FFT

Biến đổi Fourier nhanh - Fast Fourier transform

Người viết:

- Nguyễn Hoàng Vũ - Trường Đại học Công nghệ, ĐHQGHN

Reviewer:

- Trần Xuân Bách - Đại học Chicago
- Nguyễn Minh Nhật - Trường THPT chuyên Khoa học Tự nhiên, ĐHQGHN

Các bài toán tổ hợp ngày càng xuất hiện nhiều trong các cuộc thi lập trình thi đấu và thường xuyên nắm giữ các vị trí khó nhất. Bài viết này sẽ giới thiệu về một công cụ quan trọng để giải các bài toán tổ hợp, đó là **Biến đổi Fourier nhanh - Fast Fourier transform**, hay còn được viết tắt là **FFT**.

1. Các kiến thức cần biết

1.1. Số phức (Complex number)

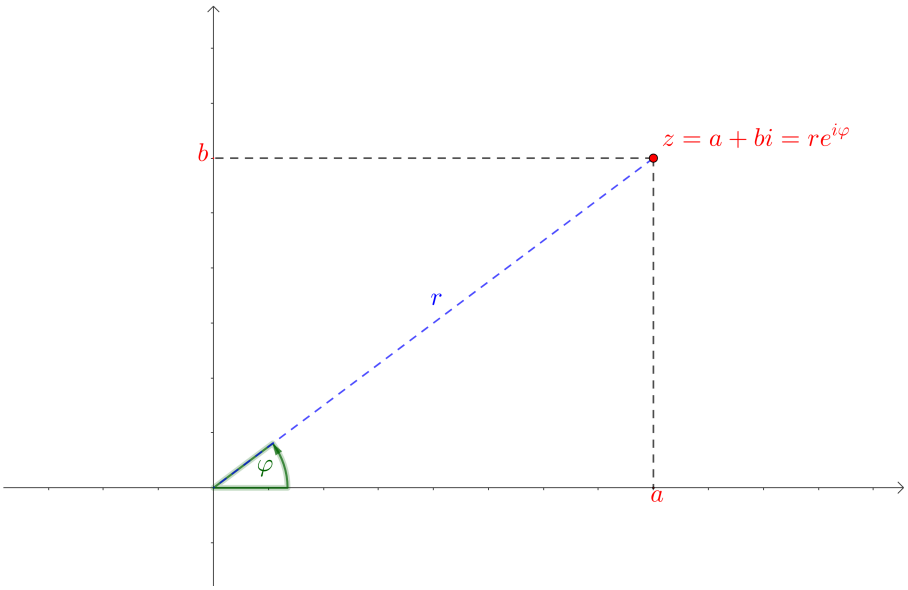
Định nghĩa

Số phức là các số có dạng $z = a + bi$ trong đó $a, b \in \mathbb{R}$ và $i^2 = -1$. Số i được gọi là đơn vị ảo. Tập hợp tất cả các số phức được kí hiệu là $\mathbb{C}$.

Ta có thể biểu diễn số phức z trên mặt phẳng toạ độ bằng vectơ (a, b).

- Phần thực (Real) và phần ảo (Imaginary): $Re(z) = a, Im(z) = b$.
- Số phức liên hợp (Conjugate): $\overline{z} = a - bi$.
- Môđun (Modulus): $|z| = r = \sqrt{a^2 + b^2}$.
- Acgumen (Argument): Góc có hướng từ trục thực Ox đến vectơ (a, b), tức góc φ trong hình dưới đây.
- Dạng lượng giác (Polar form): $z = r(\cos\varphi + i\sin\varphi) = re^{i\varphi} = r\exp(i\varphi)$.

Về biểu diễn $z = r\exp(i\varphi)$, các bạn có thể tìm hiểu ở Công thức Euler (Wikipedia) .



Hình 1. Biểu diễn số phức trên mặt phẳng

Các phép toán

Xét hai số phức $z_1 = a_1 + b_1 i = r_1 \exp(i\varphi_1)$ và $z_2 = a_2 + b_2 i = r_2 \exp(i\varphi_2)$:

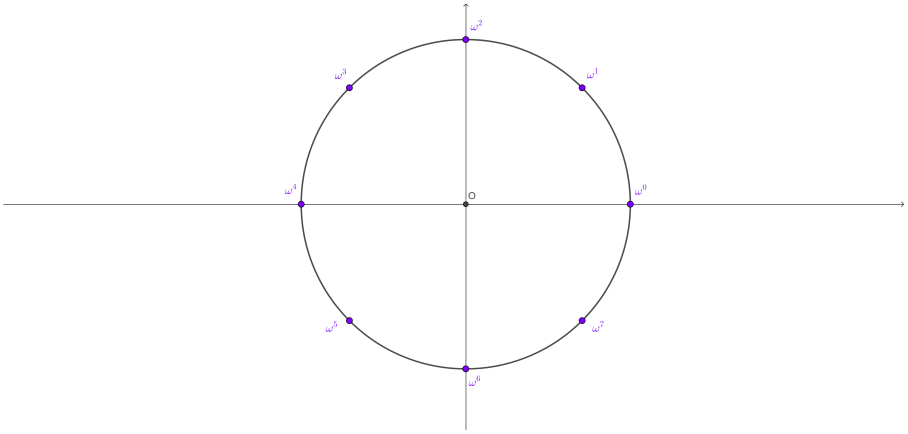
- Phép cộng: $z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)i$.
- Phép trừ: $z_1 - z_2 = (a_1 - a_2) + (b_1 - b_2)i$.
- Phép nhân: $z_1 z_2 = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1)i = r_1 r_2 \exp(i(\varphi_1 + \varphi_2))$.
- Phép chia: $\frac{z_1}{z_2} = \frac{z_1 \overline{z_2}}{z_2 \overline{z_2}} = \frac{z_1 \overline{z_2}}{|z_2|^2}$.

Phép cộng (trừ) hai số phức tương đương với phép cộng (trừ) hai vectơ biểu diễn chúng. Khi nhân hai số phức, ta nhân môđun của chúng và cộng acgumen của chúng.

Căn đơn vị (Root of unity)

Xét một số nguyên dương n:

- Căn đơn vị cấp n là các số phức z thoả mãn $z^n = 1$.
- Công thức tổng quát cho các căn đơn vị cấp n: $z_{n,i} = \exp(\frac{2k\pi i}{n})$ với k từ 0 đến $n-1$. Kí hiệu $\omega_n = z_{n,1}$ thì ta còn có thể viết dưới dạng $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$.
- Các căn đơn vị nằm trên đường tròn đơn vị tạo thành một đa giác đều n đỉnh.



Hình 2. Biểu diễn các căn đơn vị cấp 8 trên mặt phẳng

Một số tính chất của căn đơn vị

- $\omega_n^j = \omega_n^{j \bmod n}$.
- Với n chẵn, $\omega_n^{n/2} = -1$, từ đây suy ra $\omega_n^{j+n/2} = -\omega_n^j$.
- Với n chẵn, $\omega_n^2 = \omega_{n/2}$.

1.2. Ma trận

Các bạn có thể tham khảo bài viết Nhân ma trận (VNOI) .

2. Thuật toán FFT

2.1. Bài toán

Cho hai dãy $a = (a_0, a_1, \ldots, a_{n-1})$ và $b = (b_0, b_1, \ldots, b_{n-1})$. Tính dãy $c = (c_0, c_1, \ldots, c_{2n-2})$ được cho bởi công thức: $c_k = \sum a_j b_{k-j}$ với k từ 0 đến $2n-2$.

Dãy c được định nghĩa như trên được gọi là tích chập (convolution) của hai dãy a và b.

2.2. Ý tưởng

Biểu thức của dãy c phía trên gợi lại cho chúng ta về phép nhân đa thức. Cụ thể, định nghĩa:

$$A(z) = a_0 z^0 + a_1 z^1 + \ldots + a_{n-1} z^{n-1}$$
$$B(z) = b_0 z^0 + b_1 z^1 + \ldots + b_{n-1} z^{n-1}$$

thì c chính là dãy hệ số của đa thức $C(z) = A(z) \cdot B(z)$.

Có thể thấy rằng việc tính đa thức C theo định nghĩa sẽ có độ phức tạp thời gian $\mathcal{O}(n^2)$, không đủ nhanh khi độ dài n khá lớn, ta cần một hướng tiếp cận khác. Giả sử ta tính được giá trị của A và B tại m điểm khác nhau:

$$A(z_0), A(z_1), \ldots, A(z_{m-1})$$
$$B(z_0), B(z_1), \ldots, B(z_{m-1})$$

thì giá trị của C tại các điểm tương ứng là: $C(z_j) = A(z_j) \cdot B(z_j)$.

Ta có định lý quan trọng sau đây:

Định lý nội suy đa thức. Cho m cặp số phức $(u_0, v_0), (u_1, v_1), \ldots, (u_{m-1}, v_{m-1})$ thoả mãn $z_i \neq z_j$ với mọi $0 \leq i < j < m$, tồn tại duy nhất một đa thức P có bậc không quá $m-1$ thoả mãn $P(u_i) = v_i$ với mọi $0 \leq i < m$.

Ví dụ:

- Có duy nhất một đường thẳng (đa thức bậc 1) đi qua hai điểm bất kì trên mặt phẳng.
- Có duy nhất một parabol (đa thức bậc 2) đi qua ba điểm bất kì trên mặt phẳng.

Ý tưởng của thuật toán FFT là chọn ra một tập điểm $z_0, z_1, \ldots, z_{m-1}$ sao cho ta có thể tính nhanh giá trị của đa thức A và B trên đó, đồng thời có thể khôi phục được đa thức C dựa trên $C(z_0), C(z_1), \ldots, C(z_{m-1})$.

2.3. Thuật toán

Biến đổi xuôi

Ta có một đa thức $A(z) = a_0 z^0 + a_1 z^1 + \ldots + a_{n-1} z^{n-1}$. Không mất tính tổng quát, giả sử n là một luỹ thừa của 2 hay $n = 2^k$ với $k \in \mathbb{N}$. Nếu n không phải là một lũy thừa của 2, ta thêm các số hạng $a_i z^i$ bị thiếu và cho các hệ số a_i bằng 0.

Tập số mà FFT chọn để tính là tập các căn đơn vị cấp n, tức $\{\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}\}$ (nhắc lại $\omega_n = \exp(\frac{2\pi i}{n})$).

Định nghĩa. Cho một dãy $a_0, a_1, \ldots, a_{n-1}$, **Biến đổi Fourier nhanh - Fast Fourier Transform - FFT** là bất kì thuật toán nào tính dãy $A(\omega_n^0), A(\omega_n^1), \ldots, A(\omega_n^{n-1})$ trong thời gian $O(n \log n)$. Phép biến đổi này bản thân nó được gọi là **Biến đổi Fourier rời rạc - Discrete Fourier Transform - DFT**.

$$\mathrm{DFT}(a_0, a_1, \ldots, a_{n-1}) = (A(\omega_n^0), A(\omega_n^1), \ldots, A(\omega_n^{n-1}))$$

Bài viết này sẽ giới thiệu về một thuật toán FFT thông dụng là thuật toán *Cooley-Tukey*.

Ta sẽ biểu diễn bài toán dưới dạng ma trận:

$$\begin{bmatrix} A(\omega^0) \\ A(\omega^1) \\ \vdots \\ A(\omega^{n-1}) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \ldots & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \ldots & \omega^{n-1} \\ \omega^0 & \omega^2 & \omega^4 & \ldots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Trước khi đi vào trường hợp tổng quát, ta sẽ xem xét một ví dụ với $n = 8$:

$$\begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^1 & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ \omega^0 & \omega^4 & \omega^0 & \omega^4 & \omega^0 & \omega^4 & \omega^0 & \omega^4 \\ \omega^0 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega^1 & \omega^6 & \omega^3 \\ \omega^0 & \omega^6 & \omega^4 & \omega^2 & \omega^0 & \omega^6 & \omega^4 & \omega^2 \\ \omega^0 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix}$$

Các số mũ trên bảng đã được lấy dư cho 8. Ta sẽ đưa các cột màu đỏ (chỉ số chẵn) về bên trái, các cột màu xanh (chỉ số lẻ) về bên phải và chia ma trận thành bốn ma trận con như dưới đây. Chú ý rằng việc này cũng thay đổi thứ tự của vectơ hệ số.

$$\begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^1 & \omega^3 & \omega^5 & \omega^7 \\ \omega^0 & \omega^4 & \omega^0 & \omega^4 & \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ \omega^0 & \omega^6 & \omega^4 & \omega^2 & \omega^3 & \omega^1 & \omega^7 & \omega^5 \\ \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^4 & \omega^4 & \omega^4 & \omega^4 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^5 & \omega^7 & \omega^1 & \omega^3 \\ \omega^0 & \omega^4 & \omega^0 & \omega^4 & \omega^6 & \omega^2 & \omega^6 & \omega^2 \\ \omega^0 & \omega^6 & \omega^4 & \omega^2 & \omega^7 & \omega^5 & \omega^3 & \omega^1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_2 \\ a_4 \\ a_6 \\ a_1 \\ a_3 \\ a_5 \\ a_7 \end{bmatrix}$$

Nhận thấy rằng các hệ số tương ứng ở hai ma trận con bên trái bằng nhau và các hệ số tương ứng ở hai ma trận con bên phải trái dấu, do tính chất $\omega^{n/2} = -1$.

Vậy biểu thức cần tính có dạng:

$$\begin{bmatrix} A & B \\ A & -B \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} AX + BY \\ AX - BY \end{bmatrix}$$

Để ý rằng:

$$B = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^1 & \omega^3 & \omega^5 & \omega^7 \\ \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ \omega^3 & \omega^1 & \omega^7 & \omega^5 \end{bmatrix} = \begin{bmatrix} \omega^0 & & & \\ & \omega^1 & & \\ & & \omega^2 & \\ & & & \omega^3 \end{bmatrix} \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^4 & \omega^0 & \omega^4 \\ \omega^0 & \omega^6 & \omega^4 & \omega^2 \end{bmatrix}$$

Vậy ta chỉ cần tính AX và AY là đủ để tính kết quả. Mặt khác, có thể thấy tính AX và AY tương đương với tính $\mathrm{DFT}(a_0, a_2, a_4, a_6)$ và $\mathrm{DFT}(a_1, a_3, a_5, a_7)$.

Xét trường hợp tổng quát, với $0 \le i < \frac{n}{2}$, ta có (chú ý i ở đây là chỉ số, không phải đơn vị ảo):

$$\begin{aligned} A(\omega_n^i) &= \sum_j a_j \omega_n^{ij} = \sum_k a_{2k} \omega_n^{2ki} + \sum_k a_{2k+1} \omega_n^{(2k+1)i} \\ &= \sum_k a_{2k} (\omega_n^2)^{ik} + \omega_n^i \sum_k a_{2k+1} (\omega_n^2)^{ik} \\ &= \sum_k a_{2k} \omega_{n/2}^{ik} + \omega_n^i \sum_k a_{2k+1} \omega_{n/2}^{ik} \end{aligned}$$

$$\begin{aligned} A(\omega_n^{i+n/2}) &= \sum_j a_j \omega_n^{(i+n/2)j} = \sum_k a_{2k} \omega_n^{2k(i+n/2)} + \sum_k a_{2k+1} \omega_n^{(2k+1)(i+n/2)} \\ &= \sum_k a_{2k} \omega_n^{2ki} + \sum_k a_{2k+1} \omega_n^{2ki+i+n/2} \\ &= \sum_k a_{2k} (\omega_n^2)^{ik} - \omega_n^i \sum_k a_{2k+1} (\omega_n^2)^{ik} \\ &= \sum_k a_{2k} \omega_{n/2}^{ik} - \omega_n^i \sum_k a_{2k+1} \omega_{n/2}^{ik} \end{aligned}$$

Nếu coi $\mathrm{DFT}(a_0, a_2, \ldots, a_{n/2-2}) = A_0$ và $\mathrm{DFT}(a_1, a_3, \ldots, a_{n/2-1}) = A_1$ thì hai biểu thức trên có thể viết lại thành:

$$A(\omega_n^i) = A_0(\omega_{n/2}^i) + \omega_n^i A_1(\omega_{n/2}^i)$$
$$A(\omega_n^{i+n/2}) = A_0(\omega_{n/2}^i) - \omega_n^i A_1(\omega_{n/2}^i)$$

Vậy ta cần tính A_0 và A_1 là hai bài toán với kích thước giảm đi một nửa.

Độ phức tạp. Thuật toán FFT là một thuật toán chia để trị nên ta dễ thấy nó có độ phức tạp $O(n \log_2 n)$.

Biến đổi ngược

Bổ đề. $\mathrm{DFT}(\mathrm{DFT}(a_0, a_1, \ldots, a_{n-2}, a_{n-1})) = (na_0, na_{n-1}, na_{n-2} \ldots, na_1)$.

Chứng minh:
Đặt $(b_0, b_1, \ldots, b_{n-1}) = \mathrm{DFT}(a_0, a_1, \ldots, a_{n-1})$ và $(c_0, c_1, \ldots, c_{n-1}) = \mathrm{DFT}(b_0, b_1, \ldots, b_{n-1})$.

Ta có:

$$c_l = \sum_k c_k \omega^{kl} = \sum_k \omega^{kl} \sum_j a_j \omega^{jk} = \sum_j a_j \sum_k \omega^{k(j+l)}$$

Xét $j + l \not\equiv 0 \pmod{n}$, ta có:

$$\sum_k \omega^{k(j+l)} = \frac{(\omega^{j+l})^n - 1}{\omega^{j+l} - 1} = 0$$

bởi w^{j+l} là một căn đơn vị.

Từ đây ta suy ra $c_l = na_j$ với $j + l \equiv 0 \pmod{n}$.

Ngoài ra, nếu ta sử dụng $\omega^{-1} = \exp(\frac{-2\pi i}{n})$ thay cho ω ở lần DFT thứ hai, kết quả ta nhận được sẽ là $(na_0, na_1, \ldots, na_{n-1})$ (bạn đọc tự chứng minh).

Ta sẽ cài đặt chung cả biến đổi xuôi và ngược:

```
using cd = complex<long double>;

void fft(vector<cd> &a, bool invert) {
    /// invert = true tương ứng với biến đổi ngược
    int n = a.size();
    if (n == 1) return;
    vector<cd> a0, a1;
    for (int i = 0; i < n / 2; i++) {
        a0.push_back(a[2 * i]);
        a1.push_back(a[2 * i + 1]);
    }
    fft(a0, invert); fft(a1, invert);
    cd w = 1, wn = polar(1.0L, acos(-1.0L) / n * (invert ? -2 :
    /// polar(r, t) = r * exp(it) và acos(-1.0L) = pi
    /// thay wn = wn^-1 ở biến đổi ngược
    for (int i = 0; i < n / 2; i++) {
```

```
        a[i] = a0[i] + w * a1[i];
        a[i + n / 2] = a0[i] - w * a1[i];
        /// Ta sẽ chia 2 ở mỗi tầng đệ quy thay cho việc chia n
        if (invert) {
            a[i] /= 2; a[i + n / 2] /= 2;
        }
        w *= wn;
    }
}
```

Dưới đây là cài đặt để tính tích chập của hai dãy số:

```
vector<int> conv(const vector<int> &a, const vector<int> &b) {
    if (a.empty() || b.empty()) return {};
    vector<cd> fa(a.begin(), a.end());
    vector<cd> fb(b.begin(), b.end());
    int n = 1;
    while (n < int(a.size() + b.size()) - 1) n <<= 1;
    fa.resize(n); fb.resize(n);
    fft(fa, false); fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);
    vector<int> res(n);
    for (int i = 0; i < n; i++)
        res[i] = int(real(fa[i]) + 0.5);
    return res;
}
```

Cài đặt khử đệ quy

Ta xét các tầng đệ quy với $n = 8$:

- Tầng 3: $(a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$
- Tầng 2: $(a_0, a_2, a_4, a_6), (a_1, a_3, a_5, a_7)$
- Tầng 1: $(a_0, a_4), (a_2, a_6), (a_1, a_5), (a_3, a_7)$

Vậy nếu ta sắp xếp dãy a lại thành $(a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7)$ thì mỗi lần gọi đệ quy ở tầng thứ i sẽ thực hiện trên một đoạn dài 2^i.

Viết lại dãy chỉ số dưới dạng nhị phân:

$$(000, 100, 010, 110, 001, 101, 011, 111)$$

Có thể thấy rằng nếu ta đảo ngược thứ tự bit thì, ví dụ $100 \to 001$ thì ta nhận được dãy tăng dần từ 0 đến $n - 1$.

Gọi $rev(i)$ là số nhận được sau khi đảo ngược thứ tự bit của i, ta có $rev(i) = \dfrac{rev(i/2)|[(i \bmod 2) \cdot n]}{2}$ với $|$ thể hiện phép toán bitwise OR (bạn đọc tự chứng minh).

Ta có cài đặt sau:

```
void fft(vector<cd> &a, bool invert) {
    int n = a.size(), L = __builtin_ctz(n);
    vector<int> rev(n);
    for (int i = 0; i < n; i++) {
        rev[i] = (rev[i >> 1] | (i & 1) << L) >> 1;
        if (i < rev[i]) swap(a[i], a[rev[i]]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        cd wlen = polar(1.0L, acos(-1.0L) / len * (invert ? -2
        for (int i = 0; i < n; i += len) {
            cd w = 1;
            for (int j = 0; j < len / 2; j++) {
                cd u = a[i + j];
                cd v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }
    if (invert) {
        for (auto &x : a) x /= n;
    }
}
```

Vấn đề về độ chính xác

Ở cài đặt phía trên ta có viết `w *= wlen` để tính luỹ thừa của căn đơn vị. Việc nhân nhiều lần sẽ ảnh hưởng rất lớn đến độ chính xác của thuật toán vì ta đang thực hiện tính toán trên số thực.

Nhận xét rằng với mỗi len ta chỉ cần tính $\omega_{len}^0, \omega_{len}^1, \ldots, \omega_{len}^{len/2-1}$.

Định nghĩa một mảng $root$ sao cho với mỗi len và với mỗi $0 \le j < \frac{len}{2}$ ta có $root(\frac{len}{2} + j) = \omega_{len}^j$.

Cài đặt để tính mảng $root$:

```
vector<cd> root(n);
root[1] = 1;
for (int k = 2; k < n; k *= 2) {
```

```
        cd z = polar(1.0l, acos(-1.0l) / k * (invert ? 1 : -1));
        for (int j = k / 2; j < k; j++) {
            root[2 * j] = root[j];
            root[2 * j + 1] = root[j] * z;
        }
    }
```

Sau khi có mảng $root$ ta có thể sửa cài đặt FFT như sau:

```
for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k)
        for (int j = 0; j < k; j++) {
            cd z = root[j + k] * a[i + j + k];
            a[i + j + k] = a[i + j] - z;
            a[i + j] += z;
        }
```

Thử nghiệm với $n = 2^{20}$, sai số chỉ rơi vào khoảng $5.5511 \cdot 10^{-16}$.

FFT hai dãy cùng một lúc

Ta có thể tính DFT của hai dãy a và b cùng một lúc bằng cách tính DFT của dãy c với $c_j = a_j + b_j \cdot i$.

Gọi A, B, C lần lượt là các biến đổi DFT tương ứng. Ta có:

$$\overline{C(\omega^{-j})} = \overline{A(\omega^{-j})} + \overline{B(\omega^{-j}) \cdot i} = A(\overline{\omega^{-j}}) - i \cdot B(\overline{\omega^{-j}}) = A(\omega^j) - B(\omega^j) \cdot i$$

kết hợp với $C(\omega^j) = A(\omega^j) + B(\omega^j) \cdot i$ suy ra:

$$A(\omega^j) = \frac{C(\omega^j) + \overline{C(\omega^{-j})}}{2}$$
$$B(\omega^j) = \frac{C(\omega^j) - \overline{C(\omega^{-j})}}{2i}$$
$$A(\omega^j)B(\omega^j) = \frac{C^2(\omega^j) - \overline{C^2(\omega^{-j})}}{4i}$$

Sử dụng công thức này ta có thể tính tích chập chỉ dùng hai lần gọi hàm `fft` :

```
vector<int> conv(const vector<int> &a, const vector<int> &b) {
    if (a.empty() || b.empty()) return {};
    int n = 1;
    while (n < int(a.size() + b.size()) - 1) n <<= 1;
    vector<cd> in(n), out(n);
    for (int i = 0; i < int(a.size()); i++)
        in[i].real(a[i]);
    for (int i = 0; i < int(b.size()); i++)
        in[i].image(b[i]);
```

```
    fft(in, false);
    for (int i = 0; i < n; i++)
        in[i] *= in[i];
    for (int i = 0; i < n; i++) {
        /// (n - i) mod n
        int j = -i & (n - 1);
        out[i] = in[i] - conj(in[j]);
    }
    fft(out, true)
    vector<int> res(n);
    /// ở trên ta không chia cho 4i nên kết quả sẽ nằm trong ph
    for (int i = 0; i < n; i++)
        res[i] = int(imag(out[i]) / 4 + 0.5);
    return res;
}
```

3. Thuật toán NTT

Xét bài toán nhân đa thức nhưng lần này ta muốn các hệ số của đa thức chia lấy dư cho một số nguyên tố p. Nếu ta sử dụng thuật toán FFT thông thường có thể gây ra sai số lớn vì hệ số của đa thức kết quả có thể rất lớn. Thuật toán NTT cho phép ta tính toán chỉ dùng số nguyên, từ đó kết quả luôn đảm bảo chính xác.

Thuật toán FFT dựa trên các tính chất của căn đơn vị. Các tính chất này cũng xuất hiện trên căn đơn vị trong số học modulo. Cụ thể ta gọi căn đơn vị cấp n modulo p là một số nguyên ω_n thoả mãn:

$$(\omega_n)^n = 1 \pmod p$$
$$(\omega_n)^j \neq (\omega_n)^k, \forall\ 0 \leq j < k < n$$

Điều kiện thứ hai có thể được viết lại thành: $(\omega_n)^k \neq 1$ với mọi $1 \leq k < n$.

Các căn đơn vị cấp n khác được biểu diễn bằng một luỹ thừa của ω_n.

Để áp dụng thuật toán FFT, ta cần các căn đơn vị cho các luỹ thừa nhỏ hơn của 2. Ta có thể chứng minh tính chất sau:

$$(\omega_n^2)^{n/2} = 1 \pmod p$$
$$(\omega_n^2)^k \neq 1 \pmod p, \forall\ 1 \leq k < \frac{n}{2}$$

Từ đây ta có nếu ω_n là căn đơn vị cấp n, thì ω_n^2 là căn đơn vị cấp $\frac{n}{2}$ và do đó ta tính được căn đơn vị cho các luỹ thừa của 2 nhỏ hơn.

Để tính biến đổi ngược ta cần nghịch đảo modulo ω_n^{-1} tồn tại, điều này là hiển nhiên vì p là số nguyên tố.

Ta chứng minh được rằng với một số nguyên tố có dạng $p = c2^k + 1$, tồn tại $\omega_{2^k} = g^c$ với g là một căn nguyên thuỷ modulo p. Lấy ví dụ với $p = 998244353 = 119 \cdot 2^{23} + 1$ có căn nguyên thuỷ $g = 3$ và $\omega_{2^{23}} = 3^{119} \bmod p = 15311432$.

Cài đặt với $p = 998244353$ (các hằng số `root, root_pw` thể hiện căn đơn vị và số mũ tương ứng, `root_1` là nghịch đảo của căn đơn vị, hàm `inverse` tính nghịch đảo modulo của một số):

```
const int mod = 998244353;
const int root = 15311432;
const int root_1 = 469870224;
const int root_pw = 1 << 23;

void fft(vector<int> & a, bool invert) {
    int n = a.size(), L = __builtin_ctz(n);

    vector<int> rev(n);
    for (int i = 0; i < n; i++) {
        rev[i] = (rev[i >> 1] | (i & 1) << L) >> 1;
        if (i < rev[i]) swap(a[i], a[rev[i]]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        int wlen = invert ? root_1 : root;
        for (int i = len; i < root_pw; i <<= 1)
            wlen = (int)(1LL * wlen * wlen % mod);

        for (int i = 0; i < n; i += len) {
            int w = 1;
            for (int j = 0; j < len / 2; j++) {
                int u = a[i + j];
                int v = 1ll * a[i + j + len / 2] * w % mod;
                a[i + j] = u + v < mod ? u + v : u + v - mod;
                a[i + j + len / 2] = u - v >= 0 ? u - v : u - v
                w = 1ll * w * wlen % mod;
            }
        }
    }

    if (invert) {
        int n_1 = inverse(n, mod);
        for (int & x : a)
            x = 1ll * x * n_1 % mod;
    }
}
```

Nhân đa thức với modulo bất kì

Có thể thấy rằng thuật toán NTT chỉ hoạt động với nếu căn đơn vị tồn tại. Với các modulo không thoả mãn ta có hai cách sau:

Sử dụng định lý thặng dư Trung Hoa

Nếu kết quả của phép nhân đa thức có hệ số nhỏ hơn $M_1 \cdot M_2$, với M_1, M_2 là hai số nguyên tố có dạng $c2^k + 1$, ta có thể thực hiện NTT trên hai modulo này và dùng định lý thặng dư Trung Hoa để khôi phục kết quả.

Chia nhỏ đa thức

Ta cần tính $A(x) \cdot B(x)$ với hệ số modulo M, thực hiện tách hai đa thức như sau:

$$A(x) = A_0(x) + A_1(x) \cdot C B(x) = B_0(x) + B_1(x) \cdot C$$

với $C \approx \sqrt{M}$.

Khi đó tích của hai đa thức được biểu diễn thành:

$$A_0(x)B_0(x) + (A_0(x)B_1(x) + A_1(x)B_0(x)) \cdot C + A_1(x)B_1(x) \cdot C^2$$

Sử dụng kĩ thuật tương tự với FFT hai dãy cùng một lúc, ta có thể tính biểu thức trên với chỉ 4 lần gọi FFT.

Cài đặt:

```
using ll = long long;

vector<int> convMod(const vector<int> &a, const vector<int> &b,
    if (a.empty() || b.empty()) return {};
    vector<int> res(a.size() + b.size() - 1);
    int B = 32 - __builtin_clz(res.size());
    int n = 1 << B, cut = sqrt(M);

    vector<cd> L(n), R(n), outs(n), outl(n);

    for (int i = 0; i < int(a.size()); i++)
        L[i] = cd(a[i] / cut, a[i] % cut);
    for (int i = 0; i < int(b.size()); i++)
        R[i] = cd(b[i] / cut, b[i] % cut);
    fft(L, false); fft(R, false);

    for (int i = 0; i < n; i++) {
        /// j = (n - i) % n
        int j = -i & (n - 1);
        outl[i] = (L[i] + conj(L[j])) * R[i] / 2.0l;
        outs[i] = (L[i] - conj(L[j])) * R[i] / 2il;
    }
    fft(outl, true), fft(outs, true);

    for (int i = 0; i < int(res.size()); i++) {
        ll av = ll(real(outl[i]) + 0.5);
        ll cv = ll(imag(outs[i]) + 0.5);
        ll bv = ll(imag(outl[i]) + 0.5) + ll(real(outs[i]) + 0.
        res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
    }
```

```
    return res;
}
```

4. Bài tập tham khảo

- Codeforces - Nikita and Order Statistics
- AtCoder - Product Modulo
- CodeChef - Power Sum
- Kattis - A + B Problem
- Codeforces - Rusty String