Computer Algebra 2

October 31, 2018

1 Notations and conventions

Unless otherwise mentioned, we use the following notations:

- k, K, $\mathbb{K}$ are (commutative) fields
- R is a (commutative, with 1) ring

Given a ring R, R^* is the group of its invertible elements.

We assume that algebraic computations (sum, inverse, test of 0, test of 1, inverse where applicable) can be performed.

For a vector v in a vector space V of dimension n, we denote its coordinates by $(v_0, \dots, v_{n-1})$. If f is a polynomial of degree $\deg(f) = d$, its coefficients are denoted $f_0, \dots, f_d$, such that

$$f(X) = f_0 + f_1 X + \cdots + f_d X^d = \sum_{i=0}^{d} f_i X^i.$$

In order to simplify notations, we may at times use the convention that $f_i = 0$ if $i < 0$ or $i > \deg(f)$, so that

$$f = \sum_{i \in \mathbb{Z}} f_i X^i.$$

By convention, the degree of the 0 polynomial is $-\infty$.

The logarithm log, without a base, is in base 2.

Definition 1.1. Given two functions $f, g : \mathbb{N} \to \mathbb{R}_{>0}$

$$\begin{aligned} f = O(g) &\iff \frac{f(n)}{g(n)} \text{ is bounded when } n \to \infty \\ &\iff \exists c \in \mathbb{R}_{>0}, n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \leq c g(n); \\ f = \tilde{O}(g) &\iff \exists l \in \mathbb{N}, f = O(g \log(g)^l). \end{aligned}$$

1.1 Exercises

Exercise 1.1. Show that the "when $n \to \infty$" clause in the definition of O can be left out. In other words, given $f, g : \mathbb{N} \to \mathbb{R}_{>0}$, show that

$$\begin{aligned} f = O(g) &\iff \frac{f(n)}{g(n)} \text{ is bounded} \\ &\iff \exists c \in \mathbb{R}_{>0}, \forall n \in \mathbb{N}, f(n) \leq cg(n) \end{aligned}$$

2 Semi-fast multiplication

In this chapter, let R be any ring.

Given $f, g \in R[X]$ with degree less than n, we want to compute the coefficients of $h = f \cdot g$.

The complexity of the algorithm will be evaluated in number of multiplications and additions in R. Typically, multiplications are more expensive!

2.1 Naive algorithm

Each coefficient h_k $(0 \le k < 2n)$ can be computed with

$$h_k = \sum_{i=0}^{k} f_i g_{k-i},$$

each costing $O(n)$ multiplications and additions.

The total complexity of the naive algorithm is $O(n^2)$ multiplications and $O(n^2)$ additions.

2.2 Karatsuba's algorithm

Remark 2.1. Linear polynomials can be multiplied using 3 multiplications instead of 4 :

$$(a + bX)(c + dX) = ac + (ad + bc)X + bdX^2$$

with

$$ad + bc = ad + bc + ac + bd - ac - bd = (a + b)(c + d) - ac - bd.$$

This can be used recursively to compute polynomial multiplication faster.

Algorithm 1 Karatsuba

Input: $f = f_0 + \cdots + f_{n-1}X^{n-1}$, $g = g_0 + \cdots + g_{n-1}X^{n-1}$

Output: $h = h_0 + \cdots + h_{2n-1}X^{2n-1}$ such that $h = fg$

1. If $n = 1$, then return $f_0 g_0$
2. Write $f = A + BX^{\lceil n/2 \rceil}$, $g = C + DX^{\lceil n/2 \rceil}$ where all of A, B, C, D have degree $< \lceil \frac{n}{2} \rceil$.
3. Compute recursively:
   - $P = AC$
   - $Q = BD$
   - $R = (A + B)(C + D)$
4. Return $P + (R - P - Q)X^{\lceil n/2 \rceil} + RX^{2\lceil n/2 \rceil}$

Theorem 2.2. *Karatsuba's algorithm multiplies polynomials with $O(n^{\log_2(3)}) = O(n^{1.585})$ multiplications and additions.*

Proof. Let $M(n)$ (resp. $A(n)$) be the number of multiplications (resp. additions) in a run of Algo. 1 on an input with size n. Then:

$$M(n) = 3M(n/2)$$

and

$$A(n) = 3A(n/2) + O(n)$$

so $M(n) = O(n^{\log_2(3)})$ and $A(n) = O(n^{\log_2(3)})$. □

Remark 2.3. Karatsuba's algorithm hides an evaluation/interpolation mechanism:

$$\begin{aligned} a &= (a + bX)_{X=0} \\ a + b &= (a + bX)_{X=1} \\ b &= \left(\frac{a + bX}{X}\right)_{X=\infty} \end{aligned}$$

and for two linear polynomials f, g, if $fg = h = h_0 + h_1X + h_2X^2$, we have

$$\begin{aligned} &f(0)g(0) = h(0) = h_0 \\ &f(1)g(1) = h(X = 1) = h_0 + h_1 + h_2 \\ &\left(\frac{f}{X}\right)_{X=\infty} \left(\frac{g}{X}\right)_{X=\infty} = \left(\frac{h}{X^2}\right)_{X=\infty} = h_2 \end{aligned}$$

2.3 Toom-k algorithm

For the remainder of this section, assume that the ring R is an infinite field.

In general the coefficients of h can be obtained as a linear combination of $f(i)g(i)$ for $i \in \{0, \ldots, 2n-1\}$ via

$$\begin{pmatrix} h_0 \\ h_1 \\ h_2 \\ \vdots \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \ldots \\ 1 & 1 & 1 & \ldots \\ 1 & 2 & 4 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}^{-1} \left[\begin{pmatrix} 1 & 0 & 0 & \ldots \\ 1 & 1 & 1 & \ldots \\ 1 & 2 & 4 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \end{pmatrix} \odot \begin{pmatrix} 1 & 0 & 0 & \ldots \\ 1 & 1 & 1 & \ldots \\ 1 & 2 & 4 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \end{pmatrix} \right]$$

where $\odot$ is the component-wise multiplication of two vectors.

This suggests the following generalization of Algo. 1 for any fixed $k \geq 2$. First, let $V = \left(i^j\right)_{i,j=0}^{2k-1}$ (Vandermonde matrix), and precompute V^{-1}.

Algorithm 2 Toom-k

Input: $f = f_0 + \cdots + f_{n-1}X^{n-1}$, $g = g_0 + \cdots + g_{n-1}X^{n-1}$

Output: $h = h_0 + \cdots + h_{2n-1}X^{2n-1}$ such that $h = fg$

1. If $n < \max(k, 16)$, compute h naively and stop # Forget the "16" until Sec. 2.4
2. Write $f = F_0 + F_1 X^{\lceil n/k \rceil} + \cdots + F_{k-1}X^{(k-1)\lceil n/k \rceil}$ and $g = G_0 + G_1 X^{\lceil n/k \rceil} + \cdots + G_{k-1}X^{(k-1)\lceil n/k \rceil}$ where $\deg(F_i)$ and $\deg(G_i) < \frac{n}{k}$
3. Define $F_i = G_i = 0$ for $\frac{n}{k} < i \leq 2k-1$
4. Compute $\bar{f} = V\begin{pmatrix} F_0 \\ F_1 \\ \vdots \\ F_{2k-1} \end{pmatrix}$ and $\bar{g} = V\begin{pmatrix} G_0 \\ G_1 \\ \vdots \\ F_{2k-1} \end{pmatrix}$
5. Compute $\bar{h} = \bar{f} \odot \bar{g}$ recursively
6. Return $V^{-1}\bar{h}$

Remark 2.4. If we write $F_i = f_0^{(i)} + \cdots + f_d^{(i)} X^d$ for $i \in \{0, \ldots, k-1\}$, one can compute the product $V \cdot (F_i)$ as

$$
\begin{aligned}
V \cdot \begin{pmatrix} F_0 \\ F_1 \\ \vdots \\ F_{k-1} \end{pmatrix} &= V \cdot \left[\begin{pmatrix} f_0^0 \\ f_0^{(1)} \\ \vdots \\ f_0^{(k-1)} \end{pmatrix} + \begin{pmatrix} f_1^0 \\ f_1^{(1)} \\ \vdots \\ f_1^{(k-1)} \end{pmatrix} X + \cdots + \begin{pmatrix} f_d^0 \\ f_d^{(1)} \\ \vdots \\ f_d^{(k-1)} \end{pmatrix} X^d \right] \\
&= V \cdot \begin{pmatrix} f_0^0 \\ f_0^{(1)} \\ \vdots \\ f_0^{(k-1)} \end{pmatrix} + V \cdot \begin{pmatrix} f_1^0 \\ f_1^{(1)} \\ \vdots \\ f_1^{(k-1)} \end{pmatrix} X + \cdots + V \cdot \begin{pmatrix} f_d^0 \\ f_d^{(1)} \\ \vdots \\ f_d^{(k-1)} \end{pmatrix} X^d
\end{aligned}
$$

so the cost of computing that product is $O(dk^2)$.

Theorem 2.5. *A run of Algorithm 2 requires $O(n^{\log_k(2k-1)})$ operations. In particular, for any fixed $\epsilon > 0$, there exists a multiplication algorithm for $R[X]$ which requires $O(n^{1+\epsilon})$ operations in R.*

Proof. See Exercise 2.2. ◻

Remark 2.6. For fixed k, the cost of precomputing V and V^{-1} can be neglected, since it is a fixed cost of $O(k^2)$ and $O(k^3)$ respectively.

2.4 Toom-Cook algorithm

Theorem 2.7 (Toom-Cook)**.** *There exists a multiplication algorithm for $R[X]$ that requires $O(n^{1+2/\sqrt{\log(n)}})$ operations in R. This algorithm is obtained by adapting Algo. 2 to choose at each recursion level $k = \left\lfloor 2^{2\sqrt{\log(n)}} \right\rfloor$.*

Proof. See Exercise 2.3. □

Remark 2.8. This complexity is better than that of Toom-k, since it is better than $O(2^{1+\epsilon})$ for *all* $\epsilon > 0$.

Remark 2.9. Strassen's algorithm for matrix multiplication is based on the same idea as Karatsuba's algorithm, and runs in time $O(n^{\log_2(7)}) \leq O(n^{2.82})$. Is there a Toom-Cook style algorithm for matrix multiplication, with complexity better than $O(2^{2+\epsilon})$ for all $\epsilon > 0$?

For even k, we can multiply $k \times k$ matrices with $\frac{1}{3}k^3 + 6k^2 - \frac{4}{3}k$ operations, so there are matrix multiplication algorithms with complexity $O(n^{\log_k(\frac{1}{3}k^3+6k^2-\frac{4}{3}k)})$. But $\log_k(\frac{1}{3}k^3 + 6k^2 - \frac{4}{3}k)$ tends to 3 when k tends to ∞. Its minimum (over $2\mathbb{N}$) is reached at $k = 70$, leading to a complexity $O(n^{2.796})$ (Pan's algorithm).

The current record is $O(n^{2.3728639})$ (Le Gall 2014), and yes, that many decimal points are necessary! It is conjectured that a complexity of $O(2^{1+\epsilon})$ for all ϵ is realizable.

Remark 2.10. It is conjectured that polynomial multiplication in $O(n)$ operations is not possible.

2.5 Exercises

Exercise 2.1. Is it possible to use the ideas of the Algorithm of Toom-k with evaluation at $\{0, 1, \ldots, k-2, \infty\}$? Describe the matrices V and V^{-1}.

Exercise 2.2. Prove Theorem 2.5.

Exercise 2.3. Prove Theorem 2.7.

Exercise 2.4. Show that there is no algorithm which can multiply two linear polynomials (over any ring) in 2 multiplications.

3 Fast multiplication in $\bar{k}[X]$

In this chapter, let k be an *algebraically closed* field. The problem to solve is the same as previously, but this time, we assume that $\deg(f) + \deg(g) < n$.

We will be considering evaluation/interpolation methods.

Algorithm 3 Evaluation/interpolation

Input: $f = f_0 + \cdots + f_{k-1}X^k$, $g = g_0 + \cdots + g_{l-1}X^l$ with $k + l < n$

Output: $h = h_0 + \cdots + h_{n-1}X^{n-1}$ such that $h = fg$

1. Fix $(x_0, \ldots, x_{n-1}) \in k^n$
2. Compute $f(x_i)$, $g(x_i)$ for $i = 0, \ldots, n-1$
3. Compute $h(x_i) = f(x_i)g(x_i)$ for $i = 0, \ldots, n-1$
4. Compute h by interpolating $h(x_i)$ for $i = 0, \ldots, n-1$

Remark 3.1. In general, Algo. 3 requires $O(n^2) + O(n) + O(n^2) = O(n^2)$ operations in k, like the classical algorithm. The idea is to choose specific values of $x_0, \ldots, x_{n-1}$ so that steps 2 and 4 can be done faster.

3.1 Roots of unity and discrete Fourier transform

Definition 3.2. An element $\omega \in k$ is called a *n'th root of unity* if $\omega^n = 1$. It is a *primitive n*'th root of unity if additionally $\omega^i \neq 1$ for $0 < i < n$.

Example 3.3. In $\mathbb{C}$, -1 is a primitive second root of unity. i is a primitive 4th root of unity.

In $\mathbb{F}_{17}$, 2 is a primitive 8th root of unity.

Definition 3.4. The matrix

$$\mathrm{DFT}_n := \mathrm{DFT}_n^{(\omega)} := \left(\omega^{ij}\right)_{i,j=0}^{n-1} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{pmatrix} \in k^{n \times n}$$

is called the *discrete Fourier transform* (wrt ω).

Example 3.5. In $\mathbb{C}$, the discrete Fourier transform wrt i is

$$\mathrm{DFT}_4^{(i)} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}.$$

Remark 3.6. The DFT is a Vandermonde matrix. In particular, if $f = f_0 + f_1X + \cdots + f_{n-1}X^{n-1}$,

$$\mathrm{DFT}_n^{(\omega)} \cdot \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f(\omega^0) \\ f(\omega^1) \\ \vdots \\ f(\omega^{n-1}) \end{pmatrix}.$$

Definition 3.7. Let $f, g \in k^n$. The *product* $f \odot g$ is the vector whose i'th coordinate is given by $f_i g_i$. The *convolution* $f * g$ is the vector whose i'th coordinate is given by

$$\sum_{k=0}^{n-1} f_k g_{(i-k) \bmod n}.$$

Lemma 3.8. *Let ω be a primitive n'th root of unity. Then*

1. *there is a factorization*

$$X^n - 1 = (X - \omega)(X - \omega^2) \cdots (X - \omega^n);$$

2. *for any $j \in \{1, \ldots, n-1\}$,*

$$\sum_{i=0}^{n-1} \omega^{ij} = 0.$$

3. *there is a group isomorphism*

$$\left(\{\omega^i : i \in \mathbb{Z}\}, \cdot\right) \simeq (\mathbb{Z}/n\mathbb{Z}, +)$$

4. *the DFT matrix is easy to invert:*

$$\left(\mathrm{DFT}_n^{(\omega)}\right)^{-1} = \frac{1}{n}\mathrm{DFT}_n^{(1/\omega)}$$

5. *if $m \mid n$, then ω^m is a primitive (n/m)'th root of unity*
6. *the DFT is compatible with convolution*

$$\mathrm{DFT}_n(f * g) = \mathrm{DFT}_n(f) \odot \mathrm{DFT}_n(g)$$

Proof. 1. All ω^i are distinct: if $\omega^i = \omega^j$ with $1 \le i < j \le n$, then $\omega^{j-i} = 1$ with $0 < j - i < n$, which is a contradiction because ω is a primitive root of unity. All ω^i are roots of $X^n - 1$, since $(\omega^i)^n = (\omega^n)^i = 1$, so the $X - \omega^i$ are n distinct factors of $X^n - 1$. By comparing the degree and leading coefficient, we get the wanted factorization.

2. Use the formula

$$\left(\sum_{i=0}^{n-1} X^i\right)(X-1) = X^n - 1$$

Evaluated at $X = \omega^j$ for $0 < j < n$, the right hand side is 0, the factor $(\omega^j - 1)$ is non-zero, so the sum has to be zero.

3. Clear.

4. Evaluate the product:

$$\begin{aligned}
\mathrm{DFT}_n^{(\omega)}\mathrm{DFT}_n^{(1/\omega)} &= \left(\omega^{ij}\right)_{i,j=0}^{n-1} \cdot \left(\omega^{-ij}\right)_{i,j=0}^{n-1} \\
&= \left(\sum_{k=0}^{n-1} \omega^{ik}\omega^{-kj}\right)_{i,j=0}^{n-1} \\
&= \left(\sum_{k=0}^{n-1} \omega^{k(i-j)}\right)_{i,j=0}^{n-1} \\
&= \left(n\delta_{ij}\right)_{i,j=0}^{n-1}.
\end{aligned}$$

5. Clear.

6. If we associate the vector $f = (f_0, \dots, f_{n-1})$ with the polynomial $f(X) = f_0 + \cdots + f_{n-1}X^{n-1}$, convolution is equivalent to multiplication in $k[X]/\langle X^n - 1\rangle$, that is

$$(f * g)(X) = f(X)g(X) + q(X) \cdot (X^n - 1)$$

for some $q \in k[X]$. Indeed, write

$$\begin{aligned}
f(X)g(X) &= \sum_{i,j=0}^{n-1} f_i g_j X^{i+j} \\
&= \sum_{i+j<n} f_i g_j X^{i+j} + \sum_{n \le i+j < 2n} f_i g_j X^{i+j} \\
&= \underbrace{\sum_{i+j<n} f_i g_j X^{i+j} + \sum_{n \le i+j < 2n} f_i g_j X^{i+j-n}}_{(f*g)(X)} \underbrace{- \sum_{n \le i+j < 2n} f_i g_j X^{i+j-n} + \sum_{n \le i+j < 2n} f_i g_j X^{i+j}}_{(\sum_{n \le i+j < 2n} f_i g_j X^{i+j-n})(X^n - 1)}
\end{aligned}$$

The claim follows by evaluation at ω^i. □

The remark, together with property 4, makes powers of ω a good choice for evaluation and interpolation: if we can just find a fast way to evaluate $\mathrm{DFT}_n \cdot f$, we can perform both steps in a fast way.

3.2 Fast Fourier transform

Given $f = \begin{pmatrix} f_0 \\ \vdots \\ f_{2n-1} \end{pmatrix}$, we want to compute $\bar{f} = \mathrm{DFT}_{2n} \cdot f$.

Let's expand the j'th coefficient:

$$\begin{aligned}
\left(\mathrm{DFT}_{2n}^{\omega} f\right)_j &= \sum_{i=0}^{2n-1} \omega^{ij} f_i \\
&= \sum_{i=0}^{n-1} \omega^{2ij} f_{2i} + \sum_{i=0}^{n-1} \omega^{(2i+1)j} f_{2i+1} \\
&= \sum_{i=0}^{n-1} (\omega^2)^{ij} f_{2i} + \omega^j \sum_{i=0}^{n-1} (\omega^2)^{ij} f_{2i+1} \\
&= \begin{cases} \left(\mathrm{DFT}_n^{(\omega^2)} f_{\text{even}}\right)_j + \omega^j \left(\mathrm{DFT}_n^{(\omega^2)} f_{\text{odd}}\right)_j & \text{for } 0 \le j < n \\ \left(\mathrm{DFT}_n^{(\omega^2)} f_{\text{even}}\right)_{j-n} + \omega^j \left(\mathrm{DFT}_n^{(\omega^2)} f_{\text{odd}}\right)_{j-n} & \text{for } n \le j < 2n \end{cases} \\
&= \begin{cases} \left(\mathrm{DFT}_n^{(\omega^2)} f_{\text{even}}\right)_j + \omega^j \left(\mathrm{DFT}_n^{(\omega^2)} f_{\text{odd}}\right)_j & \text{for } 0 \le j < n \\ \left(\mathrm{DFT}_n^{(\omega^2)} f_{\text{even}}\right)_{j-n} - \omega^{j-n} \left(\mathrm{DFT}_n^{(\omega^2)} f_{\text{odd}}\right)_{j-n} & \text{for } n \le j < 2n \end{cases}
\end{aligned}$$

We can use this property to perform the evaluation and interpolation steps.

Algorithm 4 Fast Fourier Transform

Input: $f \in k^n$, ω a primitive n'th root of unity, $n = 2^k$

Output: $\bar{f} = \mathrm{DFT}_n^{(\omega)} f$

1. If $n = 1$ then return (f_0)
2. $u \leftarrow \mathsf{FFT}([f_0, f_2, \ldots,], \omega^2, n/2)$, $v \leftarrow \mathsf{FFT}([f_1, f_3, \ldots,], \omega^2, n/2)$
3. Return $[u_0 + v_0, u_1 + \omega v_1, u_2 + \omega^2 v_2, \ldots, u_{n/2-1} + \omega^{n/2-1} v_{n/2-1},$
   $u_0 - v_0, u_1 - \omega v_1, u_2 - \omega^2 v_2, \ldots, u_{n/2-1} - \omega^{n/2-1} v_{n/2-1}]$

Theorem 3.9. *Algo. 4 requires $O(n \log(n))$ operations in k.*

Proof. Similar to before, with the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

∎

This allows us to rewrite Algo. 3 with the FFT.

Algorithm 5 Evaluation/interpolation multiplication using FFT

Input: $f = f_0 + \cdots + f_{k-1}X^k$, $g = g_0 + \cdots + g_{l-1}X^l$ with $k + l < n$

Output: $h = h_0 + \cdots + h_{n-1}X^{n-1}$ such that $h = fg$

1. $\omega \leftarrow$ primitive n'th root of unity
2. $\bar{f} \leftarrow \mathsf{FFT}(f, \omega)$, $\bar{g} \leftarrow \mathsf{FFT}(g, \omega)$
3. $\bar{h} \leftarrow \bar{f} \odot \bar{g}$
4. Return $\frac{1}{n}\mathsf{FFT}(\bar{h}, \omega^{-1})$

Theorem 3.10. *Multiplication in* $k[X]$ *can be done with* $O(n \log n)$ *operations in* k *if* k *is algebraically closed.*

Remark 3.11. This complexity is currently the best known complexity for polynomial multiplication.

Remark 3.12. Let P be the permutation matrix such that

$$P \cdot f = \begin{pmatrix} f_{\text{even}} \\ f_{\text{odd}} \end{pmatrix}$$

and Δ be the diagonal matrix

$$\Delta = \begin{pmatrix} 1 & & & \\ & \omega & & \\ & & \omega^2 & \\ & & & \ddots \end{pmatrix}.$$

Then the computations above yield that

$$\begin{aligned} \mathrm{DFT}_{2n} &= \begin{pmatrix} \mathrm{DFT}_n & \Delta \mathrm{DFT}_n \\ \mathrm{DFT}_n & -\Delta \mathrm{DFT}_n \end{pmatrix} \cdot P \\ &= \begin{pmatrix} I & \Delta \\ I & -\Delta \end{pmatrix} \cdot \begin{pmatrix} \mathrm{DFT}_n & \\ & \mathrm{DFT}_n \end{pmatrix} \cdot P \\ &= \begin{pmatrix} I & I \\ I & -I \end{pmatrix} \cdot \begin{pmatrix} I & \\ & \Delta \end{pmatrix} \cdot \begin{pmatrix} \mathrm{DFT}_n & \\ & \mathrm{DFT}_n \end{pmatrix} \cdot P \end{aligned}$$

This can be generalized to divisions by m instead of 2. Skipping over the details, this gives

$$\mathrm{DFT}_{mn} = \begin{pmatrix} I & I & I & \dots \\ I & \omega^n I & \omega^{2n} I & \dots \\ I & \omega^{2n} I & \omega^{4n} I & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \cdot \begin{pmatrix} I & & & \\ & \Delta & & \\ & & \Delta^2 & \\ & & & \ddots \end{pmatrix} \cdot \begin{pmatrix} \mathrm{DFT}_n & & & \\ & \mathrm{DFT}_n & & \\ & & \mathrm{DFT}_n & \\ & & & \ddots \end{pmatrix} \cdot P.$$

This is a result due to Cooley and Tuckey, which can be used to refine Algo. 4 so that it reduces an FFT of *any* size quickly to FFT's of prime size.

4 Fast multiplication in $R[X]$

Remark 4.1. Algo. 4 does not require that the base ring R be an algebraically closed field, but that:

1. R contains a primitive n'th root of unity ω;
2. $n = 1 + 1 + \cdots + 1$ is invertible.

In this chapter, we will see how to perform FFT without those hypotheses.

4.1 If 2 is invertible

In this section, assume that R has no n'th root of unity, but that $2 \in R^*$.

Let $f, g \in R[X]$, with $\deg f + \deg g < n = 2^k$, as before we want to compute $h = fg$. Write $n = pq$ where $p = 2^{\lceil k/2 \rceil}$ and $q = 2^{\lfloor k/2 \rfloor}$, so $p \simeq q \simeq \sqrt{n}$.

Write

$$\begin{aligned} f &= F_0 + F_1 X^q + F_2 X^{2q} + \ldots \\ g &= G_0 + G_1 X^q + G_2 X^{2q} + \ldots \end{aligned}$$

with $\deg F_i < q$, $\deg G_i < q$, and define two polynomials in $R[X, Y]$

$$\begin{aligned} \bar{f} &= F_0 + F_1 Y + F_2 Y^2 + \ldots \\ \bar{g} &= G_0 + G_1 Y + G_2 Y^2 + \ldots . \end{aligned}$$

Then $\deg_X \bar{f}, \deg_X \bar{g} < q$, $\deg_Y \bar{f}, \deg_Y \bar{g} < p$, and $f = \bar{f}(X, X^q)$, $g = \bar{g}(X, X^q)$. Let $\bar{h} = \bar{f}\bar{g}$, then $\deg_X \bar{h} < 2q$ and $\deg_Y \bar{h} < 2p$.

Note 4.2. It suffices to compute $\bar{h} \bmod Y^p + 1$ because

$$\deg h = \deg \bar{h}(X, X^q) < pq = n.$$

Note 4.3. Since $\deg_X \bar{h} < 2q$,

$$\bar{h}(X, Y) = \bar{h}(X, Y) \bmod X^{2q} + 1.$$

Hence, together with the previous note, we can compute in

$$\left(R[X]/\left\langle X^{2q} + 1\right\rangle\right)[Y]/\langle Y^p + 1\rangle.$$

We denote by D the ring

$$D := R[X]/\left\langle X^{2q} + 1\right\rangle.$$

Proposition 4.4. *In the ring D, X is a $4q$'th primitive root of unity. Furthermore, let*

$$\omega = \begin{cases} X^2 & \text{if } p = q \\ X & \text{if } p = 2q. \end{cases}$$

Then $\omega = X$ is a $2p$'th primitive root of unity in D.

With this setting, if

$$\bar{f}(Y) \cdot \bar{g}(Y) = \bar{h}(Y) \bmod Y^p + 1$$

then

$$\bar{f}(\omega Y) \cdot \bar{g}(\omega Y) = \bar{h}(\omega Y) \bmod (\omega Y)^p + 1 = 1 - Y^p$$

Algorithm 6 Schönhage-Strassen

Input: $f, g \in R[X]$ with $\deg f, \deg g < n = 2^k$

Output: $h = fg \bmod X^n + 1$

1. If $k \leq 2$ then compute h directly
2. Define $p, q \in \mathbb{N}$, $\bar{f}, \bar{g} \in D[Y]$ and $\omega \in D$ as above
3. Use Algo. 5 to compute $\bar{h} \in D[y]$ with

$$\bar{h}(\omega Y) = \bar{f}(\omega Y)\bar{g}(\omega Y) \bmod Y^p - 1$$

using ω^2 as a p'th root of unity in D and Algo. 6 recursively for multiplications in D

4. Return $h = \bar{h}(X, X^q) \bmod X^n + 1$

Remark 4.5. The algorithm requires that 2 be invertible for the FFT step: each call to the FFT multiplication algorithm is with a power of 2 as n.

Theorem 4.6. *Algo. 6 requires $O(n \log(n) \log(\log(n)))$ operations in R.*

Remark 4.7. For all practical purposes, $\log \log n \leq 6$.

Proof. Let $n \gg 1$ and suppose that

$$T(m) \leq c_1 m \log m \log \log m$$

for all $m < n$ and some constant c_1. Recall that $n = 2^k$, $p = 2^{\lceil k/2 \rceil} \leq 2\sqrt{n}$, $q = 2^{\lfloor k/2 \rfloor} \leq \sqrt{n}$.

The runtime function satisfies the recurrence

$$T(n) \leq pT(2q) + O(n \log n)$$

where $pT(2q)$ is the cost of p component-wise multiplication of polynomials of degree at most $2q$, and the trailing $O(n \log n)$ is the cost of the FFT.

Let $T_l(k) = T(2^k)$, and expand in terms of k :

$$
\begin{aligned}
T_l(k) &\leq 2^{\lceil k/2\rceil} T_l\left(\left\lfloor \frac{k}{2} \right\rfloor + 1\right) + c_2 2^k k \\
&\leq c_1 2^{\lceil k/2\rceil} 2^{\lfloor k/2\rfloor + 1} \left(\left\lfloor \frac{k}{2} \right\rfloor + 1\right) \log\left(\left\lfloor \frac{k}{2} \right\rfloor + 1\right) + c_2 2^k k \\
&\leq c_1 \underbrace{2^{\lceil k/2\rceil + \lfloor k/2\rfloor}}_{=2^k} \cdot \underbrace{\underbrace{2\left(\left\lfloor \frac{k}{2} \right\rfloor + 1\right)}_{\leq k+2} \underbrace{\log\left(\underbrace{\left\lfloor \frac{k}{2} \right\rfloor + 1}_{\leq \frac{3}{4}k}\right)}_{\leq \log k - \log(4/3)}}_{\leq k\log k - k\log(4/3) + 2\log k - 2\log(4/3)} + c_2 2^k k \\
&\leq c_1 2^k k \log(k) + \underbrace{c_1 2^k \underbrace{\left(2\log k - 2\log(\frac{4}{3})\right)}_{\leq \frac{1/2}{k}\log(4/3)} + (c_2 - c_1 \log(\frac{4}{3})) 2^k k}_{\leq (c_2 - \frac{1}{2}\log(4/3)) 2^k k}
\end{aligned}
$$

Without loss of generality we can assume that $c_1 \geq 2c_2/\log 4/3$, so

$$T_l(k) \leq c_1 2^k k \log(k)$$

and indeed

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

□

4.2 If 2 is not invertible

The previous algorithm requires 2 to be invertible in order to divide the reverse DFT by 2^k. Without this assumption, we can skip that division, and Algo. 5 returns $2^k fg$. Analogously, we can compute $3^l fg$ using a 3-adic FFT. Then, Euclid's extended algorithm yields $u, v \in \mathbb{Z}$ such that

$$u \cdot 2^k + v \cdot 3^l = 1,$$

so

$$u \cdot 2^k fg + v \cdot 3^l fg = fg.$$

Theorem 4.8. *Polynomials in $R[X]$ of degree less than n can be multiplied using $O(n \log n \log\log n)$ operations in R, for any commutative ring R with a unity.*

Remark 4.9. This is the current world record.

4.3 Multiplication time function

Definition 4.10. Let R be a ring. A function $M : R \to \mathbb{N}$ is called *multiplication time* for $R[X]$ if there exists an algorithm that multiplies $f, g \in R[X]$ with $\deg f$, $\deg g < n$ using no more than $M(n)$ operations in R.

Finding the best possible M for various rings in an active field of research.

Proposition 4.11. *We can assume that:*

1. *if R is infinite, M is worse than linear:*

$$\frac{M(n)}{n} > \frac{M(m)}{m} \text{ if } n > m;$$

2. *in particular,*

$$M(mn) \geq mM(n)$$

and

$$M(m+n) \geq M(m) + M(n);$$

3. *M is at most quadratic:*

$$M(nm) \leq m^2 M(n)$$

4. *M is at most the complexity of the general algorithm by Schönhage and Strassen:*

$$M(n) = O(n \log n \log \log n).$$

5 Fast multiplication in $\mathbb{Z}$

Here, we are given two *integers* $f, g \in \mathbb{Z}$ with at most n digits (in base 2), and we want to compute $h = fg$.

5.1 Integer multiplication in theory

Remark that if

$$f = f_0 + 2f_1 + \cdots + 2^{n-1}f_{n-1},$$

f is the evaluation of the polynomial

$$\bar{f} = f_0 + f_1X + \cdots + f_{n-1}X^{n-1}$$

at $X = 2$.

This reduces integer multiplication to polynomial multiplication, with similar complexity results.

Theorem 5.1 (Schönhage-Strassen)**.** *Integers of length n can be multiplied in time $O(n \log n \log \log n)$.*

Remark 5.2. It is conjectured that the lower bound for the complexity of integer multiplication is $cn \log n$.

The current best results are the following.

Definition 5.3. For $x \in \mathbb{R}_{>1}$, the *iterated logarithm* of x is

$$\log^*(x) = \max\{k \in \mathbb{N} : \log^k(x) \leq 1\}.$$

Remark 5.4. For all practical purposes, $\log^*(n) \leq 4$.

Theorem 5.5 (Fürer, 2007)**.** *Integers of length n can be multiplied in time*

$$n \log n 2^{O(\log^*(n))}.$$

Remark 5.6. Beware of constants! In general,

$$2^{O(f(n))} \neq O(2^{f(n)})$$

Indeed

$$2^{cf(n)} = (2^{f(n)})^c$$

which will in general grow faster than $2^{f(n)}$.

In the recent years, researchers have focused on improving that constant, the current best result is the following:

Theorem 5.7 (Harvey, van der Hoeven, 2018)**.** *Integers of length n can be multiplied in time*

$$O(n \log n 2^{2\log^*(n)}).$$

Remark 5.8. Forgetting the constants, we have

$$\log\log n \geq 2^{2\log^* n} \iff n \geq 2^{2^{2^{12}}}.$$

Remember that n is the *number of digits* of the integers we want to multiply!

5.2 Integer multiplication in practice

Those algorithms are only of theoretical interest. The following algorithm follows a more pragmatic approach, which is usually superior.

Write $F = (f_{n-1} \dots f_1 f_0)_w$ and $G = (g_{n-1} \dots g_1 g_0)_w$ in base w with w as large as possible. In practice, one can for example choose w to be the largest possible processor word.

Define

$$\begin{aligned}
\bar{f} &= f_0 + f_1 X + \dots + f_{n-1} X^{n-1} \\
\bar{g} &= g_0 + g_1 X + \dots + g_{n-1} X^{n-1}
\end{aligned}$$

so that $\bar{f}(w) = f$ and $\bar{g}(w) = g$. Let

$$\bar{h} = \bar{f}\bar{g} = \bar{h}_0 + \bar{h}_1 X + \dots$$

Note that $0 \leq \bar{h}_i \leq nw^2$ for all i.

Assume that $n < w/8$ and fix three primes p_1, p_2, p_3 between $w/2$ and w, for which the field $\mathbb{F}_{p_i}$ contains a 2^t'th root of unity for some large t. Then compute $\bar{f}\bar{g}$ in $\mathbb{F}_{p_i}[X]$ for $i = 1, 2, 3$, and reconstruct the coefficients of $\bar{h}$ with the Chinese remainder theorem. Finally compute $h = \bar{h}(w)$.

Example 5.9. On a 64-bits processor, let's choose $w = 2^{64}$. Then

$$\begin{aligned}
p_1 &= 95 \cdot 2^{57} - 1 \\
p_2 &= 108 \cdot 2^{57} - 1 \\
p_3 &= 123 \cdot 2^{57} - 1
\end{aligned}$$

are suitable primes, with $t = 57$ and $55, 65$ and 493 the respective 57'th roots of unity.

This is the method of choice for multiplying integers up to $\simeq$ 500 millions of bits on a 64-bits architecture.

6 Fast multiplication in $R[X, Y]$

Given $f, g \in R[X, Y]$, we want to compute $h = fg$.

6.1 Isolating a variable

We can use Algo. 6 in $R[X][Y]$. But the complexity is bounded in number of operations in $R[X]$, not in R. In order to get a complete bound, we need an estimate for the degree growth in X.

If we define

$$\begin{aligned} d_X &:= \deg_X(h) = \deg_X(f) + \deg_X(g) \\ d_Y &:= \deg_Y(h) = \deg_Y(f) + \deg_Y(g) \end{aligned}$$

it suffices to compute the product in

$$R[X]/\langle X^{d_X+1} - 1\rangle[Y]/\langle Y^{d_Y+1} - 1\rangle.$$

Let

$$D := R[X]/\langle X^{d_X+1} - 1\rangle.$$

If we use for example Algo. 6 to compute the multiplication in $D[Y]/\langle Y^{d_Y+1} - 1\rangle$, it requires $M(d_Y)$ operations in D, each of them requires at most $M(d_X)$ operations in R.

Theorem 6.1. *Polynomials $f, g \in R[X, Y]$, with $\deg_X(f), \deg_X(g) \leq n$ and $\deg_Y(f), \deg_Y(g) \leq m$, can be multiplied with $M(n)M(m)$ operations in R.*

6.2 Kronecker substitution

Algorithm 7 Multiplication using Kronecker substitution

Input: $f, g \in R[X]$ with $\deg_X(fg) < n$, $\deg_Y(fg) < m$

Output: $h = fg$

1. $\bar{f} \leftarrow f(X, X^n)$, $\bar{g} \leftarrow g(X, X^N) \in R[X]$
2. Compute $\bar{h} = \bar{f} \cdot \bar{g} \in R[X]$ with a fast algorithm
3. Write $\bar{h} = h^{(0)} + h^{(1)}X^n + h^{(2)}X^{2n} + \cdots + h^{(m-1)}X^{(m-1)n}$ with $\deg(h^{(i)}) < n$
4. Return $h = h^{(0)} + h^{(1)}Y + h^{(2)}Y^2 + \cdots + h^{(m-1)}Y^{m-1}$

Theorem 6.2. *Algo. 7 requires $M(mn)$ operations in R.*

Proof. The only multiplication computed involves polynomials in $R[X]$ with degree at most nm. □

Remark 6.3. $M(mn)$ may not be strictly less than $M(m)M(n)$.

7 Fast division

Let K be a field. The task is, given $f, g \in K[X]$, to find $q, r \in K[X]$ such that $f = qg + r$ and $\deg(r) < \deg(g)$.

7.1 Horner's rule

Horner's rule is a technique for evaluating a polynomial f with degree m at some value v with $O(m)$ multiplications, instead of the naive m^2. It avoids computing successive powers of v, and instead relies on the following rewriting of f:

$$\begin{aligned} f &= a_0 + a_1 X + \cdots + a_m X^m \\ &= a_0 + X\Big(a_1 + X\big(\cdots + X(a_m)\ldots\big)\Big). \end{aligned}$$

The resulting algorithm is actually the naive Euclidean algorithm used to compute f divided by $g = X - v$. The remainder of that division is $f(v)$.

The same algorithm can be used for a polynomial g with degree n, and it then uses $O(nm)$ operations in K.

7.2 A Karatsuba-style algorithm: Jebelean's algorithm

There is also a Karatsuba-style division algorithm. Assume that $\deg f < 2 \deg g$ and $\deg g$ is a power of 2.

Algorithm 8 Jebelean's algorithm (1993)

Input: $f, g \in K[X]$, $k \in \mathbb{N}$, with $\deg g = n = 2^i$, $\deg f < 2n + k$.

Output: q, r such that $f = gX^k q + r$ and $\deg(r) < n + k$

1. If $\deg f < \deg g + k$, then return $q = 0$, $r = f$
2. If $\deg g = 1$, then use Horner's algorithm
3. Write $g = g^{(0)} + g^{(1)}X^{n/2}$ with $\deg g^{(0)} < \frac{n}{2}$ # $\deg g^{(1)} = \frac{n}{2}$

   ### Compute $q^{(1)}, r^{(1)}$ such that $f = q^{(1)}X^{n+k}g^{(1)} + r^{(1)}$ with $\deg r^{(1)} < \frac{3n}{2} + k$
4. Find $q^{(1)}, r^{(1)}$ by calling Algo. 8 with f, $g = g^{(1)}$ and $k = n + k$

   ### Compute the true remainder $u = f - q^{(1)}X^{n+k}g$
5. Compute $u = r^{(1)} - X^{n/2+k}g^{(0)}q^{(1)}$ using Algo. 1

   ### Compute $q^{(0)}, r^{(0)}$ such that $u = q^{(0)}X^{n/2+k}g^{(1)} + r^{(0)}$ with $\deg r^{(0)} < \frac{n}{2} + k$
6. Find $q^{(0)}, r^{(0)}$ by calling Algo. 8 with $f = u$, $g = g^{(1)}$ and $k = \frac{n}{2} + k$

   ### Compute the true remainder $r = u - q^{(0)}X^k g$
7. Compute $r = r^{(0)} - g^{(0)}q^{(0)}X^k$ using Algo. 1
8. Return $q = q^{(0)} + q^{(1)}X^{n/2}$ and r

Theorem 7.1. *Algo. 8 is correct.*

Proof. We prove it by induction on n, then on k. The case $n = 1$ is clear, as is the case $\deg f < n + k$. Now assume that the algorithm is correct for all input of size $< n$ or third argument $> k$. Consider $f, g \in R[X]$, $k \in \mathbb{N}$ with $\deg g = n$ and $\deg f < 2n + k$. In particular, $\deg g^{(1)} = \frac{n}{2}$.

So the call to Algo. 8 with $f = f$, $g = g^{(1)}$ and $k = n + k$ is correct, and the results are $q^{(1)}, r^{(1)}$ such that $f = g^{(1)}X^{n+k}q^{(1)} + r^{(1)}$, $\deg(r^{(1)}) < n + k$, and

$$\deg(q^{(1)}) = \deg(f) - \deg(X^{n+k}g^{(1)}) < \frac{n}{2}.$$

The polynomial u satisfies

$$\begin{aligned} u &= r^{(1)} - X^{n/2+k}g^{(0)}q^{(1)} \\ &= f - X^{n+k}g^{(1)}q^{(1)} - X^{n/2+k}g^{(0)}q^{(1)} \\ &= f - X^{n/2+k}q^{(1)}g, \end{aligned} \tag{7.1}$$

and it has degree

$$\deg(u) < \max\left(n + k, \frac{n}{2} + k + \frac{n}{2} + \frac{n}{2}\right) < \frac{3n}{2} + k.$$

The call to Algo. 8 with $f = u$, $g = g^{(1)}$ and $k = \frac{n}{2} + k$ is correct, and $\deg r^{(0)} < n + k$ and

$\deg(q^{(0)}) = \frac{n}{2} + k$. So we get

$$\begin{aligned} u &= X^{n/2+k} g^{(1)} q^{(0)} + r^{(0)} \\ &= X^{n/2+k} g^{(1)} q^{(0)} + X^k g^{(0)} q^{(0)} + r \quad \text{(by definition of } r\text{)} \\ &= g X^k q^{(0)} + r. \end{aligned}$$

The polynomial r has degree

$$\deg(r) < \max\left(n+k, \frac{n}{2} + \frac{n}{2} + k\right) < n+k,$$

and putting it all together using Eq. (7.1), we find

$$f = X^{n/2+k} q^{(1)} g + X^k q^{(0)} g + r = X^k \left(X^{n/2} q^{(1)} + q^{(0)}\right) g + r.$$

□

Theorem 7.2 (Jebelean, 1993)**.** *Algo. 8 requires at most $2M_K(n)$ multiplications in K where $M_K(n)$ is the number of multiplications performed by Algo. 1 (Karatsuba).*

Remark 7.3. There is no O in that result.

Proof. Recall the recurrence relation

$$M_K(2n) = 3M_K(n).$$

If we proceed by induction, the number of multiplications $T(n)$ performed by Algo. 8 satisfies the recurrence relation

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 2M_K\left(\frac{n}{2}\right) \\ &= 2 \cdot 2M_K\left(\frac{n}{2}\right) + 2M_K\left(\frac{n}{2}\right) \\ &= 6M_K\left(\frac{n}{2}\right) \\ &= 2M_K(n). \end{aligned}$$

□

Remark 7.4. The integer version of Algo. 8 is the best-performing division algorithm for integers of a certain size.

Remark 7.5. The total number of operations (including additions) is $O(M_K(n)\log(n))$.

7.3 Division with the cost of multiplication

We now want to perform division in time $O(M(n))$.

Definition 7.6. Let $f \in K[X]$ and $k \in \mathbb{N}$, the *k'th reversal* of f is

$$\mathrm{rev}_k(f) := X^k f\left(\frac{1}{X}\right).$$

Example 7.7. If $f = f_0 + f_1X + \cdots + f_nX^n$, then $\mathrm{rev}_n(f) = f_n + f_{n-1}X + \cdots + f_0X^n$.

Remark 7.8. In general, $\mathrm{rev}_k(f) \in K[X]$ if $k \geq n$.

Let $f, g \in K[X]$ with $\deg(f) = m$, $\deg g = n < m$, and q, r be the quotient and remainder respectively of the division of f by g. Performing the change of variable $X \mapsto 1/X$ and multiplying by X^m the equality $f = qg + r$ gives

$$\begin{aligned} X^m f\left(\frac{1}{X}\right) &= X^n g\left(\frac{1}{X}\right) X^{m-n} q\left(\frac{1}{X}\right) + X^{m-n+1}X^{n-1} r\left(\frac{1}{X}\right) \\ \mathrm{rev}_m f &= \mathrm{rev}_n g \cdot r_{m-n} q + X^{m-n+1}\mathrm{rev}_{n-1} r \end{aligned}$$

so

$$\mathrm{rev}_m f = \mathrm{rev}_n g \cdot \mathrm{rev}_{m-n} q \bmod X^{m-n+1}.$$

Furthermore, since $\deg g = n$, we have $(\mathrm{rev}_n g)_0 \neq 0$, so $\mathrm{rev}_n g$ is invertible modulo X^{m-n+1}. Therefore

$$\mathrm{rev}_{m-n} q = \mathrm{rev}_m f \cdot (\mathrm{rev}_n g)^{-1} \mod X^{m-n+1}.$$

So what we need is a fast algorithm for inversion modulo X^l: an algorithm which, given $u \in K[X]$ with $u_0 \neq 0$ and $l \in \mathbb{N}$, computes $v \in K[X]$ such that $uv = 1 \bmod X^l$.

Regard $u \in K[X] \subset K[[X]]$ as a formal power series, and consider the map

$$\begin{array}{cccc} \phi : & K[[X]]^* & \to & K[[X]] \\ & s & \mapsto & u - \frac{1}{s}. \end{array}$$

Let v be a root of ϕ, we can write

$$v = w + X^l r$$

with $w \in K[X]_{l-1}$, and w, seen as a power series, is invertible. Then

$$\begin{aligned} 0 = \phi(v) &= u - \frac{1}{w + X^l r} = u - \frac{1}{w}\frac{1}{1 + X^l r/w} \\ &= u - \frac{1}{w} + X^l \frac{r}{v_0^2} - O(X^{l+1}) \end{aligned}$$

so

$$uw = 1 + X^l \frac{r}{w} + O(X^{l+1}) = 1 \bmod X^l.$$

So we have to find an approximation of order l of a root v of ϕ. For this purpose, we use Newton iteration: we compute successive approximations of the root, starting with

$$v^{(0)} = \frac{1}{u_0}$$

and iterating with

$$\begin{aligned} v^{(k+1)} &= v^{(k)} - \frac{\phi(v^{(k)})}{\phi'(v^{(k)})} = v^{(k)} - \frac{u - \frac{1}{v^{(k)}}}{\left(\frac{1}{v^{(k)}}\right)^2} \\ &= 2v^{(k)} - u \cdot (v^{(k)})^2. \end{aligned}$$

This would give us an algorithm, if only we knew when to stop!

Theorem 7.9. *For all $k \geq 0$, $u \cdot v^{(k)} = 1 \bmod X^{2^k}$.*

Proof. Proof by induction: for $k = 0$, we have

$$u \cdot v^{(0)} = u_0 \cdot \frac{1}{u_0} + O(X) = 1 \bmod X.$$

If it is true for $k \geq 0$, then

$$\begin{aligned} 1 - uv^{(k+1)} &= 1 - u(2v^{(k)} - u \cdot (v^{(k)})^2) = 1 - 2uv^{(k)} + (uv^{(k)})^2 = \left(1 - u \cdot v^{(k)}\right)^2 \\ &= O(X^{2^{k+1}}) = 0 \bmod X^{2^{k+1}}. \end{aligned}$$

□

Remark 7.10. This theorem is a particular case of a more general fact: with a starting point sufficiently close to a root, Newton iteration converges quadratically fast.

Algorithm 9 Inversion using Newton iteration

Input: $u \in K[X]$ with $u_0 \neq 0$, $n \in \mathbb{N}$

Output: $v \in K[X]$ with $u \cdot v = 1 \bmod X^n$

1. $v \leftarrow \frac{1}{u_0}$
2. For i from 1 to $\lceil \log(n) \rceil$, do
3. $\quad v \leftarrow 2v - uv^2 \text{ rem } X^l$
4. Return v

Theorem 7.11. *Algo. 9 requires $O(M(n))$ operations in K.*

Proof. Let $T(n)$ be the number of operations required. Then

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^{\lceil \log(n) \rceil} 2M(2^i) + c2^i \\
&\leq c2^{\lceil \log(n) \rceil + 1} + 2 \sum_{i=1}^{\lceil \log(n) \rceil} \underbrace{M(2^i)}_{\leq \frac{M(n)}{n/2^i}} \\
&\leq 4cn + 2\frac{M(n)}{n} \underbrace{\sum_{i=1}^{\lceil \log(n) \rceil} 2^i}_{\leq 4n} \\
&\leq 4cn + 8M(n) = O(M(n)).
\end{aligned}$$

□

With this taken care of, we can now write down all the steps required to perform a fast division.

Algorithm 10 Fast division

Input: $f, g \in K[X]$, $k \in \mathbb{N}$, with $\deg f = m$, $\deg g < n$, $g \neq 0$

Output: q, r such that $f = qg + r$ and $\deg(r) < \deg(g)$

1. If $m < n$ then return $q = 0$, $r = f$
2. Compute $h = \mathrm{rev}_n(g)^{-1} \bmod X^{m-n+1}$ with Algo. 9
3. $\bar{q} \leftarrow \mathrm{rev}_m(f)h$
4. Return $q = \mathrm{rev}_{m-n}(\bar{q})$ and $r = f - gq$

Theorem 7.12. *Algo. 10 requires $O(M(m))$ operations in K.*

Remark 7.13. This result is the current world record for polynomial division.

Remark 7.14. In particular, if $f, g, q \in K[X]$ with $\deg(f), \deg(g), \deg(q) \leq n$, then we can compute (and reduce) $f, g \in K[X]/\langle q \rangle$ with $O(M(n))$ operations in K.

If $\gcd(f, g) = 1$, then we will see that $f^{-1} \bmod q$ can be computed using $O(M(n)\log(n))$ operation in K (using the fast GCD algorithm).

7.4 Exercises

Exercise 7.1. Assume that the field K is algebraically closed. Find a bound for the complexity of Algo. 8 if we use FFT instead of Karatsuba's algorithm for the multiplication. Is it better?

Exercise 7.2. How would you adapt Algo. 8 to work with any polynomial g (even if its degree is not a power of 2)?

Exercise 7.3.

1. Write an analogue of Algo. 8 for polynomials such that $\deg(f) \leq 3\deg(g)$. What is its complexity?
2. Generalize to any $f, g \in K[X]$. What is the resulting complexity?

8 Fast evaluation and interpolation

Fast multiplication and fast division algorithms are useful because those operations are heavily used in many higher-level algorithms. However, it is frequently not enough, in order to obtain a speed-up, to replace the operations with their fast counterparts.

Example 8.1. Given $n \in \mathbb{N}$ (with n smaller than a machine word), how to compute $n!$?

The usual algorithm uses the formula

$$n! = n \cdot (n-1)!.$$

This algorithm is recursively called linearly-many times, and at each step does one multiplication with a small integer (with a linear cost). Its complexity satisfies

$$T(n) = T(n-1) + O(n),$$

so

$$T(n) = O(n^2).$$

On the other hand, an algorithm using the following formula

$$n! = \left(\frac{n}{2}\right)! \cdot \left(\prod_{k=\frac{n}{2}+1}^{n} k\right)$$

is recursively called log-many times, and at each step adds one large multiplication. Its complexity satisfies

$$T(n) = 2T\left(\frac{n}{2}\right) + M\left(\frac{n}{2}\right)$$

so

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

If $M(n) = O(n^2)$, it's worse. If $M(n) = \tilde{O}(n)$, it's better.

The lesson is that in order to take advantage of fast multiplication, algorithms need to be adjusted. It is usually not sufficient to plug fast multiplication into a standard algorithm.

We want to do two things in this chapter:

Evaluation Given $f \in K[X]$ with $\deg(f) < n$ and $a = (a_0, \ldots, a_{n-1}) \in K^n$, compute the multipoint evaluation $f(a_0), \ldots, f(a_{n-1}) \in K$

Interpolation Given $a = (a_0, \ldots, a_{n-1}) \in K^n$ with $a_i \neq a_j$ for $i \neq j$, and $b = (b_0, \ldots, b_{n-1}) \in K^n$, compute $f \in K[X]$ with $\deg(f) < n$ such that $f(a_i) = b_i$ for all i.