TP 5 : FFT pour le produit de polynômes

Commandes utiles sur les tableaux

Dans la suite, `t` est un tableau (type `'a array`). Le nom du module (`Array`) préfixe toutes les fonctions.

- `Array.length t` est la longueur de `t` ;
- Si `t` de taille n, `t.(i)` (resp. `t.(i) <- x`) pour $0 \leq$ `i` $< n$ permet d'accéder (resp. de remplacer par `x`) l'élément à l'indice `i`. `Array.length t` est la longueur de `t` ;
- `Array.make n x` crée un tableau de taille n rempli de x ;
- `Array.append t u` concatène deux tableaux de même type en un nouveau (complexité linéaire en la somme des tailles des tableaux, ne pas confondre avec le `append` de Python !)
- `Array.map f t`, de type `('a -> 'b) -> 'a array -> 'b array`, crée un nouveau tableau obtenu par application de `f` sur tous les éléments de `t`.

1 Opérations dans un corps fini

Le but de ce TP va être de multiplier des polynômes à coefficients entiers de manière rapide, plus encore qu'avec le produit de Karatsuba. Une difficulté est que pour que la méthode fonctionne, le corps sur lequel on travaille doit posséder de bonnes *racines de l'unité*, c'est-à-dire des éléments ω tels que $\omega^n = 1$ pour un certain n. Sur $\mathbb{Z} \subset \mathbb{R}$, il n'y a que $\{\pm 1\}$ ce qui fait peu de racines de l'unité !

Une astuce pour pallier cette difficulté est de travailler sur *un corps fini*, plus particulièrement le corps $\mathbb{F}_p = (\mathbb{Z}/p\mathbb{Z}, +, \times)$ avec p premier, dont les éléments peuvent être vus comme les (classes d')entiers modulo p. Le théorème de Bézout assure que $\mathbb{F}_p$ est bien un corps, car pour tout entier x de $[\![1, p-1]\!]$, il existe un (unique) entier y de $[\![1, p-1]\!]$ tel que $xy \equiv 1[p]$: y est donc l'inverse de x modulo p.

On verra que de « bons entiers premiers » p sont ceux tels que $p-1$ est divisible par une grande puissance de 2, on fixe donc dans tout le TP :

```
let p = 12289 ;;
```

En effet, vous pouvez vérifier que $p = 3 \times 2^{12} + 1$. Le but du TP est donc de multiplier des poynômes dont les coefficients sont des entiers modulo p. La première étape est de réaliser les opérations du corps $\mathbb{F}_p$, à savoir $+$, $-$, et $\times$, à chaque fois modulo p. On va les associer aux opérateurs `++`, `--` et `**` (`**` existe déja sur les flottants, on va écraser sa définition, ce qui n'est pas gênant pour ce TP). Voici comment réaliser l'addition :

```
let (++) x y=(x+y) mod p ;;
```

Question 1. Taper le code précédent, et vérifiez que :

```
#4864 ++ 11592 ;;
- : int = 4167
```

Question 2. Définir de même les deux opérateurs `--` et `**` (utiliser le fait que $(x-y) \bmod p = (x + (p-y)) \bmod p$).

```
#4864 -- 11592 ;;
- : int = 5561
#4864 ** 11592 ;;
- : int = 1556
```

La division dans $\mathbb{F}_p$ se définit facilement : $x/y = x \times y^{-1}$ pour y non nul. Encore faut il pouvoir calculer l'inverse d'un élément non nul de $\mathbb{F}_p$, ce qui se fait grâce à l'algorithme d'Euclide (étendu), permettant de calculer un couple de Bézout.

Question 3. Soit n et m deux entiers (positifs), avec m non nul. On pose q et r le quotient et le reste de la division euclidienne de n par m, de sorte que $n = mq + r$. On suppose que l'on connaît un couple de Bézout associé à m et r (c'est-à-dire u et v tels que $um + vr = \text{PGCD}(m, r)$). Donner un couple de Bézout de n et m.

Question 4. En déduire un algorithme récursif `bezout n m` fournissant un couple de Bézout associé à n et m.

```
#bezout 8 p ;;
- : int * int = -1536, 1
#8*(-1536)+p ;;
- : int = 1
#bezout 11592 p ;;
- : int * int = -3156, 2977
```

Question 5. À partir de la fonction précédente, on peut facilement calculer l'inverse d'un élément de $\mathbb{F}_p$: si $uy + vp = \text{PGCD}(y, p) = 1$, alors l'inverse de y est u. En déduire un opérateur `//` de division dans $\mathbb{F}_p$, que vous implémenterez. Remarque : attention à prendre u modulo p et positif, lui rajouter manuellement p si nécessaire.

```
#5 // 8 ;;
- : int = 4609
#4864 // 11592 ;;
- : int = 10466
#10466**11592 ;;
- : int = 4864
```

2 Opérations sur les polynômes et produit naïf

Comme dans le cours, on représente un polynôme comme le tableau de ses coefficients : plus précisément on associe à $\sum_{k=0}^{n-1} a_k X^k$ le tableau `[|`$a_0, a_1, \ldots, a_{n-1}$`|]`.

Question 6. Écrire une fonction `add_p a b` d'addition de deux polynômes de $\mathbb{F}_p[X]$, sachant que ceux-ci peuvent être de tailles différentes (gérer les différents cas), et même éventuellement vides (un tableau vide représente le polynôme nul). Attention : vous ne devez modifier ni `a` ni `b` mais créer un nouveau tableau.

Question 7. Écrire une fonction `prod_mon a n` effectuant le produit d'un polynôme de $\mathbb{F}_p[X]$ par le monôme X^n (il suffit de rajouter n zéros au début, on utilisera `Array.append`). Attention : vous ne devez pas modifier `a` mais créer un nouveau tableau.

Question 8. Écrire de même un fonction `prod_scal a x` effectuant le produit d'un polynôme de $\mathbb{F}_p[X]$ par un élément de $\mathbb{F}_p$. Attention : vous ne devez pas modifier `a` mais créer un nouveau tableau.

Question 9. Déduire des trois questions précédentes une fonction `prod_p a b` de multiplication de deux polynômes de $\mathbb{F}_p[X]$. On pourra utiliser une référence vers un tableau initialement vide (`[| |]`).

```
#prod_p [|1665 ; 11682 ; 14 |] [|54 ; 6023 |] ;;
- : int array = [|3887; 4560; 6917; 10588|]
```

Cette fonction ne sera utilisée dans la suite qu'à des fins de vérification : le but est de faire mieux en complexité !

3 Principe de la FFT et produit des évaluations

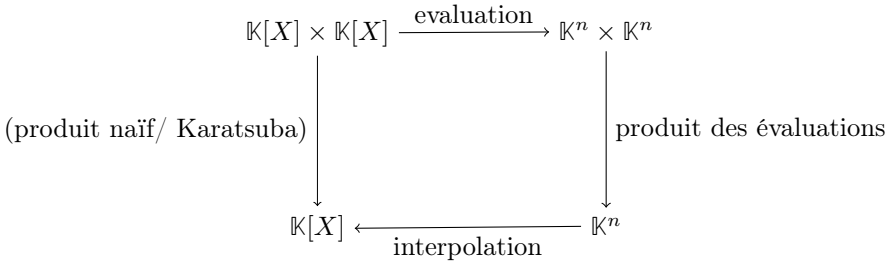
L'idée de la FFT (Fast Fourier Transform) pour le produit dans $\mathbb{K}[X]$ de deux polynômes A et B dont la somme des degrés est strictement inférieure à un entier $n > 0$ est la suivante :

— évaluer les deux polynômes A et B en n points distincts de $\mathbb{K}$: on calcule donc $(A(a_i))_{0 \leq i < n}$ et $(B(a_i))_{0 \leq i < n}$ pour un certain n-uplet $(a_0, \ldots, a_{n-1})$ d'éléments distincts de $\mathbb{K}$;

— On multiplie terme à terme les évaluations, ce qui fournit $(AB(a_i))_{0 \leq i < n}$;

— On reconstruit le polynôme AB grâce à la connsaissance des $AB(a_i)$. C'est en effet possible car l'application de $\mathbb{K}_{n-1}[X]$ dans $\mathbb{K}^n$ qui à P associe $(P(a_i))_{0 \leq i < n}$ est un isomorphime, et AB est supposé de degré strictement inférieur à n.

Voici un résumé :



On va voir qu'en suivant ce schéma, la complexité du calcul du produit AB est inférieure à celle du produit naïf, ou du produit avec la méthode de Karatsuba, si l'on choisit intelligemment les éléments a_i (en prenant des racines de l'unité). Dans la suite, un tableau de taille n à coefficients dans $\mathbb{F}_p$ pourra à la fois dénoter un polynôme de $(\mathbb{F}_p)_{n-1}[X]$ où un n-uplet d'éléments de $\mathbb{F}_p$.

Question 10. Écrire une fonction `prod_terme v w` prenant en entrée deux tableaux de même taille (à coefficients dans $\mathbb{F}_p$) et renvoyant le produit terme à terme. Attention : vous ne devez modifier ni `v` ni `w` mais renvoyer un nouveau tableau.

Comme on le voit facilement, l'étape du produit des évaluations est en $O(n)$. La suite est dévolue au calcul de l'évaluation, pour les a_i des racines de l'unité : on verra que ce choix rend l'interpolation facile.

4 Racines de l'unité et évaluation

4.1 Racines de l'unité dans $\mathbb{F}_p$

On dit que ω est une racine primitive n-ième de l'unité dans $\mathbb{K} = \mathbb{F}_p$ si $\omega^n = 1$ et si $\omega^t \neq 1$ pour tout $t \in \{1, \dots, n-1\}$. On suppose que $\mathbb{K}$ contient une telle racine primitive n-ième ω. On peut alors vérifier les faits suivants :

— ω^{-1} est également une racine primitive n-ième de l'unité.

— si $n = qd$, alors ω^q est une racine primitive d-ième de l'unité.

— si $t \in \{1, \dots, n-1\}$, alors $\sum_{i=0}^{n-1} \omega^{it} = 0$ et $\sum_{i=0}^{n-1} \omega^{-it} = 0$.

Remarque : en prenant $\omega = \mathrm{e}^{\frac{2i\pi}{n}}$, vous êtes capables de vérifier la même chose sur $\mathbb{C}$ facilement. La preuve est essentiellement la même sur $\mathbb{F}_p$!

Dans la suite, on supposera que n est une puissance de 2, et s'écrit 2^j. Voila où intervient l'intérêt de l'entier $p = 3 \times 2^{12} + 1$ choisi : il y a des racines 2^{12}-ème primitive de l'unité dans $\mathbb{F}_p$. Je vous en donne une : 41 (c'est la plus « petite »).

Question 11. À partir de 41, créer un tableau `racines_unite` de taille 13 tel que `racines_unite.(i)` contienne une racine primitive 2^i-ème de l'unité dans $\mathbb{F}_p$, pour tout $i \in [\![0, 12]\!]$.

```
#racines_unite ;;
- : int array = [|1; 12288; 10810; j'ai coupé le milieu ; 41|]
```

4.2 Divisons, pour régner ensuite

On suppose pour les questions 12 et 13 que n est pair et s'écrit $n = 2k$, et on fixe ω une racine primitive n-ième de l'unité dans $\mathbb{K}$. D'après le fait précédent, $\omega^k = -1$ (modulo p). On note φ_ω l'application :

$$\begin{array}{rccc} \varphi_\omega : & \mathbb{K}_{n-1}[X] & \longrightarrow & \mathbb{K}^n \\ & F & \longmapsto & (F(1), F(\omega), \dots, F(\omega^{n-1})) \end{array}$$

Pour $F = \sum_{i=0}^{n-1} f_i X^i$ un polynôme de $\mathbb{K}_{n-1}[X]$, on note (Q_0, R_0) le quotient et le reste dans la division euclidienne de F par $X^k - 1$, et de même (Q_1, R_1) dans la division euclidienne de F par $X^k + 1$.

Question 12. *« Division ».* Cette question est à traiter sur feuille en grande partie.

a. Pour $j \in [\![0, n-1]\!]$, quel est le reste dans la division euclidienne de X^j par $X^k - 1$? (On distinguera les cas $j < k$ et $j \geq k$.)

b. En déduire que $R_0(X) = \sum_{j=0}^{k-1} (f_j + f_{j+k}) X^j$. Donner une expression similaire pour R_1.

c. On pose $\overline{R_1}(X) = R_1(\omega X)$. Donner l'expression de $\overline{R_1}$.

d. Écrire deux fonctions `calculR0 f k` et `calculR1b f k w` prenant en entrée un tableau de coefficients `f` symbolisant le polynôme F, ainsi que l'entier $k = n/2$ et la racine primitive n-ième ω (pour `calculR1b`), et retournant les tableaux de coefficients (de tailles k) associés à R_0 et $\overline{R_1}$. On impose que les complexités de `calculR0 f k` et `calculR1b f k w` soient de $O(n)$ opérations dans $\mathbb{K}$.

Question 13. *« Règne ».* Cette question est à traiter sur feuille en grande partie également. On rappelle que ω^2 est une racine primitive k-ième de l'unité, avec $k = n/2$. On note φ_{ω^2} l'application :

$$\begin{array}{rccc} \varphi_{\omega^2} : & \mathbb{K}_{k-1}[X] & \longrightarrow & \mathbb{K}^k \\ & G & \longmapsto & (G(1), G(\omega^2), \dots, G((\omega^2)^{k-1})) \end{array}$$

a. Pour tout $j \in \{0, \dots, k-1\}$, exprimer $F(\omega^{2j})$ et $F(\omega^{2j+1})$ en fonction de $R_0(\omega^{2j})$ et $\overline{R_1}(\omega^{2j})$

b. Comment obtenir facilement $\varphi_\omega(F)$ à partir de $\varphi_{\omega^2}(R_0)$ et $\varphi_{\omega^2}(\overline{R_1})$?

c. Écrire une fonction `recomposition t0 t1 k` prenant en entrée l'entier k et deux tableaux de taille k, tels que si les tableaux contiennent les coefficients de $\varphi_{\omega^2}(R_0)$ et $\varphi_{\omega^2}(\overline{R_1})$ la fonction retourne le tableau de taille $n = 2k$ associé à $\varphi_\omega(F)$, en complexité $O(n)$.

4.3 King in the north !

On suppose dorénavant que n est une puissance de 2, et on note toujours ω une racine primitive n-ième de l'unité dans $\mathbb{K}$. Le but de la question qui suit est de donner un algorithme récursif pour calculer efficacement $\varphi_\omega(F)$, avec F un polynôme de degré au plus $n-1$ représenté par son tableau de coefficients `f` de taille n. Cet algorithme utilisera les fonctions `recomposition`, `calculR0` et `calculR1b` précédentes.

Question 14. *Un algorithme « Diviser pour Régner » pour la calcul de $\varphi_\omega(F)$.*

a. Si $n = 1$ (et donc $\omega = 1$), la représentation du polynôme F sous forme de tableau de coefficients est réduite à 1 élément. Quelle est la représentation de $\varphi_1(F)$ dans ce cas ?

b. Écrire un algorithme récursif `evaluation f n w` prenant en entrée un tableau de coefficients `f` de taille n représentant le polynôme $F = \sum_{i=0}^{n-1} f_i X^i$ et une racine primitive n-ième de l'unité `w`, et renvoyant le tableau symbolisant $\varphi_\omega(F)$. On impose que la complexité de l'algorithme vérifie :

$$C(n) = 2C(n/2) + O(n) \text{ si } n \text{ est une puissance de 2 supérieure ou égale à 2,}$$

c. Justifier que $C(n) = O(n \log_2(n))$ (on posera $n = 2^p$ et on considérera les $C(2^k)/2^k$ pour $1 \leq k \leq p$).

```
#evaluation [|1; 2; 3; 4|] 4 racines_unite.(2) ;;
- : int array = [|10; 2956; 12287; 9329|]
```

5 Interpolation, et conclusion

On a vu dans les questions précédentes comment calculer efficacement $\varphi_\omega(A)$, $\varphi_\omega(B)$ et comment en déduire $\varphi_\omega(AB)$. Il reste donc à inverser φ_ω pour en déduire le produit AB. C'est là que l'utilisation des racines de l'unité se révèle très utile. En effet, la matrice V_ω de l'application φ_ω, dans les bases canoniques de $\mathbb{K}_{n-1}[X]$ (au départ) et $\mathbb{K}^n$ (à l'arrivée), n'est autre que la matrice de Van-der-Monde associée aux (ω^i) :

$$V_\omega = (\omega^{ij})_{0 \leq i,j \leq n-1} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \omega^{(n-1)(n-2)} \\ 1 & \cdots & \omega^{(n-2)(n-1)} & \omega^{(n-1)(n-1)} \end{pmatrix}$$

On vérifie facilement grâce aux faits admis sur les racines de l'unité que $V_\omega V_{\omega^{-1}} = nI_n$, ainsi $(V_\omega)^{-1} = \frac{1}{n} V_{\omega^{-1}}$.

Pour interpoler, il suffit donc réaliser une nouvelle évaluation, avec ω^{-1} à la place de ω, et diviser ensuite tous les coefficients du tableau obtenu par n.

Question 15. Écrire une fo nction `interpolation t n w` telle que, si `t` est le tableau de taille n symbolisant $\varphi_\omega(F)$, alors `interpolation t n w` retourne le tableau de coefficients de F. Pour calculer ω^{-1}, on utilisera simplement `1 // w`, de même que pour diviser les coefficients par n.

Question 16. *Conclusion.* Écrire une fonction `produitFFT a b n w`, telle que si n est un entier qui est une puissance de 2, A et B sont deux polynômes de degré strictement inférieurs à $n/2$ représentés par leur tableaux de coefficients `a` et `b` de tailles n, et `w` une racine primitive n-ième de l'unité, `produitFFT a b n w` retourne le tableau de taille n représentant le produit $A \times B$. Donnez la complexité de `produitFFT` en fonction de n.

```
#produitFFT [|100; 200; 300; 400; 0; 0; 0; 0|] [|500; 600; 700; 800; 0; 0; 0; 0|] 8 racines_unite.(3) ;;
- : int array = [|844; 243; 8197; 10128; 7839; 3862; 486; 0|]
#prod_p [|100; 200; 300; 400|] [|500; 600; 700; 800|]  ;;
- : int array = [|844; 243; 8197; 10128; 7839; 3862; 486|]
```

Question 17. *Des tests.* Bon, multiplier des polynômes de taille 4, c'est bien gentil.

1. Avec l'entier p choisi, vous pouvez multiplier deux polynômes dont la somme des degrés fait strictement moins que $2^{12} = 4096$ (représentés comme des tableaux de taille 2^{12}, avec des zéros vers la fin, comme ci-dessus), avec $\omega = 41$ qui est une racine 2^{12}-ème de l'unité. Comparer « à vue » le temps d'exécution par rapport à l'algorithme naïf. Utiliser `Random.int` pour générer vos polynômes : cette fonction de type `int -> int` prend en entrée un entier $k > 0$ et renvoie un entier aléatoire de $[\![0, k-1]\!]$.
2. Si vous voulez plus gros, prenez $p = 167772161 = 5 \times 2^{25} + 1$. L'entier $\omega = 243$ est une racine 2^{25} de l'unité. Si on prend plus gros, on risque d'avoir des soucis avec les entiers bornés de Caml...