Introduction to Polynomials

Peng Yuxiang

Tsinghua University
Department of Computer Science and Technology
Email : pickspeng@gmail.com

January 25, 2016

Contents

Polynomial Multiplication

Formal Power Series

Polynomial Algebra

Polynomial Factorization

Reference

Polynomial Multiplication

Formal Power Series

Polynomial Algebra

Polynomial Factorization

Reference

Definitions and Notations

- For polynomial $F(x) = f_0x^0 + f_1x^1 + ... + f_nx^n = \sum_{i=0}^{n} f_ix^i$
- Vector : $\mathbf{F} = (f_0, f_1, ..., f_n)^T$
- Degree : $\deg F = n$
- Domain : $f_i \in \mathcal{A}, F \in \mathcal{A}[x]$
- Monic polynomial : $f_n = 1$.

Definitions and Notations

- Addition and Subtraction : $(F \pm G)(x) = \sum_{i=0}^{n} (f_i \pm g_i) x^i$
- Multiplication : $(F \times G)(x) = \sum_{i=0}^{2n} (\sum_{j+k=i} f_j g_k) x^i$
- Power : $F^n(x) = \prod_{i=1}^{n} F(x)$

Naive Algorithm

- By definition :
- $(F \times G)[i] = \sum_{j+k=i} f_j g_k$

Karatsuba's Algorithm

- Assume $\deg F = n - 1$.
- Let $F(x) = F_0(x) + x^{\frac{n}{2}} F_1(x), G(x) = G_0(x) + x^{\frac{n}{2}} G_1(x)$, where $\deg F_0 = \deg F_1 = \deg G_0 = \deg G_1 = \frac{n}{2}$
- Naive Algorithm :
  $(F \times G)(x) = (F_0 \times G_0)(x) + x^{\frac{n}{2}}(F_0 \times G_1 + F_1 \times G_0)(x) + x^n(F_1 \times G_1)(x)$
- 4 subtasks with degree of $\frac{n}{2}$.
- Some tricks?

Karatsuba's Algorithm

- Naive Algorithm :
  $(F \times G)(x) = (F_0 \times G_0)(x) + x^{\frac{n}{2}}(F_0 \times G_1 + F_1 \times G_0)(x) + x^n(F_1 \times G_1)(x)$
- Let $M(x) = ((F_0 + F_1) \times (G_0 + G_1))(x)$
- Amazingly :
  $(F_0 \times G_1 + F_1 \times G_0)(x) = M(x) - (F_0 \times G_0)(x) - (F_1 \times G_1)(x)$
- 3 subtasks with degree $\frac{n}{2}$!
- $T(n) = 3T(\frac{n}{2}) + O(n)$.
- $T(n) = n^{\log_2 3} \approx n^{1.585}$.

Karatsuba's Algorithm

Pseudocode

Algorithm 1 Karatsuba's Algorithm

n $\leftarrow$ max(deg $F(x)$, deg $G(x)$).
if n = 0 **then**
  Multiply $F(x)$ and $G(x)$ naively.
else
  Get $F_0(x), F_1(x), G_0(x), G_1(x)$ by definition.
  Calculate $M(x) = ((F_0 + F_1) \times (G_0 + G_1))(x)$ recursively.
  Calculate $L(x) = (F_0 \times G_0)(x), R(x) = (F_1 \times G_1)(x)$ recursively.
  Return $L(x) + x^{\frac{n}{2}} M(x) + x^n R(x)$.
end if

Karatsuba's Algorithm

Example

- $F(x) = 1 + 2x + 3x^2 + 4x^3$
- $G(x) = 4 + 3x + 2x^2 + x^3$

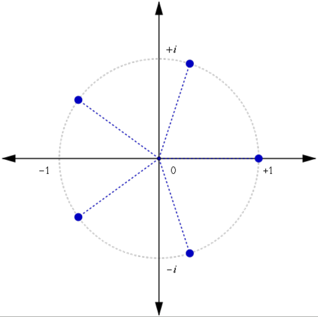
Fast Fourier Transform

- Another method to represent a polynomial :
- $\mathbf{F} = (F(x_1), F(x_2), ..., F(x_n))^T, \forall i \neq j, x_i \neq x_j$
- We can prove that it's the same as the coefficient representation.
- $F(x) = \sum_{i=1}^{n} \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)} F(x_i)$
- Advantage: $\mathbf{F} \times \mathbf{G} = (F(x_1)G(x_1), F(x_2)G(x_2), ..., F(x_n)G(x_n))$

Fast Fourier Transform

Root of unity

- The roots of $x^n = 1$
- $\omega_n^j = e^{2\pi i \frac{j}{n}} = \cos(2\pi \frac{j}{n}) + i \sin(2\pi \frac{j}{n})$
- $\omega_n^i = \omega_{\frac{n}{k}}^{\frac{i}{k}}$
- $\omega_n^i = \omega_n^{i \ mod \ n}$



Fast Fourier Transform

Discrete Fourier Transform (DFT)

- Consider calculating $\mathbf{F} = (F(\omega_n^0), F(\omega_n^1), ..., F(\omega_n^{n-1}))$
- Let $F_0(x) = \sum_{i=0}^{\frac{n}{2}} f_{2i}x^i, F_1(x) = \sum_{i=0}^{\frac{n}{2}} f_{2i+1}x^i$
- $F(x) = F_0(x^2) + xF_1(x^2)$
- $F(\omega_n^i) = F_0(\omega_n^{2i}) + \omega_n^i F_1(\omega_n^{2i}) = F_0(\omega_{\frac{n}{2}}^i) + \omega_n^i F_1(\omega_{\frac{n}{2}}^i)$
- $F(\omega_n^{i+\frac{n}{2}}) = F(-\omega_n^i) = F_0(\omega_{\frac{n}{2}}^i) - \omega_n^i F_1(\omega_{\frac{n}{2}}^i)$
- Notice : $\deg F_0 = \deg F_1 = \frac{n}{2}$
- Use recursion again.
- $T(n) = 2T(\frac{n}{2}) + O(n)$

Fast Fourier Transform

- Let's use matrix to represent the procedure.
- $$\begin{pmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \omega_n^2 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \omega_n^{2n-2} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} F(\omega_n^0) \\ F(\omega_n^1) \\ F(\omega_n^2) \\ \vdots \\ F(\omega_n^{n-1}) \end{pmatrix}$$

Fast Fourier Transform

Inversal and property of root of unity

- $$\begin{pmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \omega_n^2 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \omega_n^{2n-2} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix}^{-1} = \frac{1}{n} \begin{pmatrix} \omega_n^0 & \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^{-1} & \omega_n^{-2} & \cdots & \omega_n^{-n+1} \\ \omega_n^0 & \omega_n^{-2} & \omega_n^{-4} & \cdots & \omega_n^{-2n+2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \omega_n^0 & \omega_n^{-n+1} & \omega_n^{-2n+2} & \cdots & \omega_n^{-(n-1)(n-1)} \end{pmatrix}$$
- $\sum_{i=0}^{n-1} \omega_n^i = \frac{1-\omega_n^n}{1-\omega_n} = [n = 1]$

Fast Fourier Transform

Inverse Discrete Fourier Transform (IDFT)

- Therefore, $F[i] = \frac{1}{n} \sum_{j=0}^{n-1} F(\omega_n^j)\omega_n^{-ij}$
- When we have $D = \sum_{i=0}^{n-1} d_i x^i$, we want to get $\mathbf{D} = (D(\omega_n^{-0}), D(\omega_n^{-1}), ..., D(\omega_n^{-n+1}))$
- It's the same as DFT.
- Call the DFT algorithm with $\omega_n^i \to \omega_n^{-i}$.

Fast Fourier Transform

Pseudocode

Algorithm 2 Discrete Fourier Transform

n $\leftarrow \deg F$ and Enlarge n to a 2 power.

if n $= 1$ **then**

  Return F[0].

else

  Get $F_0(x), F_1(x)$ by definition.

  $\mathbf{F_0} =$ DFT($F_0(x)$), $\mathbf{F_1} =$ DFT($F_1(x)$).

  for i $\leftarrow 0$**to**$\frac{n}{2} - 1$ **do**

    F [i] $\leftarrow$ $\mathbf{F_0}$ [i] $+ \omega_n^i \mathbf{F_1}$ [i].

    F [i+$\frac{n}{2}$] $\leftarrow$ $\mathbf{F_0}$ [i] - $\omega_n^i \mathbf{F_1}$ [i].

  end for

end if

Cyclic Multiplication

- Indeed, DFT with degree n calculates $(F \times G)(x) = \sum_{j+k \equiv i \pmod{n}} f_j g_k x^i$
- When $n = p^k$, use similar divide and conquer algorithm.
- Time complexity is :
- $T(n) = pT(\frac{n}{p}) + O(pn)$
- $T(n) = O(pnk)$

Multivariable Polynomial Multiplication

Definitions and Idea

- $F(x_1, x_2, ..., x_d) = \sum_{i_1, i_2, ..., i_d} f_{i_1, i_2, ..., i_d} x_1^{i_1} x_2^{i_2} ... x_d^{i_d}$
- $(F \times G)(x_1, x_2, ..., x_d) = \sum_{i_1, i_2, ..., i_d} \sum_{j_1+k_1=i_1, ..., j_d+k_d=i_d} (f_{j_1, j_2, ..., j_d} g_{k_1, k_2, ..., k_d}) x_1^{i_1} x_2^{i_2} ... x_d^{i_d}$
- Expand the coefficients:
- $F(x, y) = \sum_{i=0}^{n} \sum_{j=0}^{m} f_{i,j} x^i y^j \to F(x) = \sum_{i=0}^{n} \sum_{j=0}^{m} f_{i,j} x^{i*m+j}$
- Then use the above algorithm.

Cantor's Algorithm

- Why does multiplication require root of unity or even division?
- Consider the multiplication in $\mathcal{A}[x]$.
- $\mathcal{A}$ contains $+$ with association, commutation, and $\times$ with distribution.
- $\alpha, \beta \in \mathcal{A}, k \in \mathbb{Z}$. Notice $k\alpha = \sum_{i=1}^{k} \alpha$ doesn't equal to $\alpha \times \beta$.

Cantor's Algorithm

Double DFT

- First solve the division. When $n = s^r$:
- $F(x) = \sum_{i=0}^{n-1} f_i x^i$
- $F^*(x) = \sum_{i=0}^{n-1} f_i \omega_{ns}^i x^i$
- When we calculate $C(x) = (A \times B)(x)$:
- Let $D(x) = n(A \times B)(x), E^*(x) = n(A^* \times B^*)(x)$ with cyclic multiplication of degree n, but we don't do the last division.
- Notice : $d_i = n(c_i + c_{n+i}), e_i = n(c_i + \omega_s c_{n+i})$.
- So $(1 - \omega_s) n c_i = e_i - \omega_s d_i, (1 - \omega_s) n c_{n+i} = d_i - e_i$.

Cantor's Algorithm

Double DFT

- Let $\tau_s = \prod_{1 \le i \le s, \gcd(i,s)=1} (1 - \omega_s^i)$
- $\tau_s = p$ if $s = p^k$ and p is a prime.
- $\tau_s n c_i = (e_i - \omega_s d_i) \times \prod_{2 \le i \le s, \gcd(i,s)=1} (1 - \omega_s^i)$
- $\tau_s n c_{n+i} = (d_i - e_i) \times \prod_{2 \le i \le s, \gcd(i,s)=1} (1 - \omega_s^i)$
- We choose two different s, such as 2 and 3, and let $n = s^r > \deg C$.
- So that, we can get $N_1 c_i$ and $N_2 c_i$, where
  $N_1 = \tau_{s_1} s_1^{r_1} \neq \tau_{s_2} s_2^{r_2} = N_2$.
- Employ Extended Euclidean Algorithm to find $M_1 N_1 - M_2 N_2 = 1$.
- Use doubling algorithm to calculate $M(Nc)$ and then we can get c_i without division.

Cantor's Algorithm

Cyclotomic integer and cyclotomic polynomial

- Let $\alpha = \sum_{i=0}^{\phi(n)-1} a_i \omega_n^i, a_i \in \mathcal{A}$, and $\alpha \in \mathbb{I}$.
- Notice : $\forall i \geq \phi(n), \omega_n^i$ can be linearly represented by $\omega_n^0, \omega_n^1, ..., \omega_n^{\phi(n)-1}$.
- Now, consider the multiplication of polynomial $A, B \in \mathbb{I}[x]$ whose degrees are less than n.
- Transform $\alpha = \sum_{i=0}^{\phi(n)-1} a_i \omega_n^i \leftrightarrow \sum_{i=0}^{\phi(n)-1} a_i y^i$

Cantor's Algorithm

Cyclotomic integer and cyclotomic polynomial

- Introduce the cyclotomic polynomial $\Phi_n(x) = \prod_{1 \le i \le n, \gcd(i,n)=1} (x - \omega_n^i)$
- We have $\Phi_{s^r}(x) = \Phi_s(x^{s^{r-1}})$ and $\Phi_n(x) \mid x^n - 1$.
- Meanwhile, multiplication of $\alpha, \beta \in \mathbb{I}$ is the same as the multiplication of the corresponding polynomials modulo $\Phi_n(y)$.
- Consider doing DFT to A, B with degrees ns.

Cantor's Algorithm

Cyclotomic integers DFT

- $F(\omega_n^i) = F_0(\omega_{\frac{n}{2}}^i) + \omega_n^i F_1(\omega_{\frac{n}{2}}^i)$
- $F(\omega_n^{i+\frac{n}{2}}) = F_0(\omega_{\frac{n}{2}}^i) - \omega_n^i F_1(\omega_{\frac{n}{2}}^i)$
- In order to do DFT modulo $\Phi_n(x) \mid x^n - 1$, we first do DFT modulo $x^n - 1$.
- In FFT, there are two sorts of operations:
- Addition/Subtraction : deal with them naively.
- Multiplication with ω_n^k : for $\omega_n \leftrightarrow x$, just shift the coefficients of the cyclotomic integer.
- The time complexity is : $O(sn^2r)$.
- Then we reduce the answer to $\Phi_n(x)$ naively. We can prove that complexity won't change.

Cantor's Algorithm

- Go back to polynomial multiplication.
- Let $m = s^r$ so that $\phi(m) \geq n$, and $p = s^u, q = s^v$ so that $u + v = r$ and $v + 1 \leq u \leq v + 2$.
- The following transform reveals the equivalence between polynomial and cyclotomic integer :
- $A(x) = \sum_{i=0}^{\phi(m)-1} a_i x^i \leftrightarrow \sum_{i=0}^{\phi(m)-1} a_i \omega_m^i$
- Fold up the coefficients $A(x) = \sum_{j=0}^{q-1} (\sum_{i=0}^{\phi(p)-1} a_{iq+j} x^{iq}) x^j$.
- With the equivalence, $A(x) \leftrightarrow \sum_{j=0}^{q-1} (\sum_{i=0}^{\phi(p)-1} a_{iq+j} \omega_p^i) x^j$.
- Now, doing DFT to $A(x), B(x)$ is possible according to the above algorithm.

Cantor's Algorithm

- After doing DFT, we need to multiply several pairs of new cyclotomic integers.
- Call the above algorithm recursively.
- Notice that $p, q \in O(\sqrt{m})$, and $m \in O(\sqrt{n})$.
- $T(n) = pT(q) + O(pq \log q) = \sqrt{n}T(\sqrt{n}) + O(n \log n)$.
- $T(n) = O(n \log n \log \log n)$.

Polynomial Multiplication

Formal Power Series

Polynomial Algebra

Polynomial Factorization

Reference

Definitions and Notations

- For formal power series $F(x) = \sum_{i=0}^{\infty} f_i x^i$:
- Formal derivative : $F'(x) = \sum_{i=0}^{\infty}(i+1)f_{i+1}x^i$
- Formal integral : $\int F(x)dx = \sum_{i=1}^{\infty} \frac{f_{i-1}}{i} x^i + C$
- Addition and Substraction : $(F \pm G)(x) = \sum_{i=0}^{\infty}(f_i \pm g_i)x^i$
- Multiplication : $(F \times G)(x) = \sum_{i=0}^{\infty}(\sum_{j+k=i} f_j g_k)x^i$
- Modulo x^n : $F(x) \equiv F(x) \text{ rem } x^n \equiv \sum_{i=0}^{n-1} f_i x^i \pmod{x^n}$

Formal Power Series Equation

- Composition: If $G(x)$ rem $x = 0$,
- $(F \circ G)(x) = F(G(x)) = \sum_{i=0}^{\infty} f_i G^i(x)$
- Equation: Find $X(x)$, so that $(F \circ X)(x) = 0$.
- Output $X(x)$ rem x^n.

Newton Iteration

Taylor expansion

- We try to expand power series $F(X(x))$ at point $G(x)$ with $\deg G = t$.
- $$(F \circ X)(x) = (F \circ G)(x) + \frac{(F' \circ G)(x)}{1!}(X - G)(x) + \frac{(F'' \circ G)(x)}{2!}(X - G)^2(x) + \ldots$$

Newton Iteration

Iteration

- Let $X_i(x) = X(x) \text{ rem } x^{2^i}$. Assume we'd got $X_t(x)$.
- Insert it into the Taylor expansion :
- $F \circ X_{t+1} = F \circ X_t + \frac{F' \circ G}{1!}(X_{t+1} - X_t) + \frac{F'' \circ G}{2!}(X_{t+1} - X_t)^2 + \ldots$
- $F \circ X_t + (F' \circ X_t)(X_{t+1} - X_t) \equiv 0 \pmod{x^{2^{t+1}}}$
- $X_{t+1} = X_t - \frac{F \circ X_t}{F' \circ X_t} \text{ rem } x^{2^{t+1}}$

Newton Iteration

Inversion

- Let $F(x) = G(y)x - 1$. $F'(x) = G(y)$.
- Insert it into the above formula :
- $X_{t+1} = 2X_t - G \times X_t^2 \pmod{x^{2^{t+1}}}$
- $X_0 = G[0]^{-1}$
- $T(n) = T(\frac{n}{2}) + O(n \log n)$
- $T(n) = O(n \log n)$.

Newton Iteration

Iteration

- $X_{t+1} = X_t - \frac{F \circ X_t}{F' \circ X_t} \text{ rem } x^{2^{t+1}}$
- We find that $X_{t+1}(x) \text{ rem } x^{2^t} = X_t(x)$.
- So that $(F' \circ X_{t+1})^{-1} \text{ rem } x^{2^t} = (F' \circ X_t)^{-1} \text{ rem } x^{2^t}$.
- We can maintain both terms at the same time.
- After we solve the inversion, bottleneck is the power series composition.

Polynomial Elementary Function

- Logarithm : Let $X = \ln F$, so that $X = \int \frac{F'}{F} dx$.
- Exponent: Let $F(x) = \ln x - G(y)$, so that $F'(x) = \frac{1}{x}$.
- Solve the equation $F \circ X = 0$:
- $X_{t+1} \equiv X_t(1 - \ln X_t + G) \pmod{x^{2^{t+1}}}$
- Then $X(x) = e^{G(x)}$.
- Meanwhile $e^{iG(x)} = \cos(G(x)) + i\sin(G(x))$.
- Power : Let $X = G^k$, so that $\ln x = \frac{1}{k}\ln G$.
- All the elementary functions of polynomial can be calculated in $O(n \log n)$.

Polynomial Modular Composition

Brent's and Kung's Algorithm

- Calculate $(Q \circ P)(x)$ rem x^n.
- Let $P(x) = P_m(x) + P_r(x)$, where $P_m(x) = \sum_{i=0}^{m-1} p_i \times x^i$, $l = \lceil \frac{n}{m} \rceil$.
- Taylor expansion:
- $$Q \circ P \equiv Q \circ P_m + (Q' \circ P_m) \times P_r + \frac{1}{2}(Q'' \circ P_m) \times P_r^2 + \dots \\ + \frac{1}{l!}(Q^{(l)} \circ P_m) \times P_r^l(x) \pmod{x^n}$$
- Let $Q_0(x) = \sum_{i=0}^{\frac{n}{2}-1} q_i x^i, Q_1(x) = \sum_{i=0}^{\frac{n}{2}-1} q_{i+\frac{n}{2}} x^i$
- So $Q \circ P_m = Q_0(P_m) + P_m^{\frac{n}{2}} \times Q_1(P_m)$
- This step: $T(u) \leq 2T(\frac{u}{2}) + O(\min\{u \times m, n\} \log n)$
- $T(n) \leq O(mn \log^2 n)$

Polynomial Modular Composition

Brent's and Kung's Algorithm

- Chain Law : $(Q^{(i)}(P_m))' = Q^{(i+1)}(P_m) \times P'_m$
- So $Q^{(i+1)}(P_m) = \frac{(Q^{(i)}(P_m))'}{P_m}$
- This step : $O(\frac{n}{m} n \log n)$.
- Let $m = \sqrt{n \log n}$.
- Complexity: $O((n \log n)^{1.5})$.

Polynomial Modular Composition

Bernstein's Algorithm

- When $G \in \mathbb{F}_p[x]$, $G^p(x) = \sum_{i=0}^{\infty} g_i^p x^{ip}$.
- Let $Q_i(x) = \sum_{j=0}^{\infty} q_{jp+i} x^j$
- $P^p(x) = \sum_{i=0}^{\infty} p_i^p x^{ip}$.
- So $Q \circ P \equiv \sum_{i=0}^{p} P^i Q_i(P^p) \pmod{x^n}$.
- Note that only $Q_i(x)$ rem $x^{\frac{n}{p}}$ is helpful.
- Use recursion to calculate $Q_i(P^p)$.
- $T(n) = pT(\frac{n}{p}) + O(pn \log n)$
- $T(n) = O(pn \log^2 n)$

Polynomial Multiplication

Formal Power Series

Polynomial Algebra

Polynomial Factorization

Reference

Polynomial Division

- Given $A(x), B(x), \deg A = n, 1 \le \deg B = m < n$
- Find proper polynomial $Q(x), R(x)$, so that :
- $A(x) = (B \times Q)(x) + R(x)$, where $\deg R < \deg B$.

Polynomial Division

Reversal

- For $F(x) = \sum_{i=0}^{n} f_i x^i$, Let $F^R(x) = x^n F(\frac{1}{x}) = \sum_{i=0}^{n} f_{n-i} x^i$.
- Example : $P(x) = 1 + 2x + 3x^2 + 4x^3, P^R(x) = 4 + 3x + 2x^2 + x^3$.
- $$\begin{aligned} A^R(x) &= x^n A(\tfrac{1}{x}) = x^n((B \times Q)(\tfrac{1}{x}) + R(\tfrac{1}{x})) \\ &= (B \times Q)^R(x) + x^{n-m} R^R(x). \end{aligned}$$
- $A^R(x) \equiv (B \times Q)^R(x) \pmod{x^{n-m}}$
- $Q^R(x) \equiv \frac{A^R(x)}{B^R(x)} \pmod{x^{n-m}}$

Polynomial Division

- $Q(x) = (Q^R)^R(x)$
- $R(x) = A(x) - (B \times Q)(x)$
- Time Complexity: $O(n \log n)$.

Multiplication with Remainder

- Consider polynomial multiplication modulo $P(x)$.
- Use polynomial division to get the remainder.
- $A(x) = (B \times Q)(x) + R(x)$
- Define : $A(x)$ quo $B(x) = Q(x), A(x)$ rem $B(x) = R(x)$.

Constant Coefficients Linear Recursion

Transform to matrix multiplication

- Recursion : $f_n = \sum_{i=1}^{d} a_i f_{n-i}$. Given $\forall 1 \le i \le d, a_i, f_{i-1}$.
- Construct a matrix :

$$\mathbf{M} = \begin{pmatrix} a_1 & a_2 & \cdots & a_{d-1} & a_d \\ 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \end{pmatrix}$$

- We can get $\mathbf{F}_n = \mathbf{M}^{n-d+1}\mathbf{F}_{d-1}$, where $\mathbf{F}_i = (f_i, f_{i-1}, \cdots, f_{i-d+1})^T$.

Constant Coefficients Linear Recursion

Cayley-Hamilton Theorem

- For any matrix $\mathbf{M}$, we define the characteristic polynomial $f_{\mathbf{M}}(\lambda) = \det(\lambda\mathbf{I} - \mathbf{M})$.
- Cayley-Hamilton Theorem : $f(\mathbf{M}) = \mathbf{0}$.
- In our case, $f_{\mathbf{M}} = \lambda^d + \sum_{i=1}^{d} a_i \lambda^{d-i}$.
- So, $\mathbf{M}^d + \sum_{i=1}^{d} a_i \mathbf{M}^{d-i} = \mathbf{0}$.

Constant Coefficients Linear Recursion

Polynomial module

- $\mathbf{M}^d + \sum_{i=1}^{d} a_i \mathbf{M}^{d-i} = \mathbf{0}$
- $x^d + \sum_{i=1}^{d} a_i x^{d-i} = 0$
- $x^{n-d+1} = x^{n-d+1} \text{ rem } (x^d + \sum_{i=1}^{d} a_i x^{d-i})$
- So $\mathbf{M}^{n-d+1} = \sum_{i=0}^{d-1} c_i \mathbf{M}^i$.
- Use repeated square algorithm to calculate c_i.
- $\mathbf{F}_n = \mathbf{M}^{n-d+1}\mathbf{F}_0 = \sum_{i=0}^{d-1} c_i \mathbf{M}^i \mathbf{F}_0$.
- $f_n = \sum_{i=0}^{d-1} c_i f_i$.

Modular Composition

- Consider doing polynomial composition modulo $P(x)$.
- $Q \circ P \equiv Q \circ P_m + (Q' \circ P_m) \times P_r + \frac{1}{2}(Q'' \circ P_m) \times P_r^2 + ...$
  $$+\frac{1}{l!}(Q^{(l)} \circ P_m) \times P_r^l(x) + ... \pmod{x^n}$$
- Taylor expansion doesn't work here.
- There is another method come up with by Brent and Kung based on matrix multiplication.

Modular Composition

Brent's and Kung's algorithm

- Think about the problem using grouping.
- $Q_i(x) = \sum_{j=0}^{k-1} q_{ik+j} x^j$, where $k = \lceil \sqrt{n+1} \rceil$.
- $Q \circ P = \sum_{j=0}^{k-1} P^{kj} Q_j(P)$
- First, we calculate $P^i(x)$ for $i \leq k$ naively using FFT.
- Second, we calculate $Q_j \circ P$ for $j \leq k$ naively using P^i.
- At last we calculate P^{kj} for $j \leq k$ naively using FFT.
- As a matter of fact, the second step is a matrix multiplication, which costs $\sqrt{n}^{\omega}$.
- The best known algorithm of matrix multiplication gives $\omega \approx 2.3727$.
- The total complexity is $O(n^{1.5} \log n + n^{\frac{1+\omega}{2}}) = O(n^{1.687})$.

Multipoint Evaluation

- Given $F(x), \mathbf{x} = (x_1, x_2, ..., x_n)^T$.
- Calculate $\mathbf{F} = (F(x_1), F(x_2), ..., F(x_n))$.
- Construct $L(x) = \prod_{i=1}^{\frac{n}{2}}(x - x_i), R(x) = \prod_{i=\frac{n}{2}+1}^{n}(x - x_i)$.
- Use divide and conquer to expand them. As a pre-treatment, all $L(x), R(x)$ used in the calculation can be calculated in $O(n \log^2 n)$.

Multipoint Evaluation

- Let $P_0(x) = F(x) \text{ rem } L(x), P_1(x) = F(x) \text{ rem } R(x)$.
- $\forall i, 1 \leq i \leq \frac{n}{2}, F(x_i) = P_0(x_i)$.
- $\forall i, \frac{n}{2} \leq i \leq n, F(x_i) = P_1(x_i)$.
- We find that they are the same problems, and solve them recursively.
- $T(n) = 2T(\frac{n}{2}) + O(n \log n)$.
- $T(n) = O(n \log^2 n)$.

Linear Combination

- Given $m_1(x), m_2(x), ..., m_r(x)$, with $n = \sum_{i=1}^{r} \deg m_i$, and $c_1(x), c_2(x), ..., c_r(x)$ with $\deg c_i \le \deg m_i$.
- Let $M(x) = \prod_{i=1}^{r} m_i$.
- Calculate $\sum_{i=1}^{r} c_i \frac{M}{m_i}$.
- Choose k, so that $\sum_{i=1}^{k} \deg m_i \le \frac{n}{2}$ and $\sum_{i=1}^{k+1} \deg m_i > \frac{n}{2}$.
- Let $L(x) = \prod_{i=1}^{k} m_i, R(x) = \prod_{i=k+1}^{r} m_i$.
- $F(x) = \sum_{i=1}^{r} c_i \frac{M}{m_i} = (\sum_{i=1}^{k} c_i \frac{L}{m_i})R + (\sum_{i=k+1}^{r} c_i \frac{R}{m_i})L$.
- Calculate $\sum_{i=1}^{k} c_i \frac{L}{m_i}$ and $\sum_{i=k+1}^{r} c_i \frac{R}{m_i}$ recursively.
- $T(n) = O(n \log^2 n)$.

Multipoint Interpolation

- Recall the Lagrange Interpolation:
- $F(x) = \sum_{i=1}^{n} \frac{\prod_{j\neq i}(x-x_j)}{\prod_{j\neq i}(x_i-x_j)} F(x_i)$
- The i-th numerator : $P_i(x) = \frac{M(x)}{x-x_i}$, where $M(x) = \prod_{i=1}^{n}(x - x_i)$.
- The i-th denominator : $Q_i = P_i(x_i)$.
- Use formal derivative : $M'(x) = \sum_{i=1}^{n} \frac{M(x)}{x-x_i} = \sum_{i=1}^{n} P_i(x)$.
- Notice that $\forall j \neq i, P_i(x_j) = 0$, so $Q_i = P_i(x_i) = M'(x_i)$.
- Call the multipoint evaluation to get denominators.
- Call the linear combination to calculate :
- $F(x) = \sum_{i=1}^{n} \frac{F(x_i)}{Q_i} \frac{M(x)}{x-x_i}$
- $T(n) = O(n \log^2 n)$.

Polynomial Euclidean Algorithm

- The aim is to find a monic polynomial dividing r_0, r_1.
- Traditionally, the recursion is : $r_{i-2}(x) = (r_{i-1} \times q_{i-1})(x) + r_i(x)$.
- Observe that degree of every quotient is small. Time is wasted at the calculation of polynomial division.
- Another observation is that quotients only depend on the head terms of $r(x)$.

Polynomial Euclidean Algorithm

Example

- $r_0 = 5 + 4x + 3x^2 + 2x^3 + x^4, r_1 = 1 + x + x^2 + x^3$
- $r_0 = r_1 \times q_1 + r_2$
- $q_1 = 1 + x, r_2 = 3 + 2x + x^2.$
- We find $\deg q_1 = 1$, and q_1 only depends on head terms of r_0 and r_1.

Polynomial Euclidean Algorithm

- Define $F(x)$ trc $k = F(x)$ quo x^k.
- Use divide and conquer. Consider calculating r_{k-1}, r_k.
- If we'd got $r_{\frac{k}{2}-1}, r_{\frac{k}{2}}$, calculate the rest recursively using $r_{\frac{k}{2}-1}, r_{\frac{k}{2}}$.
- An observation is that $r_{\frac{k}{2}-1}, r_{\frac{k}{2}}$ can be calculated by r_0 trc k, r_1 trc k.
- So we can calculate $r_{\frac{k}{2}-1}, r_{\frac{n}{2}}$ using the algorithm recursively under truncation k.
- $T(n) = O(n \log^2 n)$.
- You can easily modify this algorithm to polynomial extended Euclidean algorithm in order to calculate the Bézout coefficients.

Polynomial Chinese Remainder Algorithm

- Given r co-prime polynomials $m_1(x), m_2(x), ..., m_r(x)$, and $n = \sum_{i=1}^{r} \deg m_i$, and $a_1(x), a_2(x), ..., a_r(x)$.
- Find a polynomial $F(x)$, so that $F \equiv a_i \pmod{m_i}$.
- Recall the classical CRT :
- $M = \prod_{i=1}^{r} m_i$.
- $F = \sum_{i=1}^{r} a_i [(\frac{M}{m_i})^{-1}]_{m_i} \frac{M}{m_i}$.
- Imitate this.

Polynomial Chinese Reminder Algorithm

Simultaneous Reduction

- Given $F(x)$ and r co-prime polynomials $m_1(x), m_2(x), ..., m_r(x)$, with $\sum_{i=1}^{r} \deg m_i = n$.
- Calculate F rem m_1, F rem $m_2, ..., F$ rem m_r.
- Choose k, so that $\sum_{i=1}^{k} \deg m_i \leq \frac{n}{2}$ and $\sum_{i=1}^{k+1} \deg m_i > \frac{n}{2}$.
- Calculate F rem $\prod_{i=1}^{k} m_i$ and F rem $\prod_{i=k+1}^{r} m_i$.
- Calculate the reminders recursively.
- $T(n) = O(n \log n \log r)$.

Polynomial Chinese Reminder Algorithm

Simultaneous Inversion

- Given $m_1(x), ..., m_r(x)$, $M(x) = \prod_{i=1}^{r} m_i(x)$.
- Calculate all $s_i(x) = [(\frac{M(x)}{m_i(x)})^{-1}]_{m_i(x)}$.
- Call simultaneous reduction to calculate $g_i(x) = M(x) \text{ rem } m_i^2(x)$.
- Notice $m_i \mid g_i$, so $\frac{M(x)}{m_i(x)} \text{ rem } m_i(x) = \frac{g_i(x)}{m_i(x)}$.
- Call polynomial Euclidean algorithm to get all $s_i(x)$ separately.
- $T(n) = O(n \log n \log r)$.

Polynomial Chinese Reminder Algorithm

- Go back to CRT.
- $F = \sum_{i=1}^{r} a_i [(\frac{M}{m_i})^{-1}]_{m_i} \frac{M}{m_i}$.
- We've got $[(\frac{M}{m_i})^{-1}]_{m_i}$ and a_i.
- Call linear combination to get $F(x)$.
- $T(n) = O(n \log n \log r)$.

Polynomial Multiplication

Formal Power Series

Polynomial Algebra

Polynomial Factorization

Reference

Notations

- Finite field with size p : $\mathbb{F}_p$
- Example : Integers modulo p.
- Polynomials with coefficients over $\mathbb{F}_p$: $G(x) \in \mathbb{F}_p[x]$.
- In this section, we just consider polynomials over $\mathbb{F}_p[x]$.
- Reducible polynomial $F(x)$:
  $\exists A, B \in \mathbb{F}_p[x], \quad \deg A, \deg B > 0, \quad F = A \times B.$

Noname Polynomial

- Noname theorem : over $\mathbb{F}_p$, $x^{p^n} - x$ is the product of all irreducible polynomials with degrees dividing n.
- Special Case : $x^p - x = \prod_{\alpha \in \mathbb{F}_p} (x - \alpha)$.
- Example : over $\mathbb{F}_2$, $x^{2^3} - x$ is the product of all irreducible polynomials whose degrees divide 3.

Irreducibility Test

Ben-Or's algorithm

- Naively, using the noname theorem, we can try to enumerate the factors.
- When $n = \deg F$, enumerate i from 1 to $\frac{n}{2}$.
- Calculate $\gcd(x^{p^i} - x, F) = \gcd((x^{p^i} - x) \text{ rem } F, F)$.
- Repeatedly use repeated squaring algorithm : $x^{p^{i+1}} = (x^{p^i})^p$.
- Time complexity : $O(n^2 \log n \log p)$.

Irreducibility Test

Improved Ben-Or's algorithm

- If $F(x)$ is irreducible, $F(x) \mid x^{p^n} - x$.
- But $F(x)$ may be product of some irreducible polynomials with degree dividing n.
- Additional test : $\forall t \mid n, \quad \gcd(x^{p^{\frac{n}{t}}} - x, F) = 1$.
- In order to accelerate the calculation of x^{p^i}, let $P_i(x) = x^{p^i}$.

Irreducibility Test

Improved Ben-Or's algorithm

- $P_{i+j}(x) = x^{p^{i+j}} = x^{p^i p^j} = (x^{p^i})^{p^j} = (P_i \circ P_j)(x)$.
- Calculate $P_1(x) = x^p$ by repeated squaring as initialization.
- Use modular composition like repeatedly doubling algorithm.
- Complexity to calculate $P_m(x)$ rem $F(x)$: $O(n^{1.687} \log m)$.
- Let $\delta(n) = \sum_{p|n, p \text{ } is \text{ } prime} 1$, we have $\delta(n) \leq \frac{\ln n}{\ln \ln n}$.
- Total complexity : $O(n^{1.687} \frac{\log^2 n}{\log \log n} + n \log n \log p)$.

Polynomial Factorization

Outline

- Given a polynomial over $\mathbb{F}_p[x]$, we'd like to factor it.
- First, we try to find all the irreducible factors of $F(x)$.
- We can easily factor $F(x)$ using polynomial division with the irreducible factors.
- Enumerate $1 \leq i \leq \frac{n}{2}$ as usual. (This step is called distinct-degree factorization, which is the complexity bottleneck)

Polynomial Factorization

Outline

- At the same time, we reduce $F(x)$ with factors found.
- So $g_i(x) = \gcd(x^{p^i} - x, F(x))$.
- Represent $g_i(x) = \prod_j s_j(x)$, where $s_j(x)$ is irreducible polynomial with degree i.
- The next task is factoring $g_i(x)$. (This step is called equal-degree factorization)

Polynomial Factorization

Some theorems

- For a finite field $\mathbb{F}_p[x]$, p is a prime power c^k.
- We name c as the characteristics of $\mathbb{F}_p[x]$.
- When p is odd :
  $\forall F(x), P(x) \in \mathbb{F}_p[x]$ and $\gcd(F(x), P(x)) = 1$,
  $F^{\frac{p^{\deg P}-1}{2}}(x) \equiv \pm 1 \pmod{P(x)}$.
- Meanwhile, $+1$ and -1 are uniformly distributed.

Polynomial Factorization

Equal-degree factorization over $\mathbb{F}_p[x]$ with odd characteristics

- Given a polynomial $g(x) = \prod_i s_i(x)$ with degree n, where s_i is irreducible polynomial with degree d, and $g(x)$ is reducible.
- Consider a random polynomial $l(x) \in \mathbb{F}_p[x]$.
- According to the theorem above : $l^{\frac{p^d-1}{2}}(x) \equiv \pm 1 \pmod{s_i(x)}$.
- ± 1 are uniformly distributed.
- When $l^{\frac{p^d-1}{2}}(x) \equiv 1 \pmod{s_i(x)}$, we have $s_i(x) \mid l^{\frac{p^d-1}{2}}(x) - 1$.
- Calculate $r(x) = \gcd(l^{\frac{p^d-1}{2}}(x) - 1, g(x))$.
- If $r(x) \neq 1$ and $r(x) \neq g(x)$, we've found a factor of $g(x)$.
- Else, repeat it. We claim success probability is greater than $\frac{1}{2}$.
- Call the above algorithm recursively to get totally factorization.

Polynomial Factorization

Equal-degree factorization over $\mathbb{F}_p[x]$ with odd characteristics

- In expectation, the whole depth is $O(\log \frac{n}{d})$.
- So the expected time complexity of this step is : $O(dn \log n \log p \log \frac{n}{d})$.
- At the same time, $d \log \frac{n}{d} \leq n$.
- Expected time complexity is less than $O(n^2 \log n \log p)$.

Polynomial Factorization

Equal-degree factorization over $\mathbb{F}_p[x]$ with even characteristics

- When $p = 2^k$, that theorem doesn't hold. We'd like to find another transform to get $X(x) \equiv \pm 1 \pmod{s_i(x)}$ with uniformly distribution.
- Let $T(x) = \sum_{i=0}^{kd-1} x^{2^i}$.
- A theorem says :
  When we choose $l(x) \in \mathbb{F}_p[x]$ uniformly,
  $(T \circ l)(x) \equiv \pm 1 \pmod{s_i(x)}$ and ± 1 are uniformly distributed.
- Simply substitute $(T \circ l)(x)$ for $l^{\frac{p^d-1}{2}}(x)$.
- Time complexity doesn't change.

Polynomial Factorization

Overlook

- The distinct-degree factorization needs $O(n^2 \log n \log p)$.
- The equal-degree factorization needs no more than expected $O(n^2 \log n \log p)$.
- In practice, the equal-degree factorization needs much less time than the worst condition.
- So the whole complexity is $O(n^2 \log n \log p)$.
- In 2008, Umans and Kedlaya gave an algorithm to solve distinct-degree factorization in $O(n^{1.5+o(1)} + n^{1+o(1)} \log p) \log^{1+o(1)} p)$.

Polynomial Multiplication

Formal Power Series

Polynomial Algebra

Polynomial Factorization

Reference

Reference

- J. von zur Gathen and J. Gerhard : Modern Computer Algebra(3rd Edition), 2013.
- D. G. Cantor and E. Kaltofen : On Fast Multiplication of Polynomials over Arbitrary Algebra, 1991.
- R. P. Brent and H. T. Kung : Fast Algorithms for Manipulating Formal Power Series, 1978.
- D. J. Bernstein : Composing Power Series over a Finite Field in Essentially Linear Time, 1991.

Reference

- W. Keller-Gehrig : Fast Algorithms for the Characteristic Polynomial, 1984.
- D. G. Cantor and H. Zassenhaus : A New Algorithm for Factoring Polynomials over Finite Fields, 1981.
- C. Umans : Fast Polynomial Factorization, Modular Composition, and Multipoint Evaluation of Multivariate Polynomials in Small Characteristic, 2007.
- K. S. Kedlaya and C. Umans : Fast Modular Composition in Any Characteristic, 2008.