TP 3 : Algorithmes de multiplication rapide de polynômes.

Du Python ! Ici, on code en Python, qui est quand même plus pratique que Caml dans ce contexte. Il n'y a rien à préciser pour déclarer une fonction récursive en Python.

Le sujet. Ce sujet est inspiré d'un TP d'oral des ENS, voie informatique. Comme il est long, je vous ai fait la mise en place de la première partie, et j'ai détaillé les réponses aux questions mathématiques tout au long du texte (la partie sur la FFT n'est pas tout à fait la même que celle du sujet initial). Vous trouverez sur mon site web un script fournissant :

- un entier p premier, fixé à 12289 dans tout le sujet. Ce nombre n'est pas choisi au hasard : $p-1$ est divisible par une grande puissance de 2, plus précisément $p = 3 \times 2^{12} + 1$;
- des listes `U`, `V` et `W` d'entiers. Les éléments de `W` sont des entiers de $[\![0, p-1]\!]$ avec $p = 12289$. Comme p est premier les éléments de `W` peuvent être vus comme des éléments du corps $\mathbb{Z}/p\mathbb{Z}$, ce qu'on fera dans tout le sujet.
- une fonction `P(n,k)` fournissant pour $n > 0$ et $k \geq 0$ un polynôme $P_{n,k}$ construit à partir des éléments de `W`. Un polynôme de $\mathbb{Z}/p\mathbb{Z}[X]$ est représenté par la liste de ses coefficients (identifiés aux entiers de $[\![0, p-1]\!]$) par degré croissant. Le polynôme $P_{n,k}$ ayant degré $n-1$, la liste fournie par `P(n,k)` est de taille n ;
- une fonction de hachage `Hash(alpla, r, A)` associée à la fonction mathématique $\mathsf{Hash}_{\alpha,r}$. Cette fonction prend en paramètre un n-uplet de $\mathbb{Z}/p\mathbb{Z}$ (pouvant être vu comme un polynôme de $\mathbb{Z}/p\mathbb{Z}[X]$ de degré $n-1$) et retourne un élément de $\mathbb{Z}/r\mathbb{Z}$.
- La signature $\mathsf{Sig}(A)$ d'un n-uplet A de $\mathbb{Z}/p\mathbb{Z}$ est le triplet $(\mathsf{Hash}_{17,983}(A), \mathsf{Hash}_{23,991}(A), \mathsf{Hash}_{51,997}(A))$. Il est calculé en Python par la fonction `Sig(A)`. Vous pourrez vérifier que vos fonctions sont correctes via la signature des polynômes calculés.

Remarque : une fois la liste `W` construite, vous pouvez commenter la section de code qui la concerne, pour ne pas la recalculer à chaque fois.

1 Opérations de base et produit naïf

Question 1. Écrire des fonctions :

- `addition(P,Q)` d'addition de deux polynômes de $\mathbb{Z}/p\mathbb{Z}$ (une borne sur le degré du polynôme obtenu est le maximum des degrés des polynômes P et Q. Ça n'a pas d'importance s'il y a des zéros supplémentaires à la fin, la fonction de hachage `Hash` n'y est pas sensible).
- `soustraction(P,Q)` de soustraction de deux polynômes de $\mathbb{Z}/p\mathbb{Z}$.
- `mult_scal(P,a)` de multiplication d'un polynôme `P` par un scalaire `a` de $\mathbb{Z}/p\mathbb{Z}$.
- `mult_mon(P,i)` de multiplication du polynôme `P` par le monôme X^i.

Toutes ces fonctions devront renvoyer de nouveaux polynômes (elles ne modifient pas les listes Python passées en argument). Faites attention à avoir vos coefficients dans $[\![0, p-1]\!]$ (le modulo `%` est là pour ça) : la signature y est sensible.

```
>>> Sig(addition(P(2**5,1),P(2**6,2)))
(812, 701, 219)
>>> Sig(soustraction(P(2**5,1),P(2**6,2)))
(657, 656, 505)
>>> Sig(mult_scal(P(2**5,1),200))
(577, 388, 719)
>>> Sig(mult_mon(P(2**5,1),200))
(205, 40, 509)
```

Question 2. En déduire une fonction `produit_naif(P,Q)` effectuant le produit de deux polynômes de $\mathbb{Z}/p\mathbb{Z}$ de manière naïve.

```
>>> Sig(prod_naif(P(10,1),P(20,2)))
(43, 500, 89)
>>> Sig(prod_naif(P(200,1),P(100,4)))
(459, 572, 349)
>>> Sig(prod_naif(P(4000,1),P(5000,2)))
(897, 396, 706)
```

2 Implémentation de l'algorithme de Karatsuba pour la multiplication de polynômes

On rappelle le principe de la mutliplication de polynômes par l'algorithme de Karatsuba. Pour simplifier, on suppose que les polynômes à multiplier ont même taille $n = 2^\ell$, qui est une puissance de 2, avec $\ell \geq 1$. Notons $A = \sum_{i=0}^{n-1} a_i X^i$ et $B = \sum_{i=0}^{n-1} b_i X^i$ deux polynômes de $\mathbb{Z}/p\mathbb{Z}[X]$ de degré $n-1$. On décompose les deux polynômes en deux morceaux :

$$A = A_0 + X^{n/2} A_1 \text{ avec } A_0 = \sum_{i=0}^{n/2-1} a_i X^i \quad \text{et} \quad A_1 = \sum_{i=0}^{n/2-1} a_{n/2+i} X^i$$
$$B = B_0 + X^{n/2} B_1 \text{ avec } B_0 = \sum_{i=0}^{n/2-1} b_i X^i \quad \text{et} \quad B_1 = \sum_{i=0}^{n/2-1} b_{n/2+i} X^i$$

Le produit AB se calcule alors récursivement avec 3 produits de polynômes de taille $n/2$:

$$AB = A_0B_0 + X^{n/2}(A_1B_0 + A_0B_1) + X^n A_1B_1 = T_0 + X^{n/2}(T_2 - T_1 - T_0) + X^n T_1 \text{ avec } \begin{cases} T_0 = A_0B_0 \\ T_1 = A_1B_1 \\ T_2 = (A_0 + A_1)(B_0 + B_1) \end{cases}$$

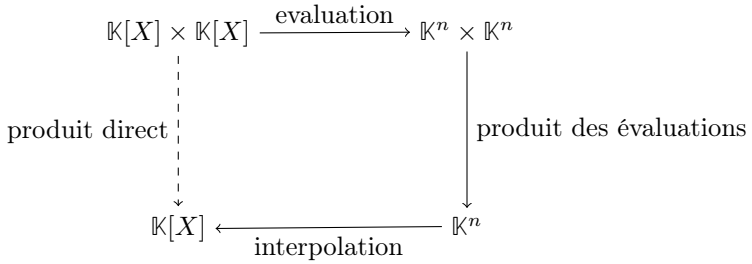
Question 3. Coder une fonction récursive `Karatsuba(A,B)` effectuant le produit de deux polynômes de $\mathbb{Z}/p\mathbb{Z}[X]$, de même taille supposée être une puissance de 2. Le cas de base est évidemment lorsque les deux polynômes ont une taille 1 (ce sont des constantes).

```
>>> Sig(Karatsuba(P(2**6,1),P(2**6,2)))
(512, 319, 960)
>>> Sig(Karatsuba(P(2**8,1),P(2**8,2)))
(148, 583, 153)
>>> Sig(Karatsuba(P(2**13,1),P(2**13,2)))
(833, 610, 142)
```

Question 4. Quelle est la complexité de cette approche ? (On pourra supposer que les deux polynômes à multiplier ont même taille, qui est une puissance de 2). Il faut savoir faire la démonstration !

3 Implémentation de l'algorithme de FFT

3.1 Explication de l'algorithme



Soit $\mathbb{K}$ un corps (dans la suite, on aura $\mathbb{K} = \mathbb{Z}/p\mathbb{Z}$). On se donne deux polynômes A et B de $\mathbb{K}[X]$, tels que AB est de degré au plus $n-1$. L'algorithme de FFT (*fast Fourier transform*) pour la multiplication repose sur le principe suivant : si $\xi_0 \ldots, \xi_{n-1}$ sont n scalaires distincts de $\mathbb{K}$, l'application d'évaluation :

$$\begin{array}{rccc} \psi : & \mathbb{K}_{n-1}[X] & \longrightarrow & \mathbb{K}^n \\ & P & \longmapsto & (P(\xi_0), \ldots, P(\xi_{n-1})) \end{array}$$

est un isomorphisme de $\mathbb{K}$ espaces-vectoriels. Pour calculer le produit des deux polynômes A et B de $K[X]$, on applique donc la stratégie suivante :

— calcul des évaluations $\psi(A)$ et $\psi(B)$ en les ξ_i ;

— produit terme à terme des évaluations : on obtient donc $\psi(A)\cdot\psi(B)=\psi(AB)$;

— interpolation, on calcule donc $AB=\psi^{-1}(\psi(AB))$.

Ceci est résumé dans le schéma ci-dessus. On va voir qu'en choisissant convenablement les ξ_i, l'évaluation et l'interpolation se calculent rapidement, la stratégie précédente donnant un algorithme de meilleure complexité que l'algorithme naïf ou même l'algorithme de Karatsuba.

3.2 Racines primitives de l'unité

On dit qu'un élément ω de $\mathbb{K}$ est une racine primitive n-ème de l'unité si $\omega^n=1$, mais $\omega^t\neq 1$ pour tout $t\in[\![1,n-1]\!]$. Supposons que $\mathbb{K}$ possède une racine primitive n-ème de l'unité ω, et considérons l'application d'évaluation ψ où les ξ_i sont précisément les ω^i. La matrice de ψ dans les bases canoniques de $\mathbb{K}_{n-1}[X]$ (au départ) et $\mathbb{K}^n$ (à l'arrivée) est la matrice suivante :

$$V_\omega=\begin{pmatrix}1&1&1&\cdots&1\\1&\omega&\omega^2&\cdots&\omega^{n-1}\\1&\omega^2&\omega^4&\cdots&\omega^{2(n-1)}\\\vdots&\vdots&\ddots&\ddots&\vdots\\1&\omega^{n-1}&\cdots&\cdots&\omega^{(n-1)(n-1)}\end{pmatrix}$$

Autrement dit, $V_\omega=(\omega^{ij})_{0\leq i,j\leq n-1}$.

Question 5. Vérifier que $(V_\omega)^{-1}=(1/n)\times V_{\omega^{-1}}$.

La question précédente montre que si l'on sait effectuer rapidement l'évaluation en les $\xi_i=\omega^i$, il est facile de réaliser l'interpolation : il suffit de faire une évaluation en les $\xi_i=\omega^{-i}$ (on identifie à nouveau un vecteur de $\mathbb{K}^n$ et un polynôme de $\mathbb{K}_{n-1}[X]$), et de multiplier ensuite le résultat par l'inverse de n dans $\mathbb{K}$. Il reste à voir comment effectuer rapidement l'évaluation : on va pour cela se restreindre au cas $\mathbb{K}=\mathbb{Z}/p\mathbb{Z}$, et voir pourquoi $p=12289$ est intéressant.

3.3 Structure de $(\mathbb{Z}/p\mathbb{Z})^*$. Racines de l'unité. Calcul.

C'est un résultat classique que si p est premier, $\mathbb{Z}/p\mathbb{Z}$ est un corps. L'ensemble des éléments non nuls de $\mathbb{Z}/p\mathbb{Z}$ a donc une structure de groupe (multiplicatif), noté $((\mathbb{Z}/p\mathbb{Z})^*,\times)$. La structure de ce groupe est loin d'être quelconque : c'est un groupe cyclique.

Théorème 1. *Soit $(\mathbb{K},+,\times)$ un corps fini, alors le groupe multiplicatif $(\mathbb{K}^*,\times)$ est un groupe cyclique.* [1]

Ainsi, dans le cas d'un corps $\mathbb{Z}/p\mathbb{Z}$, on a l'isomorphisme $((\mathbb{Z}/p\mathbb{Z})^*,\times)\simeq(\mathbb{Z}/(p-1)\mathbb{Z},+)$. Il s'ensuit que le nombre de générateurs de $((\mathbb{Z}/p\mathbb{Z})^*,\times)$ est $\varphi(p-1)$, où φ est la fonction caractéristique d'Euler.

Revenons à notre cas particulier $p=12289=3\times 2^{12}+1$. On a $p-1=3\times 2^{12}$ donc $\varphi(p-1)=2\times 2^{11}=2^{12}$. Il s'ensuit qu'un tiers des éléments de $(\mathbb{Z}/p\mathbb{Z})^*$ sont des générateurs de ce groupe, d'ordre 3×2^{12}. Si x est un tel générateur, x^3 est d'ordre 2^{12}. Ainsi, dans $\mathbb{Z}/p\mathbb{Z}$, il y a des racines primitives 2^{12}-ème de l'unité ! 2014 en est une, par exemple.

Question 6. Si ω n'est pas une racine primitive de l'unité dans $\mathbb{Z}/p\mathbb{Z}$, c'est que son ordre dans le groupe multiplicatif est un diviseur strict de 3×2^{12}. En utilisant ce principe, écrire une fonction `est_prim(w)` testant si $\omega\in[\![1,p-1]\!]$ est primitive.

```
>>> est_prim(2016)
False
>>> est_prim(2017)
True
```

Remarque : évitez de faire ça bourrinement, on peut se contenter de calculer $\omega^{3\times 2^{11}}$ et $\omega^{2^{12}}$, le tout modulo p. On calculera à l'aide d'une boucle, en réduisant à chaque étape modulo p : il est très maladroit de calculer un très grand entier pour le réduire modulo p tout à la fin.

Question 7. Calculer la plus petite racine primitive de l'unité modulo p, et en déduire une racine primitive 2^{12}-ème dans $\mathbb{Z}/p\mathbb{Z}$.

1. Si vous demandez gentiment, je vous fais une démonstration.

Question 8. Écrire un algorithme `bezout(a,b)` prenant en entrée deux entiers, qu'on suppose positifs et avec $a \neq 0$, et renvoyant le couple de Bézout (u, v) associé : on doit avoir $au + bv = \text{PGCD}(a, b)$. (Remarque : ça marche très bien en récursif, en suivant l'algorithme d'Euclide).

```
>>> bezout(36,89)
(-42, 17)
>>> -42*36+17*89
1
```

Question 9. En déduire une fonction `invmod(w)` prenant en entrée un élément de $[\![1, p-1]\!]$ et renvoyant son inverse modulo p (l'unique élément ω' de $[\![1, p-1]\!]$ vérifiant $\omega\omega' \equiv 1[p]$).

```
>>> invmod(2014)
4436
>>> (2014*4436)%p
1
```

3.4 Interpolation récursive

On se donne $n = 2^\ell$ une puissance de 2, avec $\ell \geq 1$, et ω une racine primitive 2^ℓ-ième de l'unité, et on cherche à calculer rapidement les $(P(\omega^i))_{0\leq i<n}$ pour $P = \sum_{i=0}^{n-1} p_i X^i$ un polynôme de $\mathbb{Z}/p\mathbb{Z}$. Dans la suite, ce vecteur de $(\mathbb{Z}/p\mathbb{Z})^n$ sera noté $\mathsf{TFD}_\omega(P)$.

Considérons R_0 et R_1 les restes dans les divisions euclidiennes de P par $X^{n/2} - 1$ et $X^{n/2} + 1$. On a alors $R_0 = \sum_{k=0}^{n/2-1}(p_k + p_{n/2+k})X^k$ et $R_1 = \sum_{k=0}^{n/2-1}(p_k - p_{n/2+k})X^k$. Puisque $X^{n/2} - 1$ annule toutes les racines de l'unité de la forme ω^{2i} et $X^{n/2} + 1$ annule toutes celles de la forme ω^{2i+1}, on a d'une part :

$$P(\omega^{2i}) = R_0(\omega^{2i}) \quad \text{pour tout } i \in [\![0, n/2-1]\!]$$

et d'autre part, en posant $\overline{R_1}(X) = R_1(\omega X)$:

$$P(\omega^{2i+1}) = R_1(\omega^{2i+1}) = R_1(\omega \times \omega^{2i}) = \overline{R_1}(\omega^{2i}) \quad \text{pour tout } i \in [\![0, n/2-1]\!]$$

Clairement, ω^2 est une puissance primitive $n/2$-ème de l'unité. On peut alors considérer la transformée de Fourier discrète en les puissances de ω^2 (qui fournit un élément de $(\mathbb{Z}/p\mathbb{Z})^{n/2}$). Alors $\mathsf{TFD}_\omega(P)$ s'obtient en intercalant les éléments de $\mathsf{TFD}_{\omega^2}(R_0)$ et $\mathsf{TFD}_{\omega^2}(\overline{R_1})$. On obtient donc un algorithme récursif, le cas de base étant TFD_1, qui est simplement l'identité (en identifiant les polynômes de $\mathbb{K}_0[X]$ et les éléments de $\mathbb{K}$).

3.5 Implémentation de l'évaluation

La liste `Omega` fournie dans le script Python donne des racines primitives 2^i-èmes, pour tout $i \in [\![0, 12]\!]$ (elles ont été faciles à calculer : ce sont les carrés successifs de 2014 modulo p, à l'envers). Elles est utile pour les tests.

Question 10. Écrire deux fonctions `calculR0(P)` et `calculR1b(P,w)` prenant en entrée un polynôme d'une certaine taille n (supposé être une puissance de 2) et une racine n-ème de l'unité (pour `calculR1b`) et renvoyant les deux polynômes R_0 et $\overline{R_1}$ de tailles $n/2$ décrits précédemment (attention à bien prendre les coefficients dans $[\![0, p-1]\!]$).

```
>>> Sig(calculR0(P(2**6,1)))
(720, 64, 549)
>>> Sig(calculR1b(P(2**12,1), 2014))
(761, 709, 739)
```

Question 11. Écrire une fonction `imbrique(L1,L2)` prenant en entrée deux listes Python supposées de même taille, et renvoyant une liste alternant les éléments des deux listes, en commençant par `L1`.

```
>>> imbrique([0, 1, 2], [3, 4, 5])
[0, 3, 1, 4, 2, 5]
```

Question 12. Déduire des questions précédentes une fonction `TFD(P,w)` prenant en entrée un polynôme P d'une certaine taille n, une racine primitive n-ème de l'unité dans $\mathbb{Z}/p\mathbb{Z}$, et renvoyant la liste des coefficients de $\mathsf{TFD}_\omega(P)$.

```
>>> Sig(TFD(P(2**4,0),Omega[4]))
(932, 648, 526)
>>> Sig(TFD(P(2**8,1),Omega[8]))
(728, 563, 327)
>>> Sig(TFD(P(2**12,2),Omega[12]))
(68, 2, 705)
```

3.6 On recolle les morceaux

On rappelle que l'interpolation se résume au calcul de $\mathsf{TFD}_{\omega^{-1}}$ et multiplication par l'inverse de n modulo p (avec n l'ordre de ω).

Question 13. Écrire une fonction `prod_scal(L1,L2)` prenant en entrée deux listes d'éléments de $\mathbb{Z}/p\mathbb{Z}$ de même taille et effectuant le produit terme à terme (attention à bien prendre les coefficients dans $[\![0, p-1]\!]$)

```
>>> Sig(prod_scal(P(2**4,0),P(2**4,1)))
(868, 676, 620)
```

Question 14. Déduire des questions précédentes une fonction `prod_FFT(A,B,w,n)` prenant en entrée deux polynômes dont le produit a une taille au plus n, l'entier n et une racine primitive n-ème de l'unité dans $\mathbb{Z}/p\mathbb{Z}$, et retournant les coefficients du produit AB dans $\mathbb{Z}/p\mathbb{Z}$.

```
>>> Sig(prod_FFT(P(2**8,1),P(2**8,2),2**9,Omega[9]))
(148, 583, 153)
>>> Sig(prod_FFT(P(2**10,1),P(2**10,2),2**11,Omega[11]))
(959, 926, 915)
>>> Sig(prod_FFT(P(2**11,1),P(2**11,2),2**12,Omega[12]))
(537, 70, 292)
```

Question 15. Quelle est la complexité de la multiplication par FFT, dans le cas où on multiplie dans un corps $\mathbb{K}$ deux polynômes de même taille une puissance de 2, en nombre d'opérations dans $\mathbb{K}$?

4 Approche mixte

Avec $p = 12289$, on est limité à des produits de polynômes qui ont une taille au plus 2^{12} avec l'approche précédente. Une idée pour multiplier des polynômes de plus grandes tailles est de combiner algorithmes de Karatsuba et FFT :

— si la taille du produit est inférieure à 2^{12}, on utilise la FFT ;

— sinon, on utilise l'algorithme de Karatsuba.

Question 16. Écrire une fonction `prod_mixte(A,B)` prenant en entrée deux polynômes de même taille une puissance de 2 que l'on peut supposer être supérieure ou égale à 2^{11}, et renvoyant le produit via la stratégie décrite ci-dessus.

```
>>> Sig(prod_mixte(P(2**14,1),P(2**14,2)))
(249, 846, 142)
>>> Sig(prod_mixte(P(2**15,1),P(2**15,2)))
(863, 811, 472)
>>> Sig(prod_mixte(P(2**16,1),P(2**16,2)))
(660, 950, 930)
>>> Sig(prod_mixte(P(2**18,1),P(2**18,2)))
(952, 129, 945)
```

En pratique, la FFT est bien souvent utilisée pour calculer des produits de polynômes de grande taille à coefficients rationnels ou complexes. Sur $\mathbb{C}$, les racines de l'unité sont données par les classiques $e^{\frac{2ik\pi}{n}}$. Sur les rationnels en revanche, on n'a pas de racines n-ièmes ! Pour contourner ce problème, on utilise des techniques du type « restes chinois ». Il suffit d'effectuer les calculs modulo des entiers premiers pour remonter ensuite le produit sur $\mathbb{Q}$. On est donc amené à chercher de bons entiers premiers, c'est à dire ceux pour lesquels $\mathbb{Z}/p\mathbb{Z}$ possède des racines primitives n-ièmes avec n une grande puissance de 2 : c'est par exemple le cas de 12289. Celui-ci est un peu petit : par exemple, avec $p = 4179340454199820289 = 29 \times 2^{57} + 1$, on a des racines 2^{57}-ième de l'unité (23 en est une). On peut donc multiplier deux polynômes de taille 2^{56} modulo p. Si les racines de l'unité viennent toutefois à manquer, l'algorithme de Schönage et Strassen (années 70-80) permet de multiplier deux polynômes sur un anneau quelconque $\mathbb{A}$ avec une complexité $O(n\log(n)\log(\log(n)))$ (en termes d'opérations dans $\mathbb{A}$). L'algorithme consiste à travailler dans un anneau contenant $\mathbb{A}$ dans lequel on a rajouté artificiellement les racines de l'unité qui manque.