TP ALBA algorithme sur les polynômes

20 Octobre 2020

La deadline pour le dépot du TP est le dimanche 1er Novembre 23h59. On accordera beaucoup d'importance à respecter les signatures des fonctions. C'est à dire ce que la fonction prend en argument et ce qu'elle doit retourner. En particulier, si une fonction doit retourner un polynôme, retourner la liste des coefficients n'est pas correct. Il faut renvoyer un objet de type polynôme et non pas une liste.

Il est attendu un unique fichier `tp3_nom_prenom.sage` qui doit pouvoir être loadé en terminal pour pouvoir être tester. Le fichier ne doit comporter rien de plus que les fonctions qui sont demandées dans le tp et les fonctions auxiliaires. En particulier, aucune variable ne doit être instanciée en dehors des fonctions et le fichier ne doit éxécuter aucun code. Je vous invite à commenter votre code pour aider à sa clarté.

Quelques aides pour le tp :

— Si vous avez une liste `l`. Vous pouvez récupérer les éléments entre les indices a et $b-1$ avec la commande `l[a:b]`, voir *slice* pour plus de précisions.

— Pour générer des listes, utilisez les générateurs. Par exemple le code `[2*k+1 for k in range(100)]` renvoie la liste des entiers impairs entre 1 et 200. Vous pouvez même mettre des conditions dans le générateur par exemple `[ t for t in range(100) if (t**2 + 6) % 11 == 0]` renvoie la liste des entiers t compris entre 0 et 99 tel que $t^2 + 6$ est divisible par 11.

— Sur sage, `ZZ` est l'anneau des entiers relatifs et `QQ` le corps des nombres rationnels. Pour manipuler des polynômes dans le terminal il faut d'abord instancier une indéterminée et donner un nom à notre anneau de polynôme. Par exemple si on veut appeler `A` l'anneau des polynômes à coefficients entiers en une indéterminée `x`, on fait cela comme ceci

```
A = PolynomialRing(ZZ, 'x')
```

— On peut évidemment faire la même chôse avec plusieurs variables. Supposons que `B` soit un anneau, on définit l'anneau `C` des polynômes à coefficients dans `B` à 3 variables par

```
C = PolynomialRing(B, ['a', 'b', 'c'])
```

Évidemment le nom des variables importe peu, j'aurais pu mettre `x,y,z` à la place de `a,b,c`. Le seul point important en revanche est que une fois que vous avez défini une indéterminée vous ne pouvez plus utiliser ce nom pour autre chose. Par exemple après avoir instancier `C`, je ne peux plus utiliser le nom `a` pour nommer une variable car maintenant `a` est une indéterminée. Vous pouvez taper la commande `type(a)` pour vous en convaincre.

— Si `A` est un `PolynomialRing` à une variable alors vous pouvez instancier un polynôme en donnant une liste de coefficients. Si `A = PolynomialRing(ZZ, 'x')` est instancié comme ceci, alors `A([1,2,3])` renvoie le polynôme $1 + 2x + 3x^2$.

— Pour instancier un corps fini on utilise la commande `GF()` (pour *ground field*), elle prend comme argument le cardinal du corps que vous voulez et vous pouvez donner en plus le nom de l'élément primitif qui engendre le corps. Pour rappel si $q = p^l$ avec p un nombre premier, alors il existe un unique corps de cardinal q à isomorphisme près et $\mathbf{F}_q$ est de la forme $\mathbf{F}_q = \mathbf{F}_p[\alpha]$ avec α algébrique sur $\mathbf{F}_p$. On dit que α est un élément primitif.

Exercice 1 :
Implémenter l'algorithme de Karatsuba. Plus précisément, vous devez écrire une fonction `mult_karatsuba` qui prend en entrée deux polynômes et ressort le produit entre les deux polynômes en utilisant l'algorithme de Karatsuba. Vous pouvez si vous le souhaitez faire appel à des fonctions auxiliaires.

Exercice 2 :
On cherche à implémenter l'algorithme de la FFT. On testera nos fonctions sur les polynômes

$$F = x^4 + 3x^3 + 2x^2 + 6x + 7, G = x^2 + 2x + 6$$

sur le corps $\mathbf{F}_{17}$ (Pourquoi 17 fonctionne ici ?). Pour trouver une racine primitive de l'unité on pourra commencer par utiliser la fonction `multiplicative_generator`.

1. Écrire une fonction `dft` qui prend 2 paramètres : un polynôme f, une racine de l'unité ω, et qui renvoie la transformée de Fourier discrète $(f(1), f(\omega), f(\omega^2), \cdots, f(\omega^{n-1})$.
2. Écrire une fonction `fft1` qui prend 3 paramètres : deux polynômes f, g définis sur un même corps et ω une racine primitive n-ième de l'unité et qui renvoie le produit $fg \mod X^n - 1$ avec n l'ordre de ω en utilisant l'algorithme de FFT.
3. Écrire une fonction `fft` qui prend en entrée deux polynômes de degré quelconque à coefficients dans un corps fini et renvoie le produit de ces deux polynômes en utilisant l'algorithme de FFT. On pourra utiliser la fonction `parent()`. Pour simplifier, si le degré d du résultat est trop grand pour que le corps admette une racine primitive de l'unité d'ordre $2^\alpha > d$, on affichera un message d'erreur et on retournera `None`.
4. Comparer la multiplication naïve, Karatsuba et la FFT sur $\mathbf{F}_{29 \cdot 2^{57}+1}$ (pourquoi ce cardinal ?) pour des degrés croissants. On pourra utiliser la fonction `random_element()`. Écrire une fonction `comparaison()` qui renvoie une liste de quadruplés $(d, t_{\texttt{naif}}, t_{\texttt{Karatsuba}}, t_{\texttt{FFT}})$ constitué du degré d du polynôme qui a été testé et des trois temps des différents algorithmes et affiche la courbe des 3 temps. (Conseil : Une fois qu'un algorithme est trop long pour l'ordre de grandeur de degré considéré pour le test, vous pouvez mettre `None` pour le temps de cet algorithme et ne pas faire le test).