FFT

Marc Lorenzi

22 avril 2021

1 Racines de l'unité

1.1 Introduction

Soi $A \subset \mathbb{C}^*$ un ensemble fini de cardinal $n \geq 1$. Définissons par récurrence sur n la propriété « A est *rétractable* ».

- Le seul ensemble rétractable de cardinal 1 est $\{1\}$.
- Pour tout $n \geq 2$, un ensemble A de cardinal n est rétractable si n est pair et $A^2 = \{x^2, x \in A\}$ est rétractable, de cardinal $\frac{n}{2}$.

Proposition 1. *Soit $n \geq 1$. Un ensemble A de cardinal n est rétractable si et seulement si*

- *$n = 2^p$ est une puissance de 2.*
- *$A = \mathbb{U}_{2^p}$, l'ensemble des racines 2^p-ièmes de l'unité.*

Démonstration. On fait une récurrence forte sur n. Pour $n = 1$ il n'y a rien à prouver. Soit $n \geq 2$. Supposons la propriété vraie pour tous les entiers strictement inférieurs à n. Soit A un ensemble rétractable de cardinal n. Alors, $n = 2n'$ est pair, et A^2 est rétractable de cardinal n'. Comme $n' < n$, par l'hypothèse de récurrence, $n' = 2^p$ est une puissance de 2, donc $n = 2n' = 2^{p+1}$ est aussi une puissance de 2. De plus, toujours par l'hypothèse de récurrence,

$$A^2 = U_{2^p}$$

Montrons que

$$A = U_{2^{p+1}}$$

Soit $x \in A$. On a $x^2 \in A^2 = \mathbb{U}_{2^p}$. Ainsi,

$$x^{2^{p+1}} = (x^2)^{2^p} = 1$$

ce qui prouve que $x \in \mathbb{U}_{2^{p+1}}$. Ainsi,

$$A \subset \mathbb{U}_{2^{p+1}}$$

Les deux ensembles A et $\mathbb{U}_{2^{p+1}}$ ont même cardinal 2^{p+1}, ils sont donc égaux. □

1.2 Évaluation d'un polynôme

Pour tout entier $n \geq 1$, notons $\mathbb{C}_{<n}[X]$ l'ensemble des polynômes à coefficients complexes de degré strictement inférieur à n.

Soit $n \geq 1$. Soit $P \in \mathbb{C}_{<2n}[X]$. Écrivons

$$P = \sum_{k<2n} a_k X^k$$

où les a_k appartiennent à $\mathbb{C}$. P s'écrit sous la forme

$$\begin{aligned} P &= \sum_{k<n} a_{2k} X^{2k} + \sum_{k<n} a_{2k+1} X^{2k+1} \\ &= \sum_{k<n} a_{2k} (X^2)^k + X \sum_{k<n} a_{2k+1} (X^2)^k \\ &= P_1(X^2) + X P_2(X^2) \end{aligned}$$

où $P_1 = \sum_{k<n} a_{2k} X^k$ et $P_2 = \sum_{k<n} a_{2k+1} X^k$ appartiennent à $\mathbb{C}_{<n}[X]$.

Soit A un ensemble rétractable de cardinal $2n$. On peut calculer l'ensemble $\{P(x), x \in A\}$ comme suit.

- Si $n = 1$, c'est immédiat puisque le polynôme P est constant.
- Sinon, on calcule récursivement $\{P_1(x), x \in A^2\}$ et $\{P_2(x), x \in A^2\}$. On en déduit, pour chaque $x \in A$, la valeur de $P(x)$.

Le nombre $T(n)$ d'opérations arithmétiques nécessaires pour effectuer ce calcul vérifie

$$T(2n) = 2T(n) + O(n)$$

La quantité $T(n)$ provient des deux calculs récursifs, et le $O(n)$ est le coût de la recombinaison.

Proposition 2.

$$T(n) = O(n \log n)$$

Démonstration. Notons $T'(p) = T(2^p)$. On a pour tout $p \in \mathbb{N}$,

$$T'(p) = 2T'(p-1) + O(2^p)$$

Précisément, il existe une constante C telle que, pour tout $p \in \mathbb{N}$,

$$T'(p) \leq 2T'(p-1) + C2^p$$

Montrons que par récurrence sur p que pour tout $p \in \mathbb{N}$, $T'(p) \leq Cp2^p$. Pour $p = 0$, le polynôme P est constant, donc $T'(0) = T(1) = 0$. Soit $p \geq 1$. Supposons que $T'(p-1) \leq C(p-1)2^{p-1}$. On a alors

$$T'(p) \leq 2T'(p-1) + C2^p \leq 2C(p-1)2^{p-1} + C2^p = Cp2^p$$

Revenant à T_n (rappelons que n est une puissance de 2),

$$T_n = T'(\lg n) \leq C(\lg n)2^{\lg n} = Cn \lg n$$

□

2 La FFT

2.1 Introduction

Pour tout entier $n \geq 1$, notons $\omega_n = e^{-\frac{2i\pi}{n}}$. On a ainsi

$$\mathbb{U}_n = \{\omega_n^j, 0 \leq j < n\}$$

Soit $p \in \mathbb{N}$. Soit $n = 2^p$. Soit $a = (a_0, \ldots, a_{n-1}) \in \mathbb{C}^n$. Soit $P = \sum_{k<n} a_k X^k$. Nous allons utiliser la section précédente pour calculer efficacement par récurrence la liste $(P(\omega_n^j))_{0 \leq j < n}$.

Si $p = 0$, c'est immédiat. $\omega_1 = 1$, $\mathbb{U}_1 = \{1\}$ et P est constant égal à a_0.

Supposons maintenant $p \geq 1$. Soient $a' = (a_{2k})_{0 \leq k < n/2}$ et $a'' = (a_{2k+1})_{0 \leq k < n/2}$. Soient P' et P'' les polynômes de $\mathbb{C}_{<n/2}$ correspondant à a' et a''. Supposons calculés $(P'(\omega_{n/2}^j))_{0 \leq j < n/2}$ et $(P''(\omega_{n/2}^j))_{0 \leq j < n/2}$.

On a alors pour tout $j \in [\![0, n-1]\!]$,

$$\begin{aligned} P(\omega_n^j) &= P'(\omega_n^{2j}) + \omega_n^j P''(\omega_n^{2j}) \\ &= P'(\omega_{n/2}^j) + \omega_n^j P''(\omega_{n/2}^j) \end{aligned}$$

Deux cas se présentent.

- Si $0 \leq j < n/2$, les nombres $P'(\omega_{n/2}^j)$ et $P''(\omega_{n/2}^j)$ sont connus, d'où la valeur de $P(\omega_n^j)$.
- Si $n/2 \leq j < n$, posons $j = n/2 + j'$ où $0 \leq j' < n/2$. On a alors

$$\begin{aligned} P(\omega_n^j) &= P'(\omega_{n/2}^j) + \omega_n^j P''(\omega_{n/2}^j) \\ &= P'(\omega_{n/2}^{j'}) + \omega_n^{n/2+j'} P''(\omega_{n/2}^{j'}) \\ &= P'(\omega_{n/2}^{j'}) - \omega_n^{j'} P''(\omega_{n/2}^{j'}) \end{aligned}$$

où l'on a utilisé les relations $\omega_{n/2}^{n/2} = 1$ et $\omega_n^{n/2} = -1$.

2.2 L'algorithme

On déduit de ce qui précède un algorithme dont voici le code Python. La fonction écrite est récursive. On suppose définis le nombre π et l'exponentielle complexe, par exemple par l'import du module `cmath`. L'idée du code ci-dessous est d'être le plus compact et le plus lisible possible.

```
def fft_rec(x):
    n = len(x)
    if n == 1: return x[:]
    else:
```

```
    x1 = [x[2 * k] for k in range(n // 2)]
    x2 = [x[2 * k + 1] for k in range(n // 2)]
    omega = cmath.exp(-2 * 1j * cmath.pi / n)
    U1 = fft_rec(x1)
    U2 = fft_rec(x2)
    V1 = [U1[j] + omega ** j * U2[j] for j in range(n // 2)]
    V2 = [U1[j] - omega ** j * U2[j] for j in range(n // 2)]
    return V1 + V2
```

2.3 Complexité

La complexité en temps T_n de la fonction `fft_rec`, évaluée en nombre d'opérations arithmétiques, a été étudiée dans la première section de l'article. On a

$$T_n = O(n \log n)$$

Quelle est la complexité en espace ? Notons E_n cette complexité. On a $E_1 = O(1)$ et, pour tout entier $n \geq 1$,

$$E_{2n} = 2E_n + O(n)$$

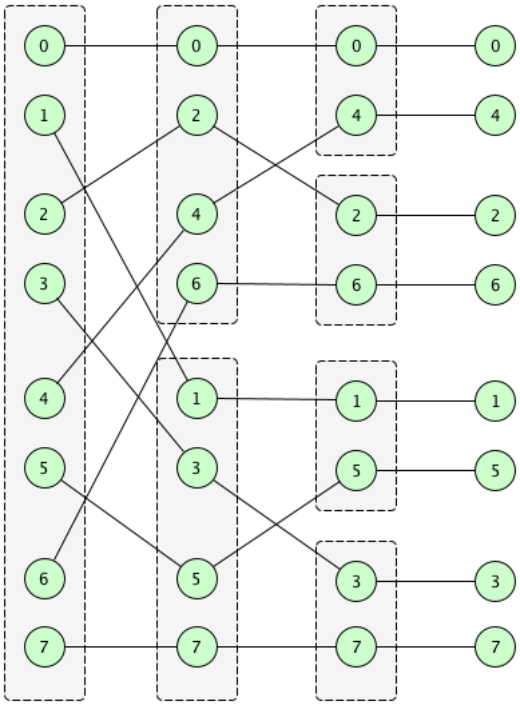
E_n vérifie donc la même relation de récurrence que T_n. On a ainsi

$$E_n = O(n \log n)$$

3 Élimination des appels récursifs

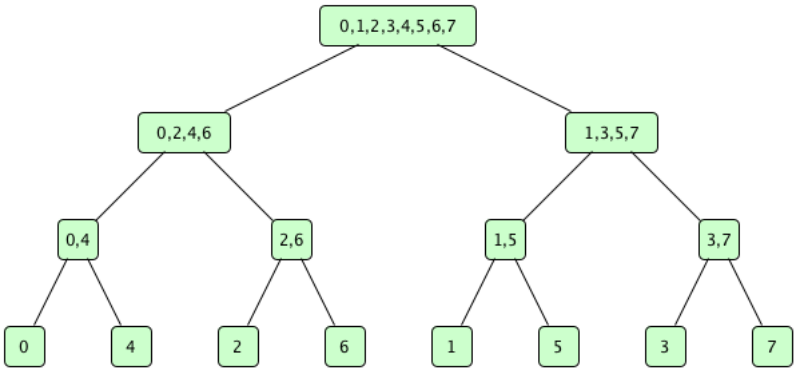
3.1 Les appels récursifs

Le schéma ci-dessous représente la succession des appels récursifs lors du calcul de la FFT d'une liste de longueur 8.



Chacun des sommets du graphe est étiqueté par l'indice d'origine de l'élément de la liste. Dans la colonne 0 se trouve la liste entière, avec ses éléments numérotés de 0 à 7. L'appel à la fonction `fft_rec` provoque deux appels récursifs sur les sous-listes des éléments d'indices pairs et des éléments d'indices impairs, ce que l'on trouve dans la colonne 1. Ces deux appels récursifs provoquent 4 appels à `fft_rec` sur des listes de taille 2, qui provoquent eux-mêmes 8 appels récursifs sur des listes de taille 1, visibles dans la dernière colonne du graphe. On peut également visualiser ceci au moyen d'un arbre. De façon plus générale, pour tout entier $p \geq 0$, cet arbre contient à sa racine la liste des entiers entre 0 et $2^p - 1$. Pour chaque noeud de l'arbre :

- Si ce noeud est étiqueté par une liste de longueur 1, c'est une feuille.
- Sinon, il est étiqueté par une liste s de longueur $m \geq 2$ où m est une puissance de 2. Il possède alors deux fils étiquetés par des listes de longueur $\frac{m}{2}$, un fils gauche qui est la liste des éléments de s d'indice pair, et un fils droit qui est la liste des éléments de s d'indice impair.



3.2 Renversement de bits

Soient p un entier naturel. Définissons la *fonction de renversement de bits*

$$\rho_p : [\![0, 2^p - 1]\!] \to [\![0, 2^p - 1]\!]$$

comme suit. Si $0 \leq a < 2^p$,

$$a = \sum_{k=0}^{p-1} a_k 2^k$$

où les a_k valent 0 ou 1, alors

$$\rho_p(a) = \sum_{k=0}^{p-1} a_{p-1-k} 2^k$$

Remarquons que si $p = 0$, alors $a = 0$ et $\rho_0(0) = 0$ (somme vide). La fonction ρ_p est une bijection égale à sa réciproque.

Soit p un entier naturel. Soit $a = \sum_{k=0}^{p-1} a_k 2^k$, où les a_k valent 0 ou 1. Il est facile d'écrire un code efficace calculant $\rho_p(a)$. En effet, on a

$$\rho_p(a) = \sum_{k=0}^{p-1} a_{p-1-k} 2^k = P(2)$$

où

$$P = \sum_{k=0}^{p-1} a_{p-1-k} X^k$$

$\rho_p(a)$ peut donc être calculé de façon efficace par l'algorithme de Hörner. Considérons la fonction suivante.

```
def rev_bit(a, p):
    b = 0
    for i in range(1, p + 1):
        b = 2 * b + a % 2
        a = a // 2
    return b
```

Pour $i = 0, \ldots, p$, soient α_i et β_i les valeurs des variables a et b après la $i$ème itération (pour $i = 0$, comprendre « avant la première itération »).

Proposition 3. *On a pour tout* $i \in [\![0, p]\!]$,

$$\alpha_i = \sum_{k=i}^{p-1} a_k 2^{k-i}$$

$$\beta_i = \sum_{k=0}^{i-1} a_{i-1-k} 2^k$$

Démonstration. Faisons une récurrence sur i. On a

$$\alpha_0 = a = \sum_{k=i}^{p-1} a_k 2^{k-i}$$

et

$$\beta_0 = 0 = \sum_{k=0}^{-1} a_{-1-k} 2^k \quad \text{(somme vide)}$$

Soit $i \in [\![0, p-1]\!]$. Supposons les égalités vérifiées pour i. On a

$$\alpha_{i+1} = \lfloor \alpha_i / 2 \rfloor = \frac{1}{2} \sum_{k=i+1}^{p-1} a_k 2^{k-i} = \sum_{k=i+1}^{p-1} a_k 2^{k-(i+1)}$$

et

$$\begin{aligned}
\beta_{i+1} &= 2\beta_i + \alpha_i \bmod 2 \\
&= 2 \sum_{k=0}^{i-1} a_{i-1-k} 2^k + a_i \\
&= \sum_{k=0}^{i-1} a_{i-1-k} 2^{k+1} + a_i \\
&= \sum_{k=1}^{i} a_{i-k} 2^k + a_i \\
&= \sum_{k=0}^{i} a_{i-k} 2^k
\end{aligned}$$

□

Corollaire 4. *L'appel* `rev_bits(a, p)` *renvoie* $\rho_p(a)$.

Démonstration. Après la $p$ième itération, la valeur de la variable b est

$$\beta_p = \sum_{k=0}^{p-1} a_{p-1-k} 2^k = \rho_p(a)$$

□

Nous aurons besoin dans ce qui va suivre de certaines relations de récurrence sur la fonction ρ_p.

Lemme 5. *Soit p un entier naturel.*

- *Pour tout $a \in [\![0, 2^{p-1}-1]\!]$, $\rho_p(a) = 2\rho_{p-1}(a)$.*
- *Pour tout $a \in [\![2^{p-1}, 2^p-1]\!]$, $\rho_p(a) = 2\rho_{p-1}(a-2^{p-1})+1$.*

Démonstration. Soit $a \in [\![0, 2^p-1]\!]$. Écrivons

$$a = \sum_{k=0}^{p-1} a_k 2^k$$

où les a_k valent 0 ou 1.

- Supposons $0 \leq a < 2^{p-1}$. On a donc $a_{p-1} = 0$. On a

$$\begin{aligned}\rho_p(a) &= \sum_{k=0}^{p-1} a_{p-1-k}2^k = \sum_{k=1}^{p-1} a_{p-1-k}2^k \\ &= 2\sum_{k=1}^{p-1} a_{p-1-k}2^{k-1} = 2\sum_{k=0}^{p-2} a_{p-2-k}2^k \\ &= 2\rho_{p-1}(a)\end{aligned}$$

- Supposons $2^{p-1} \leq a < 2^p$. On a donc $a_{p-1} = 1$. On a

$$\begin{aligned}\rho_p(a) &= \sum_{k=0}^{p-1} a_{p-1-k}2^k = 1 + \sum_{k=1}^{p-1} a_{p-1-k}2^k \\ &= 1 + 2\sum_{k=1}^{p-1} a_{p-1-k}2^{k-1} = 1 + 2\sum_{k=0}^{p-2} a_{p-2-k}2^k \\ &= 1 + 2\rho_{p-1}(a-2^{p-1})\end{aligned}$$

□

3.3 FFT et renversements de bits

Proposition 6. *Les feuilles de l'arbre des appels récursifs de la FFT sont, de gauche à droite, étiquetées par les entiers $\rho_p(a)$, pour $0 \leq a < 2^p$.*

Démonstration. Montrons par récurrence sur p le résultat plus général suivant : si la racine de l'arbre est étiquetée par la liste d'entiers $(x_0, x_1, \ldots, x_{2^p-1})$, alors les feuilles de l'arbre sont, de gauche à droite, étiquetées par les entiers $\rho_p(x_a)$, $0 \leq a < 2^p$.

- Si $p = 0$, l'arbre est composé d'une unique feuille étiquetée $0 = \rho_0(0)$.
- Soit $p \geq 1$. Supposons la propriété vraie pour l'entier $p-1$. Considérons un arbre dont la racine est étiquetée par la liste d'entiers $(x_0, x_1, \ldots, x_{2^p-1})$. Les deux appels récursifs de la FFT créent deux fils.

Le fils gauche a sa racine étiquetée par la liste d'entiers $(x_{2i})_{0\leq i<2^{p-1}} = (y_i)_{0\leq i<2^{p-1}}$. Par l'hypothèse de récurrence, les feuilles du premier fils sont étiquetées de gauche à droite, pour $0 \leq i < 2^{p-1}$, par

$$y_{\rho_{p-1}(i)} = x_{2\rho_{p-1}(i)} = x_{\rho_p(i)}$$

Le fils droit a sa racine étiquetée par la liste d'entiers $(x_{2i+1})_{0\leq i<2^{p-1}} = (y_i)_{0\leq i<2^{p-1}}$. Par l'hypothèse de récurrence, les feuilles du second fils sont étiquetées de gauche à droite, pour $0 \leq i < 2^{p-1}$, par

$$y_{\rho_{p-1}(i)} = x_{2\rho_{p-1}(i)+1} = x_{\rho_p(i+2^{p-1})}$$

Ainsi, l'arbre complet a ses feuilles étiquetées, de gauche à droite, par les$x_{\rho_p(i)}$, $0, \leq i < 2^{p-1}$, suivis des $x_{\rho_p(i+2^{p-1})}$, toujours pour $2^{p-1} \leq i < 2^p$. Les feuilles de l'arbre complet sont donc étiquetées par les $x_{\rho_p(i)}$, $0 \leq i < 2^p$.

□

Nous sommes donc en mesure, étant donnée une liste $x = (x_0, \ldots, x_n)$ de longueur $n = 2^p$, de créer la liste $(x_{\rho_p(k)})_{0\leq k<2^p}$. La fonction `shuffle` effectue ce travail.

```
def shuffle(x):
    n = len(x)
    p = log2(n)
    y = n * [0]
    for k in range(n):
        y[rev_bit(k, p)] = x[k]
    return y
```

On suppose dans cette fonction que l'on dispose d'une fonction `log2` renvoyant le logarithme en base 2 d'un entier n, lorsque n est une puissance de 2. L'écriture d'une telle fonction est immédiate.

3.4 Une version itérative de la FFT

Il est maintenant « facile » d'obtenir une version itérative de la FFT. Pour cela, rappelons-nous le graphe 3.1 vu un peu plus haut.

Si nous lisons le graphe de la droite vers la gauche, nous voyons que pour réaliser la FFT d'une liste de longueur 2^p, (pour ce graphe, $p = 3$), il suffit d'effectuer les opérations suivantes.

- Étape 0 : Appeler la fonction `shuffle` pour placer les éléments de la liste dans le « bon » ordre.
- Étape 1 : Combiner deux par deux les 2^p éléments de la liste pour obtenir 2^{p-1} sous-listes de longueur $2^1 = 2$ qui sont, dans la version récursive, les listes renvoyées par `fft_rec` lorsqu'elle est appelée sur les listes de taille 2.
- Étape 2 : Combiner deux par deux ces 2^{p-1} listes de longueur 2 pour obtenir 2^{p-2} sous-listes de longueur $2^2 = 4$ qui sont, dans la version récursive, les listes renvoyées par `fft_rec` lorsqu'elle est appelée sur les listes de taille 4.

- ...
- Étape p : Combiner deux par deux ces 2 listes de longueur 2^{p-1} pour obtenir 1 liste de longueur 2^p qui est, dans la version récursive, la liste renvoyée par `fft_rec` lorsqu'elle est appelée sur la liste de taille 2^p de départ.

La façon de combiner ces listes est la même que celle utilisée dans la fonction `fft_rec`.

```
def fft(x):
    y = shuffle(x)
    n = len(x)
    p = log2(n)
    for s in range(1, p + 1):
        m = 2 ** s
        omega = exp(-2 * 1j * pi / m)
        for k in range(0, n, m):
            for j in range(m // 2):
                u = y[k + j]
                v = omega ** j * y[k + j + m // 2]
                y[k + j]          = u + v
                y[k + j + m // 2] = u - v
    return y
```

3.5 Complexité

Nous n'en ferons pas ici une preuve formelle, mais les fonctions `fft` et `fft_rec` effectuent *exactement* les mêmes opérations arithmétiques. Elles renvoient *exactement* les mêmes résultats, avec une différence des coefficients des listes renvoyées égale au 0 machine. La complexité en termes d'opérations arithmétiques de ces deux fonctions est la même.

En revanche, la fonction `fft` possède une meilleure complexité en espace. Mise à part la liste contenant le résultat de la FFT qui est de longueur n, et utilise un espace en $O(n)$, cette fonction utilise un espace en $O(\log n)$. Il est même possible de réaliser la FFT sur place, dans la liste de départ, pour obtenir ainsi une complexité spatiale globale en $O(\log n)$. Il suffit pour cela de modifier la fonction `shuffle` comme suit.

```
def shuffle_in_place(x):
    n = len(x)
    p = log2(n)
    for k in range(n):
        l = rev_bit(k, p)
        if k < l:
            x[k], x[l] = x[l], x[k]
```