

Implémentations efficaces de calculs sur les polynômes à une variable : FFT

Pierre Lin, Enzo Roaldes

L2 DM-IM 2022, Sorbonne Université

13 Juillet 2022



Introduction

Les polynômes sont :

- un objet mathématique fondamental
- omniprésents dans notre quotidien

Plan de la soutenance :

- Algorithme Naïf & de Karatsuba
- Fast Fourier Transform (FFT)
- Vectorisation avec AVX2

Langage utilisé : C

Algorithme Naïf

Les polynômes sont représentés par des tableaux contenant leurs coefficients.

L'algorithme naïf est basé sur la formule :

$$R(X) = PQ(X) = \sum_{i=0}^n \sum_{j=0}^n p_i q_j X^{i+j}$$

Complexité = nombre d'opérations élémentaires.

Complexité de l'algorithme naïf : $O(n^2)$

Algorithme de Karatsuba

Principe de l'algorithme de Karatsuba :

- Décomposition récursive de P et Q :

$$P = P_1 + X^{n/2}P_2 \quad Q = Q_1 + X^{n/2}Q_2$$

- Reconstruction du résultat R : $R = E_1 + X^{n/2}(E_3 - E_2 - E_1) + X^n E_2$
avec $E_1 = P_1 Q_1$; $E_2 = P_2 Q_2$ et $E_3 = (P_1 + P_2)(Q_1 + Q_2)$

Complexité : $O(n^{1.58})$

Degré de P et Q	Naïf	Karatsuba
2^{15}	0.429183	0.085549
2^{16}	1.761229	0.242966
2^{17}	6.875559	0.645986
2^{18}	28.165627	2.023236

Tableau – Temps de l'algorithme naïf et de l'algorithme de Karatsuba.

Quelques notions fondamentales

1 $n = 2^k$ sera la taille du tableau du polynôme final $R = PQ$

Quelques notions fondamentales

- 1 $n = 2^k$ sera la taille du tableau du polynôme final $R = PQ$
- 2 Le corps $\mathbb{Z}/p\mathbb{Z}$

Quelques notions fondamentales

- 1 $n = 2^k$ sera la taille du tableau du polynôme final $R = PQ$
- 2 Le corps $\mathbb{Z}/p\mathbb{Z}$
- 3 Racine primitive $p - 1$ -ième de l'unité r : $r^{p-1} \% p = 1$ et que $r^k \% p \neq 1$ pour tout k dans $\{1, \dots, p - 2\}$

Quelques notions fondamentales

- 1 $n = 2^k$ sera la taille du tableau du polynôme final $R = PQ$
- 2 Le corps $\mathbb{Z}/p\mathbb{Z}$
- 3 Racine primitive $p - 1$ -ième de l'unité $r : r^{p-1} \% p = 1$ et que $r^k \% p \neq 1$ pour tout k dans $\{1, \dots, p - 2\}$
- 4 Racine principale n -ième de l'unité : $r_principale = r^{\frac{p-1}{n}} \% p$
 $\Rightarrow p - 1$ doit être divisible par n

Quelques notions fondamentales

- 1 $n = 2^k$ sera la taille du tableau du polynôme final $R = PQ$
- 2 Le corps $\mathbb{Z}/p\mathbb{Z}$
- 3 Racine primitive $p - 1$ -ième de l'unité r : $r^{p-1} \% p = 1$ et que $r^k \% p \neq 1$ pour tout k dans $\{1, \dots, p-2\}$
- 4 Racine principale n -ième de l'unité : $r_{\text{principale}} = r^{\frac{p-1}{n}} \% p$
 $\Rightarrow p - 1$ doit être divisible par n
- 5 Choix du nombre premier $p = 754974721 = 1 + 2^{24} * 3^2 * 5$
 $\Rightarrow \deg_{\max}(R) = 2^{24}$
racine primitive $p - 1$ -ième de l'unité : $r = 11$

Algorithme de multiplication par FFT

Théorème (Interpolation de Lagrange)

Soient a_1, \dots, a_n et b_1, \dots, b_n dans $\mathbb{Z}/p\mathbb{Z}$ (avec les a_i deux à deux distincts). Alors il existe un unique polynôme P de degré $< n$ tel que :

$$\forall i \in \{1, \dots, n\}, P(a_i) = b_i.$$

Algorithme de multiplication par FFT

Entrée. P et Q deux polynômes, n une puissance de 2 avec $n > \deg(PQ)$ et ω une racine principale n -ième de l'unité.

Sortie. $R = PQ$

Algorithme.

1. *Pré-calcul.* Calculer les puissances $\omega^0, \omega^1, \omega^2, \dots, \omega^{n-1} \rightarrow O(n)$
2. *Évaluation (DFT).* Calculer les valeurs :
 $Eval(P) = (P(\omega^0), \dots, P(\omega^{n-1})); Eval(Q) = (Q(\omega^0), \dots, Q(\omega^{n-1})).$
3. *Produit point à point.* $Eval(R) = (PQ(\omega^0), \dots, PQ(\omega^{n-1})) \rightarrow O(n)$
4. *Interpolation.* Retrouver R grâce à $Eval(R)$.

Discrete Fourier Transform (DFT)

Principe de la DFT (diviser pour régner) :

Soient $n = 2^k$ et $P(X) = p_n X^n + \dots + p_1 X + p_0$.

Soient P_0, P_1 les polynômes composés des coefficients de rang respectivement pair et impair de P :

$P_0(X) = p_n X^{n/2} + \dots + p_2 X + p_0$ et $P_1(X) = p_{n-1} X^{(n-2)/2} + \dots + p_3 X + p_1$.

Nous avons alors que $P(X) = P_0(X^2) + X P_1(X^2)$.

→ Algorithme récursif où nous divisons n par 2

DFT

Algorithme DFT

Entrée. $P = p_0 + \dots + p_{n-1}X^{n-1}$, n une puissance de 2 et le tableau *racines* = $[1, \omega, \dots, \omega^{n-1}]$ où ω est une racine n -ième principale de l'unité.

Sortie. $P(1), \dots, P(\omega^{n-1})$.

Algorithme.

1. Si $n = 1$, renvoyer p_0 .
2. Sinon, soit $k = n/2$. Calculer :

$$R_0(X) = \sum_{i=0}^{k-1} (p_i + p_{i+k}) X^i$$

$$R_1(X) = \sum_{i=0}^{k-1} (p_i - p_{i+k}) \omega^i X^i$$

3. Calculer récursivement $R_0(1), R_0(\omega^2), \dots, R_0((\omega^2)^{k-1})$
et $R_1(1), R_1(\omega^2), \dots, R_1((\omega^2)^{k-1})$.
4. Renvoyer $R_0(1), R_1(1), R_0(\omega^2), R_1(\omega^2), \dots, R_0((\omega^2)^{k-1}), R_1((\omega^2)^{k-1})$.

Opérations modulo p

Opération $\%$ beaucoup plus lente que les opérations $+$, $-$, $*$.

Addition modulo p :

$$(a + b) \% p = \begin{cases} a + b - p & \text{si } a + b \geq p \\ a + b & \text{sinon} \end{cases}$$

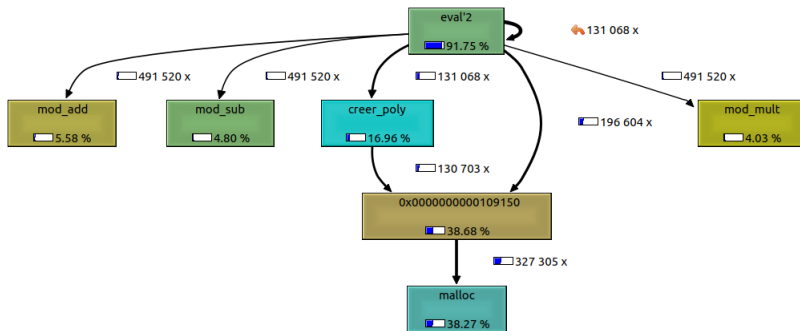
Soustraction modulo p :

$$(a - b) \% p = \begin{cases} p - (b - a) & \text{si } b > a \\ a - b & \text{sinon} \end{cases}$$

Multiplication modulo p : cast en *long* puis utilisation de l'opérateur $\%$.

Première version DFT

Malloc des tableaux R_0 , R_1 et *racines* à chaque récursion.



→ Utilisation d'un *pas* pour le tableau *racines*.

→ Utilisation du tableau des coefficients de P pour R_0 et R_1 .

DFT

Algorithme DFT

Entrée. $P = p_0 + \dots + p_{n-1}X^{n-1}$, n une puissance de 2 et le tableau *racines* = $[1, \omega, \dots, \omega^{n-1}]$ où ω est une racine n -ième principale de l'unité.

Sortie. $P(1), \dots, P(\omega^{n-1})$.

Algorithme.

1. Si $n = 1$, renvoyer p_0 .
2. Sinon, soit $k = n/2$. Calculer :

$$R_0(X) = \sum_{i=0}^{k-1} (p_i + p_{i+k}) X^i$$

$$R_1(X) = \sum_{i=0}^{k-1} (p_i - p_{i+k}) \omega^i X^i$$

3. Calculer récursivement $R_0(1), R_0(\omega^2), \dots, R_0((\omega^2)^{k-1})$
et $R_1(1), R_1(\omega^2), \dots, R_1((\omega^2)^{k-1})$.
4. Renvoyer $R_0(1), R_1(1), R_0(\omega^2), R_1(\omega^2), \dots, R_0((\omega^2)^{k-1}), R_1((\omega^2)^{k-1})$.

AVX2

Vectorisation avec AVX2 :

- opérations élémentaires (+, −, *, ...) sur des "vecteurs" de données similaires à des tableaux.
- type `__m256i` : stock 256 bits d'entiers → stockage de 8 entiers de 32 bits.
- opérations sur deux `__m256i` aussi rapide que sur deux entiers `int`.
→ possibilité d'un gain de temps fois 8 en utilisant AVX2.

Exemple :

```
__m256i a = _mm256_set_epi32(1, 2, 3, 4, 5, 6, 7, 8);  
__m256i b = _mm256_set_epi32(1, 0, 1, 0, 1, 0, 1, 0);  
__m256i x = _mm256_add_epi32(a, b);  
// nous avons x = [2, 2, 4, 4, 6, 6, 8, 8]
```


Opérations modulo p avec AVX2

Pas d'opération modulo en AVX2 !

Comment faire le modulo sur des vecteurs ?

Soient $p = 7$, $\vec{a} = [6, 2]$, $\vec{b} = [4, 1]$ et $\vec{x} = \vec{a} + \vec{b} = [10, 3]$. **Problème !**

Opérations modulo p avec AVX2

Pas d'opération modulo en AVX2 !

Comment faire le modulo sur des vecteurs ?

Soient $p = 7$, $\vec{a} = [6, 2]$, $\vec{b} = [4, 1]$ et $\vec{x} = \vec{a} + \vec{b} = [10, 3]$. **Problème !**

→ Fonction AVX2 `_mm256_min_epu32(\vec{a} , \vec{b})` : permet de prendre le minimum "positif" entre chaque élément de \vec{a} et \vec{b} .

Exemple

$\text{min_pos}([1, 4], [0, -10]) = \text{min}([1, 4], [0, \text{UINT_MAX}-10]) = [0, 4]$.

Pour l'addition, avec $\vec{x} = \vec{a} + \vec{b}$, on a $\vec{x} \% \vec{p} = \text{min_pos}(\vec{x}, \vec{x} - \vec{p})$.

Preuve

Si $\vec{x}[i] \geq p$, alors l'appel retourne $\vec{x}[i] - p$.

Sinon, $\vec{x}[i] < p$, donc $\vec{x}[i] - p < 0$, et après conversion en `Uint` : $\vec{x}[i] - p > \vec{x}[i]$ et donc l'appel retourne $\vec{x}[i]$.

De même, pour la soustraction, avec $\vec{x} = \vec{a} - \vec{b}$, on a : $\vec{x} \% \vec{p} = \text{min_pos}(\vec{x}, \vec{x} + \vec{p})$.

Opérations modulo p avec AVX2

```
void vect_mod_add(Uint *res1, Uint *tab1, Uint *tab2) {
    __m256i p = _mm256_set1_epi32(NB_P);
    __m256i a = _mm256_loadu_si256((__m256i *) tab1);
    __m256i b = _mm256_loadu_si256((__m256i *) tab2);
    __m256i x = _mm256_add_epi32(a, b);
    __m256i result = _mm256_min_epu32(x, _mm256_sub_epi32(x,
        p));
    _mm256_storeu_si256((__m256i *) res1, result);
}
```

Multiplication : Algorithme de réduction de Barrett

Opération	Sans AVX2	Avec AVX2
Addition	0.012610	0.006697
Soustraction	0.014247	0.007139
Multiplication	0.015053	0.007497

Tableau – Temps des opérations d'addition, soustraction et multiplication sans et avec AVX2.

DFT

Algorithme DFT

Entrée. $P = p_0 + \dots + p_{n-1}X^{n-1}$, n une puissance de 2 et le tableau *racines* = $[1, \omega, \dots, \omega^{n-1}]$ où ω est une racine n -ième principale de l'unité.

Sortie. $P(1), \dots, P(\omega^{n-1})$.

Algorithme.

1. Si $n = 1$, renvoyer p_0 .
2. Sinon, soit $k = n/2$. Calculer :

$$R_0(X) = \sum_{i=0}^{k-1} (p_i + p_{i+k}) X^i$$

$$R_1(X) = \sum_{i=0}^{k-1} (p_i - p_{i+k}) \omega^i X^i$$

3. Calculer récursivement $R_0(1), R_0(\omega^2), \dots, R_0((\omega^2)^{k-1})$
et $R_1(1), R_1(\omega^2), \dots, R_1((\omega^2)^{k-1})$.
4. Renvoyer $R_0(1), R_1(1), R_0(\omega^2), R_1(\omega^2), \dots, R_0((\omega^2)^{k-1}), R_1((\omega^2)^{k-1})$.

Suite de la multiplication par FFT

L'étape d'interpolation dans la multiplication par FFT :

- 1 Calculer les puissances successives de l'inverse de la racine principale : $\omega^0, \omega^{-1}, \omega^{-2}, \dots, \omega^{-(n-1)}$.
- 2 Appliquer la DFT à $Eval(R)$ avec ces inverses.
- 3 Diviser les coefficients obtenus suite à la DFT par n (c'est-à-dire multiplier par l'inverse de n dans $\mathbb{Z}/p\mathbb{Z}$).

Comment calculer ω^{-1} et n^{-1} dans $\mathbb{Z}/p\mathbb{Z}$?

→ Algorithme d'Euclide étendu appliqué à l'équation $au + pv = 1$.

Théorème de Bézout

Soient a et b deux entiers naturels non nuls. a et b sont premiers entre eux si et seulement si il existe deux entiers relatifs u et v tels que $au + bv = 1$.

Preuve que u est l'inverse de a

- 1 a et p sont premiers entre eux donc $\exists u, v \in \mathbb{Z}$ tels que $au + pv = 1$
- 2 Si $au + pv = 1$, alors dans $\mathbb{Z}/p\mathbb{Z}$:

$$au + pv = (au + pv) \% p = (au) \% p + (pv) \% p = (au) \% p = 1.$$
- 3 Donc $au = 1$ dans $\mathbb{Z}/p\mathbb{Z}$ et u est l'inverse de a dans $\mathbb{Z}/p\mathbb{Z}$.

Améliorations

n	FFT Sans AVX2	FFT Avec AVX2
2^{20}	0.158404	0.150727
2^{21}	0.503089	0.437402
2^{22}	1.240285	1.075995
2^{23}	3.028623	2.317684
2^{24}	7.108149	5.847869

Rechercher les cases désirées dans *racines* avec AVX2 prend trop de temps.

- Réutilisation du *malloc* pour le tableau *racines*.
- Gain de temps sur les polynômes de degré > 20 .

n	FFT Sans AVX2	FFT Avec AVX2 (Version 2)
2^{20}	0.158404	0.180472
2^{21}	0.503089	0.355293
2^{22}	1.240285	0.706925
2^{23}	3.028623	1.504402
2^{24}	7.108149	2.939824

La bibliothèque NTL

Meilleure bibliothèque pour la FFT dans $\mathbb{Z}/p\mathbb{Z}$: NTL

Comparaison des temps obtenus :

n	No AVX2 (NTL)	No AVX2	AVX2 (NTL)	AVX2
2^{18}	0.00297	0.01017	0.00182	0.00944
2^{20}	0.01517	0.06008	0.01113	0.06168
2^{22}	0.08613	0.38058	0.07253	0.21169

NTL est 3 à 5 fois plus rapide que nous !

Temps finaux et Conclusion

n	Naïf	Karatsuba	FFT Sans AVX2	FFT Avec AVX2 V2
2^{18}	28.17	2.02	0.037	0.045
2^{19}	2 m	6.07	0.085	0.094
2^{20}	7.5 m	18.21	0.16	0.18
2^{21}	30 m	54.63	0.50	0.36
2^{22}	2 h	2.7 m	1.24	0.71
2^{23}	8 h	8.1 m	3.03	1.50
2^{24}	32 h	24.5 m	7.11	2.94

Tableau – Comparaison finale des algorithmes implémentés

Attention ! Le degré des polynômes multipliés par FFT est limité par le choix de p .
D'un autre côté : Karatsuba et Naïf peuvent multiplier n'importe quoi !

Temps de nos algorithmes ≈ 5 fois le temps sur NTL

Algorithmes plus performants d'ici là ? Dépassé par de nouvelles technologies ?