $\omega^{\frac{n}{2}} + 1 = 0$ car $\omega^{\frac{n}{2}} - 1$ est régulier.
Réciproquement, si $\omega^{\frac{n}{2}} = -1$, alors $\omega^n = (-1)^2 = 1$. Soit $1 \leq t < n$. D'après la remarque 1.2, on peut supposer que t divise n. En remarquant que $(\omega^t - 1)(\omega^t + 1) = \omega^{2t} - 1$ et en itérant ce principe, on observe que

$$(\omega^t - 1)(\omega^t + 1)(\omega^{2t} + 1)(\omega^{4t} + 1)\dots(\omega^{\frac{n}{4}} + 1) = \omega^{\frac{n}{2}} - 1 = -2$$

est inversible donc *a fortiori* régulier. On en déduit que $\omega^t - 1$ est régulier. □

B La transformée de Fourier rapide

B.1 Motivation

Le but de cette partie est de montrer comment calculer efficacement le produit de deux polynômes à une indéterminée à coefficients dans un anneau $\mathbb{A}$.

Supposons que l'on ait $A, B \in \mathbb{A}[X]$ non nuls tels que $\deg(A) + \deg(B) < n$, que l'on note

$$A(X) =: \sum_{i=0}^{n-1} a_i X^i \quad \text{et} \quad B(X) =: \sum_{i=0}^{n-1} b_i X^i\,.$$

On sait alors que

$$A(X) \times B(X) = \sum_{i=0}^{n-1} c_i X^i \quad \text{avec } c_i := \sum_{j=0}^{i} a_j b_{i-j} \quad \text{pour tout } 0 \leq i < n.$$

Or, un algorithme calculant le produit $A \times B$ en passant par la formule donnant les c_i en fonction des a_i, b_i aurait une complexité en $\mathcal{O}(n^2)$ opérations dans $\mathbb{A}$.

On propose ici une approche différente, dont la complexité est en $\mathcal{O}(n\log(n))$ opérations dans $\mathbb{A}$. Grâce à l'hypothèse faite sur le degré, pour connaître le polynôme AB, il suffit de déterminer les valeurs qu'il prend en n points distincts de $\mathbb{A}$. L'approche est alors la suivante.

- Évaluer A et B en des points $x_0, x_1, \dots, x_{n-1} \in \mathbb{A}$ deux à deux distincts.
- Calculer les produits point par point : $A(x_i) \times B(x_i)$ pour $0 \leq i \leq n-1$.

— Interpoler les valeurs trouvées pour obtenir le polynôme AB.

Le calcul des produits point par point demande exactement n opérations dans $\mathbb{A}$ donc il s'agit d'expliquer comment évaluer et interpoler un polynôme de manière efficace.

Pour faire tourner la théorie qui suit, les points $x_0, x_1, \ldots, x_{n-1}$ ne peuvent pas être choisis n'importe comment mais doivent correspondre aux puissances successives $1, \omega, \omega^2, \ldots, \omega^{n-1}$ d'une racine principale n-ième de l'unité. Un tel élément ω n'existe pas dans tout anneau $\mathbb{A}$, et quand bien même il existe, il n'est pas toujours simple de l'exhiber. Cependant, on ne fera appel à l'algorithme décris ci-après seulement dans un cadre où une racine n-ième principale de l'unité est naturellement donnée par la forme de $\mathbb{A}$ (cf. C L'algorithme de Schönhage-Strassen).

Ainsi, dans la suite, on se place dans un anneau $\mathbb{A}$ unitaire, commutatif, dans lequel 2 est inversible et dans lequel on connait ω une racine n-ième principale de l'unité dans $\mathbb{A}$.

Un exemple simple où toutes ces hypothèses sont vérifiées est le corps des nombres complexes. Mais on préfèrera ici prendre pour $\mathbb{A}$ un anneau de la forme $\mathbb{Z}/N\mathbb{Z}$ car ses éléments sont plus simples à manipuler avec un ordinateur. Il reste possible de faire fonctionner les idées suivantes pour multiplier deux polynômes à coefficients dans un anneau $\mathbb{A}$ plus général : on renvoie au livre « Algorithmes efficaces en calcul formel » pour une description détaillée.

B.2 La transformée de Fourier discrète

La théorie présentée dans cette section fut introduite au début du XIX$^{\text{ème}}$ siècle par le mathématicien français Joseph Fourier.

Définition 1.5

La *transformée de Fourier discrète* (abrégé **DFT** pour **D**iscrete **F**ourier **T**ransform en anglais) relativement à ω est l'opération d'évaluations suivante.

$$DFT_\omega : \left\{ \begin{array}{ccc} \mathbb{A}[X] & \longrightarrow & \mathbb{A}^n \\ P & \longmapsto & (P(1), P(\omega), ..., P(\omega^{n-1})) \end{array} \right.$$

***Remarque* 1.6** En munissant $\mathbb{A}^n$ de la multiplication coordonnée par coordonnée, la transformée de Fourier est un morphisme surjectif de $\mathbb{A}$-algèbres. Comme le noyau de DTF_ω est l'idéal $(X^n - 1)$, on obtient en factorisant un isomorphisme d'algèbres de $\mathbb{A}[X]/(X^n - 1)$ vers $\mathbb{A}^n$. Enfin, moyennant les identifications classiques $\mathbb{A}[X]/(X^n - 1) \simeq \mathbb{A}[X]_{<n} \simeq \mathbb{A}^n$, on fait correspondre à l'opération DFT_ω l'automorphisme de $\mathbb{A}^n$ dont la matrice dans la base canonique est la suivante.

$$V(\omega, n) := \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)^2} \end{pmatrix}$$

Théorème 1.7

$$DFT_{\omega}^{-1} = \frac{1}{n} DFT_{\omega^{-1}} \tag{1.3}$$

Démonstration. D'après le lemme 1.3, ω^{-1} est encore une racine n-ième principale de l'unité, et d'après la remarque précédente, il suffit de vérifier que $V(\omega, n) \times V(\omega^{-1}, n) = nI_n$. Or :

$$\left[V(\omega, n) \times V(\omega^{-1}, n)\right]_{i,j} = \sum_{k=0}^{n-1} \omega^{ik}\omega^{-jk} = \sum_{k=0}^{n-1} \omega^{(i-j)k} = n \times \delta_{i,j}$$

pour tous $1 \leq i, j \leq n$ encore d'après le lemme 1.3. □

***Remarque* 1.8** Ce dernier résultat montre que la troisième étape, qui consiste à interpoler un polynôme, est essentiellement analogue à la première étape, qui consiste à évaluer les polynômes. Ainsi, il ne nous reste plus qu'à expliquer comment calculer la transformée de Fourier discrète de manière efficace. On voit apparaître ici un des intérêts d'avoir évalué nos polynômes en les puissances d'une racine principale de l'unité.

B.3 L'algorithme de Cooley-Tukey

Il existe plusieurs algorithmes pour calculer la transformée de Fourier discrète. Ceux dont la complexité est en $\mathcal{O}(n \log(n))$ sont généralement appelés « algorithmes de transformée de Fourier rapide » (abrégé **FFT** pour **F**ast **F**ourier **T**ransform en anglais). Celui que nous avons choisi est dû aux mathématiciens américains James Cooley et John Tukey, il repose sur le principe de « diviser pour régner ». Quitte à remplacer n par la plus petite puissance de 2 qui lui est supérieure ou égale, on peut supposer que n est une puissance de 2.

Théorème 1.9

Notons d tel que $n = 2d$ et introduisons la matrice $D(\omega, d) := \text{diag}(1, \omega, \ldots, \omega^{d-1}) \in \mathcal{M}_d(\mathbb{A})$. Alors on a l'identité matricielle suivante.

$$V(\omega, n) = \left(\begin{array}{c|c} I_d & D(\omega, d) \\ \hline I_d & -D(\omega, d) \end{array}\right) \times \left(\begin{array}{c|c} V(\omega^2, d) & 0_d \\ \hline 0_d & V(\omega^2, d) \end{array}\right) \times \Pi(n) \tag{1.4}$$

où $\Pi(n) \times {}^t(a_0, a_1, \ldots, a_{n-1}) = {}^t(a_0, a_2, \ldots, a_{n-2}, a_1, a_3, \ldots, a_{n-1})$.

Démonstration. Soit $P = a_0 + a_1 X + \cdots + a_{n-1}X^{n-1} \in \mathbb{A}[X]$. Par construction, on sait que :

$$V(\omega, n) \times \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(1) \\ P(\omega) \\ \vdots \\ P\left(\omega^{n-1}\right) \end{pmatrix}.$$

Vu l'arbitraire de P, on aura montré le théorème si on montre que :

$$\left(\begin{array}{c|c} I_d & D(\omega,d) \\ \hline I_d & -D(\omega,d) \end{array}\right) \times \left(\begin{array}{c|c} V(\omega^2,d) & 0_d \\ \hline 0_d & V(\omega^2,d) \end{array}\right) \times \Pi(n) \times \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} P(1) \\ P(\omega) \\ \vdots \\ P\left(\omega^{n-1}\right) \end{pmatrix}.$$

Or, en introduisant $Q = a_0 + a_2X + \cdots + a_{n-2}X^{d-2}$ et $R = a_1 + a_3X + \cdots + a_{n-1}X^{d-2}$, on observe que $P(X) = Q(X^2) + XR(X^2)$. Vu la définition de $\Pi(n)$ et le fait que $V(\omega^2, d)$ corresponde à l'opération $DFT_{\omega^2} : \mathbb{A}^d \to \mathbb{A}^d$, on obtient l'identité suivante.

$$\left(\begin{array}{c|c} V(\omega^2,d) & 0_d \\ \hline 0_d & V(\omega^2,d) \end{array}\right) \times \Pi(n) \times \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} V(\omega^2,d) \times \begin{pmatrix} a_0 \\ a_2 \\ \vdots \\ a_{n-2} \end{pmatrix} \\ \hline V(\omega^2,d) \times \begin{pmatrix} a_1 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} \end{pmatrix} = \begin{pmatrix} Q(1) \\ Q\left(\omega^2\right) \\ \vdots \\ Q\left(\omega^{2(d-1)}\right) \\ R(1) \\ R\left(\omega^2\right) \\ \vdots \\ R\left(\omega^{2(d-1)}\right) \end{pmatrix}$$

Et donc finalement, en multipliant par la dernière matrice, on trouve que :

$$\begin{aligned} &\left(\begin{array}{c|c} I_d & D(\omega,d) \\ \hline I_d & -D(\omega,d) \end{array}\right) \times \left(\begin{array}{c|c} V(\omega^2,d) & 0_d \\ \hline 0_d & V(\omega^2,d) \end{array}\right) \times \Pi(n) \times \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \\ = &\left(\begin{array}{c|c} I_d & D(\omega,d) \\ \hline I_d & -D(\omega,d) \end{array}\right) \times \begin{pmatrix} Q(1) \\ Q\left(\omega^2\right) \\ \vdots \\ Q\left(\omega^{2(d-1)}\right) \\ R(1) \\ R\left(\omega^2\right) \\ \vdots \\ R\left(\omega^{2(d-1)}\right) \end{pmatrix} = \begin{pmatrix} Q(1) + R(1) \\ Q\left(\omega^2\right) + \omega R\left(\omega^2\right) \\ \vdots \\ Q\left(\omega^{2(d-1)}\right) + \omega^{d-1} R\left(\omega^{2(d-1)}\right) \\ Q(1) - R(1) \\ Q\left(\omega^2\right) - \omega R\left(\omega^2\right) \\ \vdots \\ Q\left(\omega^{2(d-1)}\right) - \omega^{d-1} R\left(\omega^{2(d-1)}\right) \end{pmatrix} \end{aligned}$$

ce qui permet de conclure, compte tenu du fait que $\omega^d = -1$ d'après le lemme 1.4. □

C L'algorithme de Schönhage-Strassen

On fait ici le lien entre notre objectif initial, la multiplication de grands entiers, et la tranformée de Fourier rapide décrite à la section précédente.

C.1 De la multiplication de deux polynômes ...

On rappelle ici le squelette d'algorithme pour calculer le produit de deux polynômes dont le degré du produit est inférieur à n. On suppose ici être dans un anneau où l'on dispose d'une racine principale de l'unité d'ordre n.

Multiplication de polynômes - squelette de l'algorithme

Entrées :
- $n > 0$ un entier
- $\omega \in \mathbb{A}$ une racine principale n-ième de l'unité.
- $P, Q \in \mathbb{A}[X]$ tels que $deg(PQ) < n$

Sortie : PQ

Les étapes :

1. *Précalcul :* Calculs de $\omega^2, \omega^3, \ldots, \omega^{n-1}$
2. *Évaluations :*

$$DFT_\omega(P) := (P(\omega^i))_{i=0}^{n-1}$$
$$DFT_\omega(Q) := (Q(\omega^i))_{i=0}^{n-1}$$

   C'est dans cette étape qu'intervient l'algorithme de FFT (Cooley-Tukey matriciel).

3. *Produits point à point* :

$$\begin{aligned} DFT_\omega(PQ) &= DFT_\omega(P) \times DFT_\omega(Q) \\ &= (P(\omega^i) \times Q(\omega^i))_{i=0}^{n-1} \end{aligned}$$

4. *Interpolation :* $(PQ(\omega^i))_{i=0}^{n-1} \overset{DFT_\omega^{-1}}{\longmapsto} PQ$
   Cette étape revient à faire une évaluation sur les puissances successives de ω^{-1}.

Comment utiliser cet algorithme pour calculer le produit de deux entiers a et b ?

C.2 ... À la multiplication de deux entiers

Cadre : On veut multiplier deux entiers a et b qui sont codés respectivement sur u et v bits.

Multiplication de deux entiers a et b : algorithme de Schönhage-Strassen

Entrées : $a, b \in \mathbb{N}$
Sortie : ab

Les étapes :

1. *Calculs préliminaires :* Calculs de u, v, k, K, M tels que :
   - u et v nombres de bits respectifs de a et b
   - k minimal tel que $2^k > \sqrt{u+v}$
   - $K = 2^k$
   - $M = \lceil \frac{u+v}{K} \rceil$ (partie entière supérieure)
2. *Décompositions en base 2^M :*

$$a = \sum_{i=0}^{K-1} a_i \left(2^M\right)^i$$
$$b = \sum_{i=0}^{K-1} b_i \left(2^M\right)^i$$

   puis on considère les polynômes associés dans l'anneau $A_K[X]$ où $A_K = \mathbb{Z}/(2^{3K}+1)\mathbb{Z}$:

$$A(X) = \sum_{i=0}^{K-1} \overline{a_i} X^i$$
$$B(X) = \sum_{i=0}^{K-1} \overline{b_i} X^i$$

3. *Calcul de $C = AB$ dans $A_K[X]$:*
   On exploite l'algorithme étudié précédemment avec $\omega = 2^6$ qui est une racine principale d'ordre K de l'unité dans A_K
4. *Évaluation :* $ab = C(2^M)$

Démonstration. À partir de la première étape on peut en déduire :

$$KM \geq u + v \quad \text{et} \quad M < K - 1\,.$$

En particulier on a bien $a, b < 2^{KM}$.

Par ailleurs, on a d'une part :

$$ab = \left(\sum_{i=0}^{K-1} a_i \left(2^M\right)^i\right)\left(\sum_{i=0}^{K-1} b_i \left(2^M\right)^i\right) = \sum_{i=0}^{2K-2} c_i \left(2^M\right)^i$$

où on a posé $c_i = \sum_{j=0}^{i} a_j b_{i-j}$. Or, comme $a < 2^u$ et $b < 2^v$, il vient $ab < 2^{u+v} \leq 2^{KM}$ donc $c_i = 0$ dès que $i \geq K$. De plus, pour tout $i \in [\![0, K-1]\!]$, $0 \leq c_i \leq (i+1)2^{2M} \leq K2^{2M} < 2^{3K}$.

Et d'autre part, on a :

$$AB(X) = \left(\sum_{i=0}^{K-1} \overline{a_i} X^i\right)\left(\sum_{i=0}^{K-1} \overline{b_i} X^i\right) = \left(\sum_{i=0}^{2K-2} \overline{p_i} X^i\right)$$

où $\overline{p_i} = \sum_{j=0}^{i} \overline{a_j}\overline{b_{i-j}} = \overline{c_i}$.

D'après ce qui précède on a donc $\deg(AB) < K$ et à condition de trouver une racine principale K-ième de l'unité dans A_K on peut faire appel à notre algorithme de multiplication de polynômes dans $A_K[X]$ qu'on a rappelé plus haut.

Comme de plus on a montré que $c_i < 2^{3K}$ on a que c_i est l'unique représentant de $\overline{c_i}$ dans l'intervalle $[\![0, 2^{3K}]\!]$ donc la connaissance des $\overline{c_i}$ permet de déterminer les c_i (et sur un ordinateur, la manipulation des classes $\overline{c_i}$ s'effectuant à partir de ces uniques représentants dans $[\![0, 2^{3K}]\!]$, il n'y aura rien à faire).

Reste à voir que $\omega = 2^6$ est une racine principale K-ième de l'unité dans A_K, mais ceci est clair d'après le **Lemme 1.4** car $(2^6)^{\frac{K}{2}} = 2^{3K} = -1$ dans A_K.

□

Chapitre 2

Présentation du code

A Architecture et fonctions intermédiaires

Le cœur de l'algorithme est la transformée de Fourier rapide, c'est-à-dire le calcul du produit d'un vecteur de $\mathbb{A}^K$ par la matrice $V(\omega, K)$ où ω désigne une racine K-ième principale de l'unité dans $\mathbb{A}$. L'une des premières difficultés à laquelle nous avons dû faire face a été la gestion des puissances de ω. En effet, vu que l'on appelle récursivement l'opération de transformée de Fourier, il est déraisonnable de recalculer les puissances de ω à chaque fois que l'on en a besoin. L'idée est de les calculer une bonne fois pour toute au début et de les stocker dans une liste.

Mais alors, que faire passer en argument de notre fonction FFT qui sera appelée récursivement ? Une liste avec seulement les puissances de ω dont on a besoin, c'est-à-dire deux fois plus petite à chaque appel de la fonction ? Ou bien la liste complète avec toutes les puissances de ω et un indicateur pour lui dire quelles puissances il aura besoin ? C'est la deuxième option que nous avons retenue car la manipulation de listes que demande la première option semblait trop couteuse.

On a donc introduit une nouvelle variable $e \in \mathbb{N}$ qui correspond à l'étage auquel on se situe à l'intérieur de l'arbre des appels de la fonction FFT. On commence avec $e = 0$: on dispose d'un vecteur de longueur K et des puissances $P_\omega = [1, \omega, \ldots, \omega^{K-1}]$ d'une racine K-ième principale de l'unité. On doit multiplier par la matrice $\Pi(K)$, ce qui est réalisé par la fonction `pi`, puis on appelle récursivement la fonction FFT avec des vecteurs deux fois plus petits et on doit alors renseigner les puissances de ω^2, racine $\frac{K}{2}$-ième principale de l'unité. Pour cela, on lui passe en argument toute la liste P_ω et on incrémente e de 1 pour lui indiquer qu'il doit sauter de $2(= 2^e)$ en 2 les éléments de P_ω. Plus généralement, la puissance i-ième de la racine dont il est question à l'étage e est en $(2^e i)$-ième position de P_ω. Enfin, on a codé une fonction `diag` qui correspond à la multiplication par la matrice $D(\omega, K)$, et une fonction `papillon` qui calcule $(F_1 + F_2 | F_1 - F_2)$ à partir des deux sous-listes F_1, F_2. Le pseudo-code de la fonction FFT est donné à la page suivante.

FFT (L, K, m, P, e)

Entrées K une puissance de 2
$m = 2^{3K} + 1$
$e \in \mathbb{N}$ tel que $N := \frac{K}{2^e} \geq 1$
$L = [q_0, q_1, \ldots, q_{N-1}]$ à coefficients dans $\mathbb{Z}/m\mathbb{Z}$
$P = [1, \omega, \ldots, \omega^{K-1}]$ où $w \in \mathbb{Z}/m\mathbb{Z}$ racine principale K-ième de l'unité
Sortie $[Q(1), Q(\alpha), \ldots, Q(\alpha^{N-1})]$ où $\alpha = \omega^{2^e}$ et $Q = q_0 + q_1 X + \cdots + q_{N-1} X^{N-1}$

* * * * *

Si $2^e < K$, **alors :**
$L \leftarrow$ `pi` (L, K, e)
$L_1 \leftarrow$ « première moitié de L »
$L_2 \leftarrow$ « deuxième moitié de L »
$L_1 \leftarrow$ **FFT** $(L_1, K, m, P, e+1)$
$L_2 \leftarrow$ **FFT** $(L_2, K, m, P, e+1)$
$L_2 \leftarrow$ `diag` (L_2, K, m, P, e)
$L \leftarrow$ `papillon` (L_1, L_2, m, e)

Nous avons également codé les fonctions :

- `parametres` qui prend les entiers a et b et qui détermine les valeurs de k et M comme décris dans l'algorithme de Schönhage-Strassen,
- `decomposition` qui prend un entier a et renvoie son écriture $a_0, \ldots, a_{K-1}$ en base 2^M,
- `precalcul` qui calcule les puissances successives de ω et celles de ω^{-1},
- `point_par_point` qui effectue le produit coordonnée par coordonnée entre deux listes,
- `div_par_K` qui divise tous les éléments d'une liste par K,
- `evaluation` qui effectue exactement l'inverse de la fonction `decomposition`.

Le pseudo-code de la fonction `multiplication` qui reprend l'algorithme de Schönhage-Strassen et qui est le cœur de notre code est présenté à la page suivante.

`multiplication` (a, b)

Entrées $a, b \in \mathbb{N}$
Sortie $ab \in \mathbb{N}$

* * * * *

$k, M \leftarrow$ `parametres` (a, b)
Si $k < 7$, **alors :**
    Retourner $a \times b$ (calculé avec un autre algorithme)
Sinon :
    $K \leftarrow 2^k$
    $L_a \leftarrow$ `decomposition` (a, K, M)
    $L_b \leftarrow$ `decomposition` (b, K, M)
    $\omega \leftarrow 2^6$
    $\text{mod} \leftarrow 2^{3K} + 1$
    $P_\omega, P_{\omega^{-1}} \leftarrow$ `precalcul` (ω, K, mod)
    $F_a \leftarrow$ `FFT` $(L_a, K, \text{mod}, P_\omega, 0)$
    $F_b \leftarrow$ `FFT` $(L_b, K, \text{mod}, P_\omega, 0)$
    $F_{ab} \leftarrow$ `point_par_point` $(F_a, F_b, K, \text{mod})$
    $L \leftarrow$ `FFT` $(F_{ab}, K, \text{mod}, P_{\omega^{-1}}, 0)$
    $L_{ab} \leftarrow$ `div_par_K` (L, K, mod)
    $ab \leftarrow$ `evaluation` (L_{ab}, K, M)
    Retourner ab

B La librairie GMP

Pour que l'algorithme de Schönhage-Strassen présenté dans ce rapport soit vraiment plus efficace que l'algorithme naïf ou que l'algorithme de Karatsuba et ses variantes, il est nécessaire de l'appeler pour des entiers grands, très grands. Or, en langage C, un nombre entier est codé sur 4 octets, voire 8 octets si on considère des `long int` plutôt que des `int`. Quoi qu'il en soit, on est limité dans la taille des entiers (ils ne peuvent pas dépasser une vingtaine de chiffres décimaux).

C'est pourquoi on a fait appel à la librairie GMP qui permet de manipuler des entiers sans limite de taille. La syntaxe pour travailler avec des entiers de GMP – appelés `mpz_t` – étant lourde, nous employons tantôt des entiers `mpz_t` lorsque cela est nécessaire (les entiers de l'anneau A_K par exemple), tantôt des entiers `int` quand on sait que la taille de l'entier considéré est relativement petite, c'est le cas des variables k, K, M, e par exemple. La fonction `precalcul` présentée ci-dessous illustre bien ce propos.

```c
#include <gmp.h>

void precalcul (mpz_t liste[] , mpz_t liste_inv[] , mpz_t w , int K , mpz_t mod)
/* Cette fonction remplie liste=[1,w,w^2,...,w^(K-1)] et
liste_inv=[1,w^(-1),...,w^2,w] avec w une racine d ordre K dans Z/modZ    */
{
    int i ;
    int Ksur2 = K>>1 ;
    mpz_t x , moinsx ;
    mpz_init(moinsx) ;


    mpz_init_set_ui(x,1) ;
    mpz_sub(moinsx,mod,x) ;
    mpz_init_set(liste[0],x) ;
    mpz_init_set(liste[Ksur2],moinsx) ;
    mpz_init_set(liste_inv[0],x) ;
    mpz_init_set(liste_inv[Ksur2],moinsx) ;

    /* Remplissage simultané en utilisant les symétries */
    for (i=1 ; i<Ksur2 ; i++)
    {
        multiplication(&x,x,w) ;
        reste(&x,x,mod) ;
        mpz_sub(moinsx,mod,x) ;
        mpz_init_set(liste[i],x) ;
        mpz_init_set(liste[Ksur2+i],moinsx) ;
        mpz_init_set(liste_inv[K-i],x) ;
        mpz_init_set(liste_inv[Ksur2-i],moinsx) ;
    }
}
```

Comme on peut le voir, chaque entier `mpz_t` doit être déclaré puis initialisé avant de pouvoir être utilisé. Expliquons le fonctionnement d'une fonction de GMP, par exemple `mpz_fdiv_r`. Si `a`, `b`, `n` désigne trois entiers `mpz_t`, la commande `mpz_fdiv_r(a,b,n)` remplace la valeur de `a` par le reste de la division euclidienne de `b` par `n`.

C Coder « in place »

Dans l'algorithme FFT, le polynôme auquel on applique la transformation est représenté par la liste de ses coefficients $[a_0, a_1, \ldots, a_{K-1}]$. Au cours de l'algorithme, on appelle récursivement FFT sur deux listes deux fois plus petites $[a_0, a_2, \ldots, a_{K-2}]$ et $[a_1, a_3, \ldots, a_{K-1}]$.

Une première solution serait de créer à chaque étage deux nouvelles listes et de construire une autre liste à partir des deux sous-listes obtenues récursivement. L'inconvénient de cette méthode est la taille mémoire qu'elle requiert. En effet, considérons $K \geq 1$ une puissance de deux et notons $\mathrm{mem}(K)$ la place mémoire que prend cette méthode pour une liste de taille K. On a $\mathrm{mem}(1) = 1$ et pour $K > 1$,

$$\mathrm{mem}(K) = \frac{K}{2} + \frac{K}{2} + 2 \times \mathrm{mem}\left(\frac{K}{2}\right) + K \,,$$

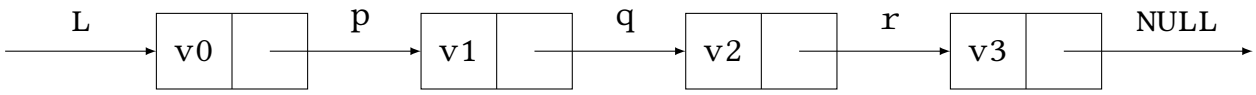
et il s'ensuit alors que $\mathrm{mem}(K) = K + 2K \log_2(K)$.

Ce paragraphe propose une autre manière de faire, pour laquelle la place mémoire requise au rang K est exactement K. L'idée est de gérer une seule liste tout au long de l'algorithme et d'appliquer des transformations au sein même de cette liste. On parle dans ce cas de « in place transform ».

Les listes classiques du langage C ne se prêtant peu bien à ce jeu, on décide de travailler avec des « listes chaînées ». Un des intérêts des listes chaînées est aussi l'absence de contrainte (autre que physique) sur la longueur de la liste. Mais ici ce n'est pas cette raison qui nous a poussés à les utiliser : en effet, pour gérer un entier avec un million de décimales, K est calculé dès le début et $K \sim 1000$ est relativement petit.

On définit alors une nouvelle structure qui va servir de brique de base pour construire nos listes chaînées. Comme tout objet en langage C, cette brique de base – que l'on appellera « élément » – est déterminée par son adresse. Dans la suite, on identifie l'élément en question avec un pointeur vers cet élément (*i.e.* tel que la valeur du pointeur soit l'adresse de l'élément).

Dans chaque élément est stocké un couple `(v,p)` où `v` désigne la valeur que l'on associe à cet élément et `p` désigne un pointeur vers un autre élément ou bien le pointeur « NULL » pour symboliser la fin de la liste. Nous avons choisi pour le type de `v` la classe des entiers de la librairie GMP. Par exemple, on peut représenter une liste chaînée `L` comportant quatres éléments de la manière suivante.



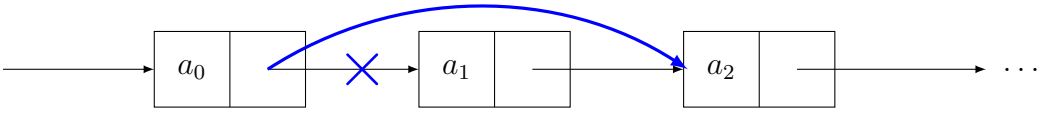
Voici comment on peut définir un tel objet en langage C.

```c
typedef struct element
{
    mpz_t valeur;
    struct liste * suivant;
}element;
```

Montrons maintenant comment gérer ce genre de listes sur un exemple : l'implémentation de la fonction `pi` dont la transformation souhaitée est la suivante :

$$[a_0, a_1, a_2, a_3, \ldots, a_{K-2}, a_{K-1}] \longmapsto [a_0, a_2, \ldots, a_{K-2}, a_1, a_3, \ldots, a_{K-1}] .$$

Pour cela, on ne va pas modifier la valeur des éléments de la liste mais on va plutôt profiter de pouvoir agir sur les pointeurs pour ainsi modifier l'ordre de succession des éléments de la liste. Le premier élément de la liste est composé de la valeur a_0 et d'un pointeur vers le deuxième élément (celui correspondant à a_1), on remplace ce pointeur par le pointeur contenu dans le deuxième élément, c'est-à-dire un pointeur vers le troisième élément (celui correspondant à a_2).



Puis on recommence en partant cette fois du deuxième élément : on remplace le pointeur qu'il stocke initialement (*i.e.* un pointeur vers le troisième élément) par un pointeur vers le quatrième élément et ainsi de suite jusqu'au $(K-1)$-ième élément (celui correspondant à a_{K-2}), pour lequel on remplace le pointeur qu'il stocke par un pointeur vers le deuxième élément (dont on doit avoir gardé une trace, car il ne correspond plus au pointeur stocké dans le premier élément). Ci-dessous le code de la fonction `pi` suivant cette méthode.

```
element *pi (element *liste, int K, int e)
/* La commande pi(L,K,e) sépare les éléments d'incides pairs de ceux
d indices impairs parmi les K/(2^e) premiers elements de L.            */
{
    element *p1 = liste ;
    element *p2 = p1->suivant ;
    element *p_sauvegarde = p2 ;
    int i ;
    for (i=2 ; i < K/(1<<e) ; i++)
    {
        p1->suivant = p2->suivant ;
        p1 = p2 ;
        p2 = p1->suivant ;
    }
    p1->suivant = p_sauvegarde ;
}
```

Chapitre 3

Évaluation de la complexité

A Étude de la fonction FFT

Rappelons un résultat de théorie de la complexité qui nous sera utile pour la suite.

Lemme 3.1

Soit $(u_n)_{n\geq 0}$ une suite croissante à termes strictement positifs.

1) Si $u_n \leq u_{\lceil \frac{n}{2} \rceil} + \mathcal{O}(1)$, alors $u_n = \mathcal{O}(\log_2 n)$.
2) Si $u_n \leq 2u_{\lceil \frac{n}{2} \rceil} + \mathcal{O}(n)$, alors $u_n = \mathcal{O}(n \log_2 n)$.

Démonstration. 1) Notons $C > 0$ telle que $u_n \leq u_{\lceil \frac{n}{2} \rceil} + C$. Soit n une puissance de deux. Par récurrence, il vient que $u_n \leq u_{\frac{n}{2^k}} + kC$ et pour $k = \log_2(n)$, on obtient $u_n \leq u_1 + C\log_2(n)$. Si n n'est pas une puissance de 2, il existe une telle puissance dans $[\![n, 2n]\!]$ et on conclut par croissance. 2) La suite $v_n = \frac{u_n}{n}$ vérifie $v_n \leq v_{\lceil \frac{n}{2} \rceil} + \mathcal{O}(1)$ donc le 1) termine la démonstration. □

Notons $\mathbb{A}$ l'anneau des coefficients du polynôme auquel on applique la transformée de Fourier et considérons C_K le coût total de la fonction `fft` sur un polynôme de degré $< K$ qui prendra en compte les opérations dans $\mathbb{A}$ et le nombre de modifications de pointeurs.

Propriété 3.2

Le nombre C_K d'opérations effectuées par la fonction `fft` sur un polynôme de degré $< K$ vérifie :

$$C_K = \mathcal{O}(K \log_2 K).$$

Démonstration. L'algorithme de FFT repose sur le principe de "Diviser pour régner" cela signifie dans notre cas que pour une liste chaînée en entrée, on applique la fonction `fft` sur chacune de ses deux moitiés. On peut se référer au pseudo-code de la fonction `fft` présenté à la page 12.

À chaque appel de la fonction `fft`, on va appliquer la fonction `pi`. Cette dernière consiste à parcourir les éléments de la liste chaînée pour les réordonner, il n'y a donc pas d'opérations effectuées dans A_K à ce niveau mais il nous faut tout de même compter les opérations consistant en

des modifications de pointeurs. La fonction `fft` se termine également par un appel aux fonctions `diag` et `papillon`. Analysons les coûts respectifs des sous-fonctions mentionnées ci-dessus.

- `pi` : $< 3K$ modifications de pointeurs
- `diag` :
  - $\frac{K}{2}$ multiplications dans $\mathbb{A}$
  - $K + 1$ modifications de pointeurs
- `papillon` :
  - $\leq 2K$ additions dans $\mathbb{A}$
  - $2K$ modifications de pointeurs

On retiendra donc que ces fonctions coûtent globalement $\mathcal{O}(K)$ opérations, d'où finalement :

$$C_K \leq 2C_{\frac{K}{2}} + \mathcal{O}(K) \,,$$

et donc d'après le **Lemme 3.1**, on conclut que

$$C_K = \mathcal{O}(K \log_2 K) \,.$$

□

***Remarque* 3.3** Étant un peu pris par le temps, nous n'illustrons pas ce résultat via des résultats expérimentaux mais des graphes d'évaluation du temps de nos fonctions apparaitront lors de notre présentation orale.

B Étude de la complexité générale de l'algorithme de Schönhage-Strassen

On s'intéresse dans cette partie à la complexité de la fonction `multiplication`. Comme on a pu le voir en étudiant le détail des fonctions mises en jeu dans `multiplication`, pour calculer la multiplication de deux entiers, on a besoin de faire des multiplications dans $A_K = \mathbb{Z}/(2^{3K}+1)\mathbb{Z}$, donc on a envie d'appeler récursivement la fonction `multiplication`. Pour cela il nous faut régler le cas de base qui permet de renvoyer un résultat. Autrement dit, pour des entiers suffisamment petits (en un sens qu'on précisera) on n'utilise pas l'algorithme de Schönhage-Strassen, mais un algorithme de type naïf par exemple. Nous avons plutôt fait le choix d'utiliser la commande `mpz_mul` pour faire le calcul dans ce cas précis.

Pour analyser le coût d'une multiplication dans $A_K = \mathbb{Z}/(2^{3K}+1)\mathbb{Z}$, nous utiliserons le résultat suivant.

Lemme 3.4

Soient deux entiers a, b codés sur au plus n bits. Alors, on peut calculer a mod b en $\mathcal{O}(C(n) \log_2 n)$ opérations $(+, -, \times)$ dans $\mathbb{Z}$.

Démonstration. Voir [A.E.C.F] p.98 (ou le code). □

Motivation

Rappel : Avec les notations de la partie **1.C : L'algorithme de Schönhage-Strassen**, et en notant $n = \max(u, v)$, on a les résultats suivants :

$$K > \sqrt{u+n} > \sqrt{n} \quad \text{et} \quad \frac{K}{2} \leq \sqrt{u+v} \quad \text{donc} \quad K \leq 2\sqrt{2n}\,.$$

Par ailleurs, si on suppose $a, b \in [\![0, 2^{3K}+1[\![$, alors ab sera codé sur moins de $6K+1$ bits. Pour ne pas tourner en rond dans notre algorithme (voir **Lemme 3.4**), il est raisonnable de demander que

$$6K + 1 < \frac{n}{2}\,.$$

De cette manière, à chaque appel à la fonction `multiplication`, la taille des entiers en entrée est toujours plus petite de moitié (au moins) par rapport à celle des entiers considérés au tour précédent.

Quels sont les K qui conviennent ?

Comme on a :

$$6K + 1 < 8K \leq 2^4\sqrt{2n}\,,$$

il suffit de vérifier :

$$2^4\sqrt{2n} \leq \frac{n}{2} \quad \text{, c'est-à-dire} \quad 2^5\sqrt{2} \leq \sqrt{n}\,.$$

Or dès que $k \geq 7$, on a :

$$n \geq \frac{K^2}{8} = 2^{2k-3} \geq 2^{11} = (2^5\sqrt{2})^2\,.$$

Ainsi, les $K \geq 128\ (= 2^7)$ conviennent.

Quand le K calculé par la fonction `parametres` vérifie $K \geq 128$, on lance notre algorithme de Schönhage-Strassen (fonction `multiplication`). Dans le cas contraire on fera appel à `mpz_mul`.

***Remarque* 3.5** On a en fait montré un peu mieux que ce qu'on voulait, on a montré que pour $k \geq 7$, quand on fait un sous appel à la fonction `multiplication`, les entiers en entrée ne sont plus codés sur n bits mais sur $24\sqrt{n} < \frac{n}{2}$ bits.

Étude de la complexité

Notons $C(n)$ le nombre d'opérations binaires (ou modifications de pointeurs) nécessaires pour multiplier deux entiers codés sur au plus n bits avec la fonction `multiplication`.

Regardons le coût en opérations binaires de chacune des fonctions qui interviennent dans la fonction `multiplication`.

- `decomposition` : $\mathcal{O}(MK) = \mathcal{O}(n)$ opérations (car $MK < K^2 \leq 8n$).
- `parametres` : $\mathcal{O}(\log_2(n)\log_2(\log_2(n)))$ opérations.
- `precalcul` : $< C_1\, K \log(24\sqrt{n})\, C(24\sqrt{n}) + \mathcal{O}(K)$ opérations.
- `point_par_point` : $< C_2\, K \log(K)\, C(24\sqrt{n}) + \mathcal{O}(K^2)$ opérations.
- `evaluation` : $\mathcal{O}(MK) = \mathcal{O}(n)$ opérations.
- `div_par_K` : $< C_3\, K \log(24\sqrt{n})\, C(24\sqrt{n}) + \mathcal{O}(K^2)$ opérations.
- `fft` : $< C_4\, K \log(K) \log(24\sqrt{n})\, C(24\sqrt{n}) + \mathcal{O}(K^2 \log K)$ opérations.

En réunissant toutes ces informations, et vu que $K = \mathcal{O}(\sqrt{n})$, on obtient l'existence d'une constante $C > 0$ telle que :

$$C(n) \leq C\, \sqrt{n}\, (\log n)^2\, C(24\sqrt{n}) + \mathcal{O}(n \log n)\,. \tag{3.1}$$

On peut finalement énoncer le résultat général suivant sur la complexité de notre algorithme.

Théorème 3.6 : Complexité de la fonction `multiplication`

Avec les notations précédentes, nous avons, pour tout $\epsilon > 0$:

$$C(n) = \mathcal{O}(n^{1+\epsilon})\,.$$

Démonstration. Soient $\epsilon > 0$ et $\delta = 1 + \frac{\epsilon}{2} > 1$, considérons $f(n) := \frac{C(n)}{n^\delta}$. Montrons que $f(n) = \mathcal{O}(n^{\frac{\epsilon}{2}})$, ce qui concluera.

D'après l'inégalité **(3.1)**, comme $\delta - \frac{1}{2}(1+\delta) > 0$, on a par croissance comparée :

$$f(n) \leq \underbrace{C 24^\delta \frac{\log_2(n)^2}{n^{\delta - \frac{1}{2}(1+\delta)}}}_{\substack{n \to +\infty \\ \longrightarrow 0}\ \text{donc est borné}} f(24\sqrt{n}) + \mathcal{O}(1)\,.$$

Ainsi, il existe des constantes $A, B > 0$ telles que :

$$f(n) \leq A f(24\sqrt{n}) + B\,.$$

En itérant, on trouve que

$$f(n) \leq A^p\, f\big(\underbrace{24^{(1+\frac{1}{2}+\cdots+\frac{1}{2^{p-1}})}}_{\leq 24^2} n^{\frac{1}{2^p}}\big) + B(\underbrace{1 + A + \cdots + A^{p-1}}_{=\frac{A^p-1}{A-1}})\,.$$

Compte tenu de ce qui précède, on peut demander $A \geq 2$, d'où :

$$f(n) \leq A^p (B + f(24^2 n^{\frac{1}{2^p}}))\,.$$

Par ailleurs :

$$\begin{aligned} 24^2 n^{\frac{1}{2^p}} \leq n_0 &\iff \frac{1}{2^p} \log_2 n \leq \log_2\left(\frac{n_0}{24^2}\right) \\ &\iff 2^p \geq \frac{\log_2 n}{\log_2\left(\frac{n_0}{24^2}\right)} \\ &\iff p \geq \log_2\left(\frac{\log_2 n}{\log_2 \frac{n_0}{24^2}}\right)\,. \end{aligned}$$

Posons alors :

$$n_0 := 2 \times 24^2 \text{ et } p := \lceil \log_2(\log_2 n) \rceil ,$$

de telle sorte que :

$$\begin{aligned} f(n) &\leq A^{\log_2(\log_2 n)}(B + f(n_0)) \\ &= \left(2^{log_2 A}\right)^{(\log_2(\log_2 n))}(B + f(n_0)) \\ &= \left(2^{(\log_2(\log_2 n))}\right)^{\log_2 A}(B + f(n_0)) \\ &= (B + f(n_0))(\log_2 n)^{log_2 A} . \end{aligned}$$

Par croissances comparées, on en déduit bien que $f(n) = \mathcal{O}(n^{\frac{\epsilon}{2}})$. □

***Remarque* 3.7** Nous n'obtenons donc pas la complexité connue pour cet algorithme, mais une complexité, au moins en théorie, qui bat les algorithmes de Karatsuba et de Toom-Cook.

Références

Bibliographie

— [A.E.C.F] Algorithmes Efficaces en Calcul Formel, *Frédéric Chyzak, Alin Bostan, Marc Giusti, Romain Lebreton, Grégoire Lecerf, Bruno Salvy, Éric Schost*, 2017.

— Cours de « Programmation en langage C », *Michaël Quisquater*, UVSQ, 2019.

— Fast Fourier transforms : a tutorial review and a state of the art, *Pierre Duhamel, Martin Vetterli*, 1990.

— Some historical notes on number theoretic transform, *Mrinmoy Bhattacharya, Reiner Creutzburg, Jaakko Astola*, 2004.

— Fast convolution using Fermat number transforms with applications to digital flitering, *Ramesh Agarwal, Charles Burrus*, 1974.

Webographie

— A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm, *Pierrick Gaudry, Alexander Kruppa, Paul Zimmermann* , 2007.

`https://hal.inria.fr/inria-00126462v1`

— La page « Cooley–Tukey FFT algorithm » de Wikipedia, visitée le mardi 18 février 2020.

`https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm`

— La page « Schönhage-Strassen algorithm » de Wikipedia, visitée le mardi 18 février 2020.

`https://en.wikipedia.org/wiki/Schönhage-Strassen_algorithm`