Produit de polynômes

24 février 2020

L'objectif de ce TP est d'implanter 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.

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", */
    size_t deg; /* plus grand indice d'un coeff non nul,
                0 si polynome nul */
    size_t taille; /* doit être une puissance de 2 > deg */
};
```

On rappelle que `size_t` est un alias de type pour un type entier *non signé*, suffisamment grand pour contenir les tailles en octet des objets (peut être par exemple `long unsigned`, mais cela dépend de la machine et du compilateur).

Pour des raisons techniques, le degré du polynome nul est 0, et non $-\infty$, comme on en a l'habitude.

On vous demande maintenant d'aller lire attentivement le code fourni, et de répondre aux questions suivantes :

1. Pourquoi le paramètre de `pn_agrandir` s'appelle `min_taille` ? À quoi sert la boucle `while` dans cette fonction ? Pourquoi la fonction `pn_agrandir` est-elle utilisé dans `pn_zero` ?
2. Quelle est la complexité de la fonction `pn_est_nul` ? Si la structure `struct` ↩ `polynome` n'avait pas de champ `deg`, comment écrirait-on cette fonction ? Quelle serait sa complexité ?
3. Dans la fonction `pn_agrandir`, pourquoi est-il correct d'initialiser `sz` à `p->taille` dans la première boucle ? À quoi sert la deuxième boucle `for` ?
4. La fonction `pn_set_coeff` sert à modifier un coefficient du polynome. Pourquoi ne se contente-t-on pas de l'instruction `p->coeffs[i] = coeff;` ? Quel est le rôle de chaque bloc `if` dans cette fonction ? Que dire de sa complexité ? Quels cas peuvent être coûteux ?

2 Échauffement

Implanter les fonctions

```
complexe pn_eval(const struct polynome *p, complexe z);
complexe pn_horner(const struct polynome *p, complexe z);
struct polynome *pn_somme(const struct polynome *p, const struct ↩
   polynome *q);
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.gc` : par exemple, pour tester la fonction `pn_eval`, il suffit, avant de compiler, de changer

```
#define TEST_EVAL 0
```

en

```
#define TEST_EVAL 1
```

Pour la fonction `pn_somme`, on pourra, si on en a le temps, faire attention aux points suivants :

1. La complexité doit être linéaire dans le pire des cas.
2. On ne doit parcourir qu'une fois chaque tableau.
3. Le degré du polynôme retourné doit être correct.
4. La taille du polynôme retourné doit être la plus petite possible (tout en étant une puissance de 2).

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 des polynômes de degré inférieur ou égal à $2^{n-1} - 1$, et de même pour Q. Le produit PQ peut alors être écrit de la façon (très astucieuse !) suivante :

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

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 :

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

On peut alors récursivement calculer ces trois produits.

1. Une fonction `pn_karatsuba` est déjà écrite pour vous (ne vous réjouissez pas trop vite, c'est `aux_karatsuba` qui fait tout le travail). Quel est son rôle principal ?
2. La fonction `pn_karatsuba` utilise une fonction

```
 static struct polynome * aux_karatsuba(const struct polynome *↩
p, const struct polynome *q)
```

qui prend deux polynômes dont taille est garanti d'être la même puissance de 2. Implementez la fonction `aux_karatsuba`. Vous pouvez introduire des fonctions supplementaires à vos bésoin, par exemple, une fonction qui calcule la somme des parties gauche et droite du tableau et la tourne en polynôme.

4 Produit via la FFT

On considere deux polynomials de degré $n-1$:

$$A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}, \quad B(x) = b_0 + b_1 x + \ldots + b_{n-1}x^{n-1}.$$

Si nous évaluons ce polynôme en des points distincts $x_1, \ldots, x_n$, alors en utilisant les valeurs $A(x_1), \ldots, A(x_n)$, il est possible de récupérer les coefficients $a_0, \ldots, a_{n-1}$. Une stratégie naïve d'évaluation des $A(x_1), \ldots, A(x_n)$ donne une complexité $O(n^2)$, nous devrons donc rechercher de nouvelles accélérations.

Après avoir calculé les valeurs de deux polynômes, les valeurs de leur produit peuvent être calculées en temps linéaire. En utilisant les valeurs des polynômes, nous devons récupérer leurs coefficients. Encore une fois, ne pensons pas encore à la complexité de cette opération, car nous allons exploiter la structure spécifique des points $x_1, \ldots, x_n$.

Nous considérons l'équation $x^n = 1$ en nombre complexe et prenons $x_1 = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n} = e^{2\pi i/n}$. Ensuite, $x_k := x_1^k = \cos\frac{2\pi k}{n} + i\sin\frac{2\pi k}{n}$. Une telle structure nous permet de remarquer deux propriétés importantes qui permettent enfin nous pour construire un algorithme plus rapide :

— La transformation du vecteur $[a_0, a_1, \ldots, a_n]$ dans un vecteur $[A(x_1), \ldots, A(x_n)]$ peut être écrit sous la forme

$$A(x_k) = \sum_{j=0}^{n-1} a_j e^{2i\pi kj/n}$$

qui est une version discrète de la transformée de Fourier continue

$$\widehat{f}(t) = \int_{\mathbb{R}} f(y) e^{2i\pi yt} dy$$

ayant une propriété importante que la transformation de Fourier inverse est presque la même chose que la transformation directe jusqu'à un changement de signe

$$f(y) = \int_{\mathbb{R}} \widehat{f}(t) e^{-2i\pi yt} dt.$$

— Le fait que x_k est une puissance de x_1 permet d'utiliser le principe de diviser-pour-régner pour accélérer le calcul de la transformation de Fourier.

4.1 Transformation de Fourier naif

Implementez la transformation de Fourier

```
struct polynome * fft(const struct polynome * p)
```

à façon naif : calculez la liste des arguments $x_1, \ldots, x_n$ et evaluez $A(x_i)$ avec `pn_horner`. Testez votre code dans `test-pn.c`.

4.2 Transformation de Fourier inverse

Une transformation de Fourier inverse peut être implémentée comme une transformation directe avec le deux changements $x_1 = \cos\frac{2\pi}{n} - i\sin\frac{2\pi}{n} = e^{-2\pi i/n}$ et que chaque élément de resultat finale doit être divisé par n. Supposons que la fonction directe

```
struct polynome * fft(const struct polynome * p)
```

est implémenté. Implémentez la fonction inverse

```
struct polynome * ifft(const struct polynome * p)
```

Il faut renverser le liste de coefficients obtenu dans `fft` et diviser par n qui est la puissance de 2 correspondante.

4.3 Strategie diviser pour régner

Divisons le polynôme $A(x)$

$$A(x) = a_0 + a_1 x + \ldots + a_{n-1}x^{n-1}$$

dans deux polynômes plus petits :

$$A_0(x) = a_0 x^0 + a_2 x^2 + \ldots + a_{n-2}x^{n/2-1};$$

$$A_1(x) = a_1 x^0 + a_3 x^2 + \ldots + a_{n-1}x^{n/2-1};$$

Le polynôme initial $A(x)$ alors peut être calculé comme

$$A(x) = A_0(x^2) + xA_1(x^2). \tag{1}$$

On peut utiliser cette identité afin d'accelerer la computation de transformation Fourier. On divise le tableau initial des coefficients en deux parties, avec les positions paires et impaires. Après, on calcule la transformation FFT recursivement sur deux parties. Enfin, on récupère la transformation FFT de tableau du taille n avec l'identité (1).

Implementez la fonction `fft` :

```
struct polynome *fft(const struct polynome *p);
```

4.4 Produit rapide avec FFT et inverse FFT

Afin de calculer la produit de deux polynomes A et B, on fait :

1. Agrandir la taille de polynomes A et B pour allouer suffisement espace pour contenir peur produit. Il suffit d'allouer une puissance de 2 superieur ou egale à $2\max(\deg(A), \deg(B))$. Le deux polynômes doivent avoir la même taille qui est une puissance de 2.
2. Calculer les transformation Fourier

$$\widehat{A} = [A(x), A(x^2), \ldots, A(x^N)], \quad \widehat{B} = [B(x), B(x^2), \ldots, B(x^N)]$$

3. Calculer la produit de deux tableaux

$$\widehat{C} = [\widehat{A}(x)\widehat{B}(x), \widehat{A}(x^2)\widehat{B}(x^2), \ldots, \widehat{A}(x^N)\widehat{B}(x^N)]$$

4. Calculer la transformation FFT inverse de $\widehat{C}$.

Implementez la function

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