Produit de polynômes

Sergey Dovgal, Pierre Rousselin, Mehdi Naima, Alexandros Singh

25 février 2022

L'objectif de ce TP est d'implémenter certains algorithmes de calcul polynomial vus en cours, avec un accent sur le produit des polynômes : méthode naïve, algorithme de Karatsuba et enfin le produit via la FFT (algorithme de Schönhage–Strassen).

Une petite bibliothèque de manipulation des polynômes (à coefficients complexes) est déjà partiellement écrite. Vous la trouverez en tapant cette commande dans votre terminal :

```
wget https://wintershammer.github.io/teaching/TP2_G2PI_2022.zip
```

1 Découverte de la bibliothèque `polynomes`

1.1 Les nombres complexes en C

Les types complexes sont apparus dans le langage C avec le standard C99 (en-tête `<complex.h>`). Pour déclarer une variable complexe, on utilise le mot-clé `complex`, suivi du type flottant que l'on veut utiliser pour les parties réelles et imaginaires.

```
complex double z1;
complex float z2;
complex long double z3;
```

Pour simplifier les déclarations, la bibliothèque `polynomes` contient l'alias de type

```
typdef complex double complexe;
```

Les opérations sur les complexes se font de façon naturelle (avec `+ - * /`). Le nombre imaginaire i est accessible via la macro `I`. Les fonctions `creal`, `cimag`, `conj` et `cabs` permettent d'obtenir, respectivement, les parties réelles et imaginaires, le conjugué et la valeur absolue.

Il n'y a pas de spécificateur de format pour imprimer un complexe, donc la bibliothèque `polynomes` contient les fonctions utilitaires

```
char *sprint_cplx(char *s, complexe z);
void print_cplx(complexe z);
```

1.2 Le type `struct polynome`

On représente les polynômes de la façon suivante :

```
struct polynome {
    complexe *coeffs; /* tableau de taille "taille", */
    int taille; /* doit être une puissance de 2 */
};
```

Par exemple la suite de commandes suivante

```
    struct polynome *p = (struct polynome *) malloc(sizeof(struct ↩
   polynome));
    p->taille = 3;
    p->coeffs = (complexe*) malloc(3 * sizeof(complexe));
    p->coeffs[0] = 1;
    p->coeffs[1] = I;
    p->coeffs[2] = 3;
```

donne le polynôme :

$$P(X) = 1 + IX^1 + 3X^2.$$

2 Échauffement

Implémenter les fonctions

```
complexe pn_eval(const struct polynome *p, complexe z);
struct polynome *pn_produit(const struct polynome *p, const struct ↩
   polynome *q);
```

en respectant les spécifications données dans le fichier `polynomes.h`.

Tester chaque fonction en mettant à 1 (au lieu de 0) la constante symbolique associée dans le programme de test `test-pn.c` : par exemple, pour tester la fonction `pn_eval`, il suffit, avant de compiler, de changer

```
#define TEST_EVAL 0
```

en

```
#define TEST_EVAL 1
```

3 Karatsuba

On rappelle le principe de l'algorithme de Karatsuba : Soient P et Q deux polynômes de degré inférieur ou égal à $2^n - 1$, avec $n \in \mathbb{N}$: On peut écrire P de la façon suivante :

$$P(X) = \sum_{i=0}^{2^n-1} \alpha_i X^i = P_g(X) + X^{2^{n-1}} P_d(X),$$

où P_g et P_d sont les polynômes de degré inférieur ou égal à $2^{n-1} - 1$:

$$P_g(X) = \sum_{i=0}^{2^{n-1}-1} \alpha_i X^i \quad \text{et} \quad P_d(X) = \sum_{i=0}^{2^{n-1}-1} \alpha_{2^{n-1}+i} X^i$$

et de même pour Q. Le produit PQ peut alors être écrit de la façon (très astucieuse !) suivante :

$$R(X) = PQ(X) = P_g Q_g(X) + ((P_g + P_d)(Q_g + Q_d) - P_g Q_g - P_d Q_d) X^{2^{n-1}} + X^{2^n} P_d Q_d(X).$$

et l'on voit qu'il ne faut plus calculer que 3 produits de polynômes dont la taille a été divisée par 2, ces produits sont :

$$R_g = P_g Q_g \quad R_d = P_d Q_d \quad R_s = P_s Q_s,$$
$$\text{où } P_s = (P_g + P_d) \quad \text{et} \quad Q_s = (Q_g + Q_d).$$

On peut alors récursivement calculer ces trois produits. Une fonction `pn_karatsuba` est déjà écrite pour vous (ne vous réjouissez pas trop vite, c'est `aux_karatsuba` qui fait tout le travail).

Question : La fonction `pn_karatsuba` utilise une fonction

```
static struct polynome *aux_karatsuba(complexe *p, complexe *q, int ↩
    taille);
```

qui prend deux adresses de tableaux de complexes de taille `taille` (qui est une puissance de 2) et retourne l'adresse d'un tableau nouvellement alloué de taille $2 \times$ `taille` contenant le produit des deux tableaux calculé par l'algorithme de Karatsuba.

Implémentez la fonction `aux_karatsuba`. Vous pouvez introduire des fonctions supplementaires selon vos besoins, par exemple, une fonction qui calcule la somme des parties gauche et droite d'un tableau.

Testez votre fonction et sa performance à l'aide du programme de test fourni.

4 Produit via la FFT

4.1 Fast Fourier Transform

Soit P un polynôme de degré inférieur à n. La transformée de Fourier discrète à n points du polynôme P est le vecteur de $\mathbb{C}^n$:

$$\mathcal{F}_n(P) = \left[P(\omega_n^0), P(\omega_n^1), \ldots, P(\omega_n^{n-1})\right],$$

où $\omega_n = \exp(2i\pi/n)$.

Question : Avec une méthode « naïve », quelle est la complexité du calcul de cette transformée de Fourier ?

L'algorithme FFT, très astucieux, utilise le principe « diviser pour régner » pour avoir une complexité en $O(n\log(n))$. À partir de maintenant, on suppose que n est une puissance de 2 : $n = 2^k$.

On écrit

$$P(X) = \sum_{i=0}^{2^k-1} a_i X^i = P_{\text{pair}}(X^2) + X P_{\text{impair}}(X^2),$$

où $P_{\text{pair}}(X) = \sum_{i=0}^{2^{k-1}-1} a_{2i} X^i$ est le polynôme obtenu en ne conservant que les coefficients de degré pairs de P et $P_{\text{impair}}(x) = \sum_{i=0}^{2^{k-1}-1} a_{2i+1} X^i$ est celui qu'on obtient avec les termes de degré impair de P. Remarquer que la taille des polynômes P_{pair} et de P_{impair} est deux fois moindre que celle de P. On suppose qu'on sait calculer la transformée de Fourier discrète à $n/2$ points de P_{pair} et P_{impair}. Alors (magie...), pour tout i entre 0 et $n-1$,

$$P(\omega_n^i) = P_{\text{pair}}((\omega_n^i)^2) + \omega_n^i P_{\text{impair}}((\omega_n^i)^2) = P_{\text{pair}}(\omega_{n/2}^i) + \omega_n^i P_{\text{impair}}(\omega_{n/2}^i).$$

On remarque que si $i \geq n/2$, alors on peut écrire $i = n/2 + j$ et

$$\omega_{n/2}^i = \omega_{n/2}^{n/2+j} = \omega_{n/2}^j$$

On peut donc calculer la transformée de Fourier à n points de P grâce à la transformée de Fourier à $n/2$ points de P_{pair} et P_{impair} (qui sont de taille $n/2$) :

$$\begin{aligned}
P(\omega_n^0) &= P_{\text{pair}}(\omega_{n/2}^0) + \omega_n^0 P_{\text{impair}}(\omega_{n/2}^0) \\
P(\omega_n^1) &= P_{\text{pair}}(\omega_{n/2}^1) + \omega_n^1 P_{\text{impair}}(\omega_{n/2}^1) \\
&\dots \\
P(\omega_n^{n/2-1}) &= P_{\text{pair}}(\omega_{n/2}^{n/2-1}) + \omega_n^{n/2-1} P_{\text{impair}}(\omega_{n/2}^{n/2-1}) \\
P(\omega_n^{n/2}) &= P_{\text{pair}}(\omega_{n/2}^0) + \omega_n^{n/2} P_{\text{impair}}(\omega_0^{n/2}) \\
P(\omega_n^{n/2+1}) &= P_{\text{pair}}(\omega_{n/2}^1) + \omega_n^{n/2+1} P_{\text{impair}}(\omega_{n/2}^1) \\
&\dots \\
P(\omega_n^{n-1}) &= P_{\text{pair}}(\omega_{n/2}^{n/2-1}) + \omega_n^{n-1} P_{\text{impair}}(\omega_{n/2}^{n/2-1})
\end{aligned}$$

Description informelle de l'algorithme :

L'algorithme est récursif et calcul le résultat directement dans le tableau des entrées *tab*.

1. Case de base : Si la taille actuelle du tableau de complexes est égale à 2 (donc $P(X) = A + BX$), alors $tab[0] = A + B$ et $tab[1] = A - B$.

2. Sinon on sépare le polynôme comme décrit ci-dessus en 2 polynômes (pair et impair). On appelle récursivement la transformée de Fourier rapide sur chacun des deux polynômes puis on recombine les résultats comme décrit auparavant. On pensera aussi à exploiter le fait que pour $k \geq \frac{n}{2}$, $\omega_n^k = -\omega_n^{k-\frac{n}{2}}$.

Question : Implémenter la fonction

```
int fft(complexe *tab, const complexe *racines, int taille, int ↩
    taille_racines);
```

qui prend une tableau de complexes `tab` de taille `taille`, un tableau de racines de l'unités de taille `taille_racines` (qui est un multiple de `taille`) et met dans `tab` la transformée de Fourier à `taille` points du polynôme dont les coefficients étaient les éléments de `tab`.

4.2 Transformée inverse

La transformée de Fourier inverse à n points d'un vecteur

$$[\hat{\alpha}_0, \hat{\alpha}_1, \ldots, \hat{\alpha}_{n-1}]$$

est le tableau

$$\left[\frac{1}{n}\hat{P}(\omega_n^{-0}), \frac{1}{n}\hat{P}(\omega_n^{-1}), \ldots, \frac{1}{n}\hat{P}(\omega_n^{-(n-1)})\right]$$

où le polynôme $\hat{P}$ est donné par

$$\hat{P}(X) = \sum_{i=0}^{n-1} \hat{\alpha}_i.$$

En fait, presque tout le travail est déjà fait car $\omega_n^{-i} = \omega_n^{n-i}$ donc il suffit de calculer la transformée de Fourier à n points de $\hat{P}$, de diviser ses coefficients par n et de les réordonner.

Question : Implémenter la fonction

```
int ifft(complexe *tab, const complexe *racines, int taille, int ↩
    taille_racines);
```

qui met dans le tableau `tab` le résultat de la transformée de Fourier inverse des coefficients initialement présents dans `tab` en utilisant le tableau de racines `racines`. Tester votre fonction à l'aide de la partie `TEST_IFFT` et vérifiez que la transformée inverse de la transormée redonne le polynôme (ses coefficients) de départ.

4.3 Produit par FFT

Description informelle de l'algorithme :
On suppose que les degrés des deux polynômes est une puissance de deux.

1. La taille du tableau de complexes résultat est donc $n = 2\,max(deg(P), deg(Q))$ qu'on appellera .
2. On calcule les racines n de l'unité dans un tableau.
3. On calcule la transformée de Fourier rapide sur chacun des polynômes en entrée.
4. On calcule dans un tableau de complexes les points du polynôme résultat en multipliant simplement les éléments des deux tableaux de complexes obtenus lors de la transormée de Fourier.
5. Finalement, on utilise la transormée de Fourier inverse sur le tableau obtenu afin d'obtenir les coefficients du polynôme résultat.

Question : Implémenter la fonction

```
struct polynome *pn_produit_fft(const struct polynome *p, const struct ↩
    polynome *q);
```

qui retourne un pointeur vers un polynome nouvellement alloué contenant le produit des polynômes d'adresses `p` et `q` en utilisant la FFT.