Théorie de Fourier et convolution

MULTIPLICATION RAPIDE & FFT

Quentin Deschamps - Ruxue Zeng

Décembre 2020

Table des matières

1 Introduction

Un algorithme de multiplication donne le résultat du produit de deux nombres. Nous allons nous intéresser à la multiplication des nombres entiers. La complexité d'un tel algorithme est alors calculée en fonction du nombre de chiffres des entiers. Il existe de nombreuses méthodes pour multiplier deux nombres. On peut les classer en deux catégories.

Tout d'abord, on trouve les méthodes basées sur l'écriture des entiers dans leur base numérique. Celles-ci consistent à multiplier chaque chiffre du premier nombre par chaque chiffre du second. On peut citer par exemple l'algorithme "standard", ou *Long multiplication* en anglais, qui correspond à la méthode enseignée à l'école primaire. Si les deux nombres ont n chiffres, alors cette méthode exige n^2 produits. Le calcul a donc une complexité en temps quadratique, c'est-à-dire en $\mathcal{O}(n^2)$.

Ensuite, l'apparition des ordinateurs a permis la mise au point d'algorithmes plus rapides pour les grands nombres. Ces algorithmes ont une complexité en temps de l'ordre de $\mathcal{O}(n \log n)$. On parle alors d'algorithmes de multiplication rapide [1].

Parmi ceux-ci, on trouve l'algorithme de Schönage et Strassen. Cet algorithme, publié en 1971, permet de multiplier deux grands entiers efficacement en utilisant la transformée de Fourier rapide [2].

Ce rapport vise à décrire l'algorithme de Schönage et Strassen et étudier sa complexité. Il sera comparé à d'autres algorithmes de multiplication, comme l'algorithme de Karatsuba ou encore celui de Fürer.

Vous pouvez trouver l'implémentation des algorithmes en MATLAB à l'adresse suivante :

https://github.com/Quentin18/fft-fast-multiplication

Par la suite, l'ensemble $\mathbb{Z}_b^n$ désignera l'ensemble des entiers de base b ayant n chiffres. On prend la convention qu'un nombre entier x est associé à sa représentation dans la base b comme un vecteur de taille n.

$$x = \sum_{i=0}^{n-1} x_i b^i \equiv (x_0, ..., x_{n-1})_b$$

2 Algorithme de Schönhage-Strassen

L'algorithme de Schönhage et Strassen est un algorithme de multiplication pour les grands entiers. Il a été développé par Arnold Schönhage et Volker Strassen en 1971. Cet algorithme est basé sur la transformée de Fourier rapide, ainsi que sur le théorème de convolution pour multiplier deux polynômes.

La méthode présentée dans ce rapport n'est pas forcément la plus optimale. Il s'agit ici de montrer le fonctionnement global de l'algorithme par l'utilisation de la FFT.

2.1 Transformée de Fourier discrète

L'algorithme de Schönhage et Strassen repose principalement sur la transformée de Fourier discrète.

Définition (Transformée de Fourier discrète)**.** Soit $n \geq 1$. On pose $\omega = e^{\frac{2\pi i}{n}}$. La transformée de Fourier discrète d'un vecteur $x = (x_0, ..., x_{n-1}) \in \mathbb{C}^n$ est le vecteur $\hat{x} = (\hat{x}_0, ..., \hat{x}_{n-1}) \in \mathbb{C}^n$ défini par :

$$\hat{x}_k = \sum_{j=0}^{n-1} x_j \omega^{kj} \quad , \quad 0 \leq k \leq n-1$$

1. *Fast multiplication algorithm* en anglais
2. *Fast Fourier Transform* ou *FFT* en anglais

La transformation inverse est donnée par :

$$x_k = \frac{1}{n}\sum_{j=0}^{n-1} \hat{x}_j \omega^{-kj} \quad , \quad 0 \leq k \leq n-1$$

Ainsi, la transfomation de Fourier discrète est une bijection de $\mathbb{C}^n$ dans $\mathbb{C}^n$.
Par exemple, pour n = 4, on a :

$$\begin{cases} \hat{x}_0 = x_0 + x_1 + x_2 + x_3 \\ \hat{x}_1 = x_0 + ix_1 - x_2 - ix_3 \\ \hat{x}_2 = x_0 - x_1 + x_2 - x_3 \\ \hat{x}_3 = x_0 - ix_1 - x_2 + ix_3 \end{cases}$$

La méthode naïve pour calculer la transformation de Fourier discrète d'un vecteur de taille n a une complexité en $\mathcal{O}(n^2)$. Grâce à l'algorithme FFT (*Fast Fourier Transform*), et plus particulièrement à l'algorithme de Cooley–Tukey [3], on peut effectuer cette opération en $\mathcal{O}(n \log n)$.

2.2 Produit de convolution des polynômes

Le principe fondamental de l'algorithme de Schönhage et Strassen est de considérer les entiers comme des polynômes. Ainsi, multiplier les deux entiers revient alors à calculer le produit de convolution des polynômes associés.

Considérons deux vecteurs $x = (x_0, ..., x_{n-1}) \in \mathbb{C}^n$ et $y = (y_0, ..., y_{n-1}) \in \mathbb{C}^n$. On peut alors interpréter x et y comme les coefficients de deux polynômes f et g.

$$\begin{cases} f(X) = x_0 + x_1 X + ... + x_{n-1} X^{n-1} \\ g(X) = y_0 + y_1 X + ... + y_{n-1} X^{n-1} \end{cases}$$

On peut tout d'abord remarquer que calculer la TFD [4] du vecteur x revient à évaluer le polynôme f aux racines n-ièmes de l'unité. En effet,

$$\hat{x} = (f(1), f(\omega), ..., f(\omega^{n-1}))$$

À partir de cette observation, on peut alors définir la méthode suivante pour calculer le produit de convolution de deux polynômes avec la FFT :

1. On utilise la FFT pour évaluer f et g aux racines n-ièmes de l'unité.
2. On multiplie les deux vecteurs obtenus point par point.
3. On utilise la transformée de Fourier inverse pour obtenir $f * g$.

Cette méthode se résume alors par le théorème suivant.

Théorème (Théorème de convolution)**.** *Soit deux polynômes $f, g \in \mathbb{C}[X]$. Notons $\mathcal{F}$ la transformation de Fourier et $\mathcal{F}^{-1}$ sa transformation inverse. Alors,*

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$$

Autrement dit,

$$f * g = \mathcal{F}^{-1}(\mathcal{F}(f) \cdot \mathcal{F}(g))$$

3. Implémentation en MATLAB : https://github.com/Quentin18/fft-fast-multiplication/tree/master/fft-cooley-tukey
4. Transformée de Fourier Discrète

Le produit de convolution $f * g$ peut donc être réalisé en $\mathcal{O}(n \log n)$ en utilisant la transformée de Fourier rapide.

On considère les vecteurs $x, y \in \mathbb{C}^n$ tel que $n = 2^k, k \in \mathbb{N}$. Voici le pseudo-code de l'algorithme de convolution rapide.

Algorithm 1: Fast convolution algorithm

Input : $x \in \mathbb{C}^n$, $y \in \mathbb{C}^n, n = 2^k, k \in \mathbb{N}$
Output: $c \in \mathbb{C}^n : c = x * y$

$\hat{x} \leftarrow \text{FFT}(x)$;
$\hat{y} \leftarrow \text{FFT}(y)$;
$\hat{c} \leftarrow 0_n$;
for $i \leftarrow 1$ **to** n **do**
  $\hat{c}_i \leftarrow \hat{x}_i \cdot \hat{y}_i$;
end
$c \leftarrow \text{FFT}^{-1}(\hat{c})$;

2.3 Étapes de l'algorithme

Après avoir vu la transformée de Fourier discrète et le produit de convolution des polynômes par la FFT, nous pouvons exposer les étapes de l'algorithme de Schönhage et Strassen.

Soient deux entiers $x, y \in \mathbb{Z}_b^n$ de base b. Voici les étapes :

(1) On écrit les entiers x et y comme des vecteurs de taille m tel que $m = 2^k, k \in \mathbb{N}$. C'est-à-dire :

$$\begin{cases} x = (x_1, x_2, ..., x_n, 0, ..., 0), & |x| = m \\ y = (y_1, y_2, ..., y_n, 0, ..., 0), & |y| = m \\ m = 2^k, k \in \mathbb{N} \end{cases}$$

On définit alors les deux polynômes $f, g \in \mathbb{Z}[X]$ tels que :

$$\begin{cases} x = \sum_{i=1}^{m} x_i b^{i-1} & = f(b) \\ y = \sum_{i=1}^{m} y_i b^{i-1} & = g(b) \end{cases}$$

Autrement dit, f et g sont les polynômes dont les coefficients sont les coordonnées dans la base b des vecteurs x et y respectivement. Il faut alors calculer le produit de convolution de f et g.

(2) On calcule la transformée de Fourier discrète de x et y, notée respectivement $\hat{x}$ et $\hat{y}$.

(3) On multiplie point par point les vecteurs $\hat{x}$ et $\hat{y}$, ce qui donne le vecteur $\hat{x} \circ \hat{y}$.

(4) On calcule la transformée de Fourier inverse de $\hat{x} \circ \hat{y}$, notée v. Le résultat calculé correspond alors au polynôme $f * g$. On arrondit les coefficients de v pour obtenir un vecteur d'entiers, c'est-à-dire pour que $f * g \in \mathbb{Z}[X]$.

(5) On évalue le polynôme $f * g$ obtenu en b. Cela revient à sommer les coefficients du vecteur v tout en effectuant les décalages dans la base b pour calculer les chiffres du résultat. On obtient donc le résultat de l'opération $x \times y$.

Ainsi, l'algorithme de Schönhage-Strassen se résume à la formule suivante :

$$x \times y = (f * g)(b)$$

2.4 Exemple

Prenons $x = 41$ et $y = 37$. En utilisant l'algorithme standard, on trouve 1517 :

$$\begin{array}{r} 4\ 1 \\ \times\ 3\ 7 \\ \hline 2\ 8\ 7 \\ 1\ 2\ 3\ \\ \hline 1\ 5\ 1\ 7 \end{array}$$

Retrouvons ce résultat avec l'algorithme de Schönhage et Strassen. Sous forme vectorielle, on a :

$$\begin{cases} x = (1, 4) \\ y = (7, 3) \end{cases}$$

(1) On écrit x et y sous forme de vecteurs de taille $2^2 = 4$:

$$\begin{cases} x = (1, 4, 0, 0) \\ y = (7, 3, 0, 0) \end{cases}$$

(2) On calcule la transformée de Fourier discrète de x et y, notée respectivement $\hat{x}$ et $\hat{y}$:

$$\begin{cases} \hat{x} = (5, 1 - 4i, -3, 1 + 4i) \\ \hat{y} = (10, 7 - 3i, 4, 7 + 3i) \end{cases}$$

(3) On multiplie point par point les vecteurs $\hat{x}$ et $\hat{y}$:

$$\hat{x} \circ \hat{y} = (50, -5 - 31i, -12, -5 + 31i)$$

(4) On calcule la transfomée de Fourier inverse de $\hat{x} \circ \hat{y}$, notée v :

$$v = (7, 31, 12, 0)$$

(5) On recombine pour obtenir le résultat :

$$\begin{array}{rr} & 7 \\ + & 3\ 1\ \bullet \\ + & 1\ 2\ \bullet\ \bullet \\ \hline & 1\ 5\ 1\ 7 \end{array}$$

D'où $41 \times 37 = 1517$.

Remarque. En pratique, on utilise cet algorithme pour les très grands entiers, c'est-à-dire ayant entre 10000 et 40000 chiffres. Cet exemple avec de petits entiers permet de bien comprendre le processus.

2.5 Pseudo-code

Voici le pseudo-code de l'algorithme de Schönhage et Strassen. On prend en entrée des entiers x et y de taille respective n et m.

Algorithm 2: Schönhage-Strassen algorithm

Input : $x \in \mathbb{Z}_b^n$, $y \in \mathbb{Z}_b^m$
Output: $p \in \mathbb{Z}_b^{n+m} : p = x \times y$

1 Set $k : \max(n, m) \leq 2^k, k \in \mathbb{N}$;
2 $X \leftarrow (x_1, ..., x_n, 0, ..., 0) : X \in \mathbb{Z}^{2^k}$;
3 $Y \leftarrow (y_1, ..., y_m, 0, ..., 0) : Y \in \mathbb{Z}^{2^k}$;
4 $v \leftarrow$ fastConvolution(X, Y);
5 $p \leftarrow 0_{2^k}$;
6 $c \leftarrow 0$; `// carry`
7 **for** $i \leftarrow 1$ **to** 2^k **do**
8 $\quad p_i \leftarrow (v_i + c) \mod b$;
9 $\quad c \leftarrow (v_i + c)/b$;
10 **end**

À la ligne 1, on choisit k tel que la taille des vecteurs X et Y donnés à la fonction de convolution rapide soit une puissance de 2 (voir implémentation).

2.6 Complexité

Pour calculer la complexité de l'algorithme, on prend en entrée deux entiers ayant n chiffres. La complexité de l'algorithme de Schönhage et Strassen repose sur celle de l'algorithme utilisé pour calculer le produit de convolution des polynômes f et g.

- Pour calculer la transformée de Fourier discrète avec la méthode naïve, la complexité serait en $\mathcal{O}(n^2)$. Grâce à la FFT, la complexité de cette étape est finalement en $\mathcal{O}(n \log n)$.
- Le produit point par point s'effectue en temps linéaire, $\mathcal{O}(n)$.
- La transformée de Fourier discrète inverse s'effectue en $\mathcal{O}(n \log n)$.

Au final, la complexité de l'algorithme est en $\mathcal{O}(n \log n \log \log n)$. La preuve est un peu complexe, mais on peut voir cette méthode comme la réduction d'une multiplication de deux entiers de n chiffres par $\mathcal{O}(n)$ multiplications de taille $\mathcal{O}(\log n)$ chacune. En notant M la complexité en temps de l'algorithme, on obtient alors la récursion suivante :

$$M(n) = \mathcal{O}(n(M(\log n))) = \mathcal{O}(n \log n(M(\log \log n))) = \dots$$

Cela en faisait la méthode asymptotiquement la plus rapide connue pour la multiplication d'entiers jusqu'en 2007, et la publication de l'algorithme de Fürer.

2.7 Implémentation

Voici l'implémentation en MATLAB de l'algorithme de Schönhage et Strassen [5].

```
1 function product = ssmult(x, y, b)
2     % Default base: 10
3     if nargin == 2
```

5. https://github.com/Quentin18/fft-fast-multiplication/blob/master/ssmult.m

```
4           b = 10;
5       end
6       % Choose the nearest highest power of 2 of the size of numbers
7       max_size = max(numel(int2str(x)), numel(int2str(y)));
8       k = pow2(ceil(log2(max_size)));
9       % Slit numbers into two vectors X and Y
10      X = number2vector(x, k);
11      Y = number2vector(y, k);
12      % Calculate the convolution
13      v = fastConvolution(X, Y);
14      % Recombinate
15      product = zeros(1, 2^k);
16      carry = 0;
17      for i = 1 : 2^k
18          product(i) = mod(v(i) + carry, b);
19          carry = fix((v(i) + carry) / b);
20      end
21      product = str2double(sprintf('%d', flip(product)));
22  end
```

Les lignes 3 et 4 permettent de définir la base à 10 si celle-ci n'est pas indiquée. On peut alors appeler la fonction *ssmult(x, y)* où x et y sont deux nombres en base 10.

Ensuite, avec les lignes 7 et 8, on choisit k tel qu'on puisse définir x et y comme des vecteurs de taille 2^k. On prend pour cela la puissance de 2 la plus proche supérieurement à la taille des nombres.

Avec les lignes 10 et 11, on convertit alors x et y en deux vecteurs nommés X et Y grâce à la fonction *number2vector*[6] suivante.

```
1  % Split a number into a vector of size 2^k
2  % Example: number2vector(12, 2) -> [2 1 0 0]
3  function v = number2vector(x, k)
4      v = zeros(1, 2^k);
5      str_x = int2str(x);
6      size_x = numel(str_x);
7      for i = 1 : size_x
8          v(i) = str2double(str_x(size_x - i + 1));
9      end
10 end
```

Cette fonction prend en entrée un entier x en base b et un entier k et retourne un vecteur de taille 2^k correspondant aux coordonnées de x dans la base b.

Une fois les vecteurs créés, on calcule leur produit de convolution avec la fonction *fastConvolution*[7], ligne 13.

```
1  % Returns the convolution of two vectors
2  function v = fastConvolution(X, Y)
3      v = round(ifft(fft(X) .* fft(Y)));
4  end
```

6. https://github.com/Quentin18/fft-fast-multiplication/blob/master/number2vector.m
7. https://github.com/Quentin18/fft-fast-multiplication/blob/master/fastConvolution.m

Cette fonction applique le théorème de convolution énoncé précédemment. Elle calcule la transformée de Fourier discrète des vecteurs X et Y avec *fft*, multiplie les vecteurs obtenus point par point avec l'opérateur "*.**", puis retourne la transformée de Fourier discrète inverse en utilisant *ifft*. On arrondit le vecteur final avec la fonction *round* afin d'obtenir un vecteur d'entiers.

Les lignes 15 à 20 permettent de recombiner le vecteur v obtenu par la convolution de X et Y afin de trouver les chiffres du résultat du produit nommé *product*.

Enfin, la ligne 21 permet de convertir le vecteur *product* en entier. La fonction *ssmult* renvoie alors le résultat de l'opération $x \times y$.

3 Autres algorithmes de multiplication

Dans cette partie, nous allons étudier trois autres algorithmes de multiplication :

- Algorithme standard (*Long multiplication*)
- Algorithme de Karatsuba
- Algorithme de Fürer

Leur complexité temporelle est aussi étudiée, et vous trouverez leur implémentation en MATLAB [8].

3.1 Algorithme standard

3.1.1 Description

L'algorithme standard, appelé *Long multiplication* en anglais, est la méthode de multiplication qui est enseignée à l'école primaire. Elle peut être utilisée pour multiplier des entiers de taille quelconque.

Cette méthode utilise la décomposition décimale des nombres. Elle consiste à multiplier chaque chiffre du premier nombre par chaque chiffre du second, puis de sommer les résultats en appliquant un décalage (*shift* en anglais). Pour utiliser cette méthode, il est donc juste nécessaire de connaître les tables de multiplication d'un chiffre par un autre.

3.1.2 Exemple

Voici un exemple où on multiplie 23958233 par 5830 en utilisant la méthode standard, ce qui donne 139676498390 :

$$
\begin{array}{rr}
 & 2\,3\,9\,5\,8\,2\,3\,3 \\
\times & \quad\quad 5\,8\,3\,0 \\
\hline
 & 7\,1\,8\,7\,4\,6\,9\,9\,\cdot \\
 & 1\,9\,1\,6\,6\,5\,8\,6\,4\, \\
1\,1\,9\,7\,9\,1\,1\,6\,5\, & \\
\hline
1\,3\,9\,6\,7\,6\,4\,9\,8\,3\,9\,0 &
\end{array}
$$

8. Sauf pour l'algorithme de Fürer

3.1.3 Pseudo-code

On considère deux entiers $x \in \mathbb{Z}_b^n$ et $y \in \mathbb{Z}_b^m$ de base b tel que $n \geq m$. Voici le pseudo-code décrivant ce processus.

Algorithm 3: Long multiplication algorithm

Input : $x \in \mathbb{Z}_b^n$, $y \in \mathbb{Z}_b^m$
Output: $p \in \mathbb{Z}_b^{n+m} : p = x \times y$

$p \leftarrow 0_{n+m}$;
for $i \leftarrow 1$ **to** m **do**
  $c \leftarrow 0$; // carry
  for $j \leftarrow 1$ **to** n **do**
    $p_{i+j-1} \leftarrow p_{i+j-1} + c + x_j + y_i$;
    $c \leftarrow p_{i+j-1}/b$;
    $p_{i+j-1} = p_{i+j-1} \mod b$;
  end
  $p_{i+n} = c$;
end

3.1.4 Complexité

Cette méthode exige que les n chiffres de x soient multipliés avec les m chiffres de y. Ce résultat est indépendant de la base b des deux nombres. Il en résulte que le nombre d'opérations élémentaires est de l'ordre de $\mathcal{O}(n \times m)$. On en déduit qu'avec cette méthode, le produit de deux nombres ayant n chiffres a une complexité de l'ordre de $\mathcal{O}(n^2)$.

Au milieu du XXème siècle encore, cet algorithme, utilisé depuis l'Égypte antique, paraissait optimal pour multiplier deux entiers. En effet, le mathématicien russe Andrey Kolmogorov a conjecturé en 1956 que la complexité minimale pour multiplier deux entiers de taille n est de l'ordre de $\mathcal{O}(n^2)$.

3.1.5 Implémentation

Voici l'implémentation en MATLAB de l'algorithme standard [9].

```
function product = longmult(x, y, b)
    % Default base: 10
    if nargin == 2
        b = 10;
    end
    str_x = reverse(num2str(x));
    str_y = reverse(num2str(y));
    p = numel(str_x);
    q = numel(str_y);
    product = zeros(1, p + q);
    for y_i = 1:q
        carry = 0;
        for x_i = 1:p
            product(x_i + y_i - 1) = product(x_i + y_i - 1) + carry ...
                + str2double(str_x(x_i)) * str2double(str_y(y_i));
```

9. https://github.com/Quentin18/fft-fast-multiplication/blob/master/longmult.m

```
16                carry = fix(product(x_i + y_i - 1) / b);
17                product(x_i + y_i - 1) = mod(product(x_i + y_i - 1), b);
18            end
19            product(y_i + p) = carry;
20        end
21        product = str2double(sprintf('%d', flip(product)));
22    end
```

Les lignes 3 et 4 permettent de définir la base à 10 si celle-ci n'est pas indiquée. On peut alors appeler la fonction *longmult(x, y)* où x et y sont deux nombres en base 10.

Pour manipuler la représentation vectorielle des deux entiers, on définit des chaînes de caractères qu'on inverse pour avoir les chiffres dans le bon ordre (lignes 6 et 7).

Le résultat est d'abord initialisé sous forme d'un vecteur (ligne 10), puis est converti en entier avec la commande ligne 21.

3.2 Algorithme de Karatsuba

3.2.1 Description

L'algorithme de Karatsuba est un algorithme de multiplication rapide. Il a été développé par Anatolli Alexevich Karatsuba en 1960 et publié en 1962.

On considère deux grands entiers $x, y \in \mathbb{Z}_b^n$. La méthode de Karatsuba consiste à multiplier x et y par trois multiplications de nombres plus petits. En effet, pour tout entier $k < n$, on peut scinder x et y en deux parties :

$$\begin{cases} x = x_1 \times b^k + x_0 \\ y = y_1 \times b^k + y_0 \end{cases}$$

Où $x_0 < b^k$ et $y_0 < b^k$. Ainsi, le produit xy devient :

$$xy = (x_1 \times b^k + x_0)(y_1 \times b^k + y_0) = x_1y_1 \times b^{2k} + (x_1y_0 + x_0y_1) \times b^k + x_0y_0$$

Le produit xy peut alors être effectué en calculant les quatre produits x_1y_1, x_1y_0, x_0y_1 et x_0y_0. En fait, il nécessite seulement les trois produits x_1y_1, x_0y_0 et $(x_0 + x_1)(y_0 + y_1)$ en regroupant les calculs sous la forme suivante :

$$xy = (x_1 \times b^k + x_0)(y_1 \times b^k + y_0) = x_1y_1 \times b^{2k} + ((x_0 + x_1)(y_0 + y_1) - (x_1y_1 + x_0y_0)) \times b^k + x_0y_0$$

La méthode peut être appliquée de manière récursive pour des grands nombres en calculant les trois produits avec l'algorithme de Karatsuba. C'est donc un algorithme de type *diviser pour régner*[10].

La multiplication par la base de numération b correspond à un décalage de chiffre, et les additions sont peu coûteuses en temps. Ainsi, le fait de calculer des grands nombres en trois produits au lieu de quatre optimise le temps de calcul, et donc la complexité temporelle de l'algorithme.

3.2.2 Exemple

On applique la méthode de Karatsuba pour calculer le produit 1237×2587. Ici, $n = 4$ et $b = 10$.

En utilisant l'algorithme standard, on trouve 3200119 :

10. Technique algorithmique consistant à découper un problème initial en sous-problèmes, afin de calculer sa solution à partir des solutions des sous-problèmes

$$\begin{array}{r} \times \begin{array}{r} 1\,2\,3\,7 \\ 2\,5\,8\,7 \end{array} \\ \hline \begin{array}{r} 8\,6\,5\,9 \\ 9\,8\,9\,6 \\ 6\,1\,8\,5 \\ 2\,4\,7\,4 \end{array} \\ \hline 3\,2\,0\,0\,1\,1\,9 \end{array}$$

Retrouvons ce résultat avec la méthode de Karatsuba :

- Pour calculer 1237×2587, on écrit $1237 \times 2587 = z_2 \times 10^4 + (z_1 - z_2 - z_0) \times 10^2 + z_0$ où $z_2 = 12 \times 25$, $z_1 = (12 + 37)(25 + 87)$ et $z_0 = 37 \times 87$.
  - Pour calculer 12×25, on écrit $12 \times 25 = z'_2 \times 10^2 + (z'_1 - z'_2 - z'_0) \times 10 + z_0$ où $z'_2 = 1 \times 2$, $z'_1 = (1 + 2)(2 + 5)$ et $z'_0 = 2 \times 5$.
    - Les calculs $1 \times 2 = 2$, $2 \times 5 = 10$ et $3 \times 7 = 21$ se réalisent en temps constants.
    - On obtient $z_2 = 12 \times 25 = 2 \times 100 + (21 - 2 - 10) \times 10 + 10 = 300$.
  - De la même façon, on obtient $z_1 = 49 \times 112 = 5488$.
  - De la même façon, on obtient $z_0 = 37 \times 87 = 3219$.
- D'où $1237 \times 2587 = 300 \times 10^4 + (5488 - 300 - 3219) \times 10^2 + 3219 = 3200119$.

En utilisant l'algorithme de Karatsuba, nous avons effectué seulement 9 multiplications de deux chiffres au lieu de 16 par la méthode standard.

Remarque. Dans cet exemple, les entiers x et y ont le même nombre de chiffres. Cependant, l'algorithme peut aussi être utilisé pour des entiers de taille différente (voir pseudo-code et implémentation).

3.2.3 Pseudo-code

La méthode de Karatsuba fonctionne pour n'importe quelle base b et n'importe quel entier k, mais l'algorithme récursif est plus efficace lorsque k est égal à $n/2$.

On considère deux entiers $x \in \mathbb{Z}_b^n$ et $y \in \mathbb{Z}_b^m$ de base b tel que $n \geq m$. Voici le pseudo-code décrivant ce processus.

Algorithm 4: Karatsuba algorithm

Input : $x \in \mathbb{Z}_b^n$, $y \in \mathbb{Z}_b^m$
Output: $p \in \mathbb{Z}_b^{n+m} : p = x \times y$

```
if x < b or y < b then
    p ← x × y;
else
    max_size ← max(size_base_x, size_base_y);
    k ← max_size/2;
    x1 ← E(x/b^k);
    x0 ← Mod(x, b^k);
    y1 ← E(y/b^k);
    y0 ← Mod(y, b^k);
    z2 ← karatsuba(x1, y1);
    z1 ← karatsuba(x1 + x0, y1 + y0);
    z0 ← karatsuba(x0, y0);
    p ← z2 × b^(2k) + (z1 − z0 − z2) × b^k + z0;
end
```

Entre les lignes 6 et 9, la fonction E désigne la partie entière et *Mod* désigne le reste de la division euclidienne.

Les lignes 10, 11, et 12 appellent récursivement la fonction pour calculer les trois produits définis précédemment.

3.2.4 Complexité

La méthode de Karatsuba fonctionne pour toute base b et n'importe quel entier k, mais elle est la plus efficace lorsque $n = 2^p, p \in \mathbb{N}$. Dans ce cas, la récursion s'arrête lorsque $n = 1$. L'algorithme effectue donc p fois la séparation des entiers, et le nombre de multiplications à un chiffre est de 3^p, ce qui donne n^c où $c = \log_2(3)$.

On note $T(n)$ le nombre total d'opérations élémentaires que l'algorithme effectue pour multiplier deux nombres à n chiffres. Comme on peut négliger le coût des additions et soustractions lorsque n augmente, $T(n)$ correspond au nombre de multiplications. On obtient alors :

$$T(n) = 3T(n/2)$$

D'après la récurrence, on obtient le résultat asymptotique suivant :

$$T(n) = \mathcal{O}(n^{\log_2(3)})$$

Par conséquent, la complexité de l'algorithme de Karatsuba est $\mathcal{O}(n^{\log_2(3)}) \approx \mathcal{O}(n^{1.585})$. Ceci est plus rapide que l'algorithme standard. Par exemple, pour $n = 1000$, $n^{\log_2(3)}$ est de l'ordre de 50000 alors que $n^2 = 1000000$.

3.2.5 Implémentation

Voici l'implémentation en MATLAB de l'algorithme de Karatsuba [11].

```
1  function product = karatsuba(x, y, b)
2      % Default base: 10
3      if nargin == 2
4          b = 10;
5      end
6      % If x < b or y < b : general multiplication
7      if (x < b || y < b)
8          product = x * y;
9      % Karatsuba algo
10     else
11         % Calculate the split size k of the numbers
12         max_size = max(numel(int2str(x)), numel(int2str(y)));
13         k = pow2(ceil(log2(max_size / 2)));
14         % Split x and y about the middle
15         x1 = floor(x / (b^k));
16         x0 = mod(x, b^k);
17         y1 = floor(y / (b^k));
18         y0 = mod(y, b^k);
19         % 3 calls multiplications to numbers approximately half the size
20         z2 = karatsuba(x1, y1, b);
```

11. https://github.com/Quentin18/fft-fast-multiplication/blob/master/karatsuba.m

```
21          z1 = karatsuba(x1 + x0, y1 + y0, b);
22          z0 = karatsuba(x0, y0, b);
23          % Result
24          product = z2*b^(2*k) + (z1 - z2 - z0)*b^k + z0;
25      end
26  end
```

3.3 Algorithme de Fürer

3.3.1 Description

L'algorithme de Fürer est un algorithme de multiplication pour les très grands entiers. Il a été publié en 2007 par le mathématicien suisse Martin Fürer. Cet algorithme est aussi basé sur la FFT, et il possède asymptotiquement une meilleure complexité que l'algorithme de Schönhage-Strassen.

Pour cet algorithme, nous ne proposons pas d'implémentation dû à la difficulté de le mettre en œuvre.

3.3.2 Complexité

L'algorithme de Fürer possède une complexité en $\mathcal{O}(n \log n K^{\log^* n})$ où $K > 1$ et $\log^* n$ désigne le logarithme itéré défini comme cela :

$$\log^* n = \begin{cases} 0 & \text{si } n \leq 1 \\ 1 + \log^*(\log n) & \text{si } n > 1 \end{cases}$$

Le logarithme itéré d'un nombre n correspond au nombre de fois que le logarithme doit lui être appliqué avant que le résultat soit inférieur ou égal à 1.

La différence entre les termes $\log \log n$ et $K^{\log^* n}$ est asymptotiquement à l'avantage du second. Cependant, le régime asymptotique n'est atteint que pour les très grands nombres.

Au fil des années, l'algorithme a été démontré pour de nouvelles constantes K, améliorant ainsi sa complexité. En 2014, Harvey, van der Hoeven et Lecerf publient un article qui montre une optimisation de l'algorithme permettant de passer à $K = 8$, puis à $K = 4$. La validité de cette optimisation repose néanmoins sur une conjecture sur les nombres premiers de Mersenne [12].

4 Comparaison des algorithmes

Après avoir étudié différents algorithmes de multiplication, nous allons comparer leur complexité. On considère pour cela le produit de deux entiers de taille n.

Tout d'abord, l'algorithme standard a une complexité en temps quadratique, en $\mathcal{O}(n^2)$.

En 1962, l'algorithme de Karatsuba a été publié. Sa complexité est en $\mathcal{O}(n^{\log_2(3)}) \approx \mathcal{O}(n^{1.585})$, ce qui en fait le premier algorithme ayant une complexité sub-quadratique. Ceci est bien plus rapide que l'algorithme standard. Pour donner une idée, pour $n = 1000$, $n^{\log_2(3)}$ est de l'ordre de 50000 alors que $n^2 = 1000000$.

En 1963, l'algorithme de Toom-Cook, qui est une amélioration de l'algorithme de Karatsuba, permet de passer à une complexité de l'ordre de $\mathcal{O}(n^{\log_3(5)}) \approx \mathcal{O}(n^{1.465})$.

Ensuite, en 1971, grâce à la FFT, l'algorithme de Schönhage et Strassen arrive à une complexité de l'ordre de $\mathcal{O}(n \log n \log \log n)$, ce qui en fait la méthode asymptotiquement la plus rapide.

Il faut alors attendre 2007 et l'algorithme de Fürer pour avoir un meilleur algorithme que celui de Schönhage et Strassen. Sa complexité est en $\mathcal{O}(n \log n K^{\log^* n})$, où $\log^* n$ désigne le logarithme itéré.

12. Nombres premiers de la forme $2^n - 1, n \in \mathbb{N}^*$

Voici un tableau récapitulatif de la complexité temporelle des algorithmes de multiplication en fonction du nombre de chiffres des entiers en entrée.

Année	Algorithme	Complexité
1962	Karatsuba	$\mathcal{O}(n^{\log_2(3)}) \approx \mathcal{O}(n^{1.585})$
1963	Toom-Cook	$\mathcal{O}(n^{\log_3(5)}) \approx \mathcal{O}(n^{1.465})$
1971	Schönhage-Strassen	$\mathcal{O}(n \log n \log\log n)$
2007	Fürer	$\mathcal{O}(n \log n K^{\log^* n})$

TABLE 1 – Complexité temporelle des algorithmes de multiplication rapide

5 Conclusion

On peut voir avec cette étude que la transformation de Fourier rapide a permis d'énormes progrès en arithmétique rapide. En effet, l'algorithme de Schönhage et Strassen est un exemple d'utilisation de la FFT. La transformation de Fourier rapide est alors un outil puissant dans la quête de l'optimisation algorithmique, permettant de passer d'une complexité polynomiale en complexité logarithmique.

Références

[1] *Fürer's algorithm*. URL : https://en.wikipedia.org/wiki/F%C3%BCrer%27s_algorithm.

[2] David Harvey. « Integer multiplication in time O(n log n) ». In : 2019. URL : https://web.maths.unsw.edu.au/~davidharvey/talks/nlogn-unsw2019.pdf.

[3] *Karatsuba algorithm*. URL : https://en.wikipedia.org/wiki/Karatsuba_algorithm.

[4] *Multiplication algorithm*. URL : https://en.wikipedia.org/wiki/Multiplication_algorithm.

[5] *Schönhage–Strassen algorithm*. URL : https://en.wikipedia.org/wiki/Sch%C3%B6nhage%E2%80%93Strassen_algorithm.

[6] Raazesh Sainudiin Thomas Steinke. « A Rigorous Extension of the Schönhage-Strassen Integer Multiplication Algorithm Using Complex Interval Arithmetic ». In : (2010). URL : https://arxiv.org/pdf/1006.0405.pdf.

List of Algorithms

Listings

Implémentation en MATLAB : https://github.com/Quentin18/fft-fast-multiplication