Polynomial Composition: How Fast Can We Do It?

Ye Kai

December, 2024

Abstract

In this survey, I have reviewed several algorithms for polynomial composition over a commutative ring $\mathbb{A}$. Here we suppose $\deg F < m, \deg G < n$ and $\mathsf{M}(n)$ is the time for calculate two polynomials with length n. We want to calculate $F(G(z)) \bmod z^n$. Firstly it's Naive algorithm that directly computes the answer in $O(m\mathsf{M}(n))$ or $O(n\mathsf{M}(n) + \mathsf{M}(m) \log m)$ time. Then it's Chunking that optimizes the Naive algorithm to $O(mn + \sqrt{m}\mathsf{M}(n))$ or $O(\sqrt{n}\mathsf{M}(n) + n^2 + \mathsf{M}(m) \log m)$ time. Brent-Kung algorithm in 1978 [3] achieve the time complexity of $O(\mathsf{M}(m)+\mathsf{M}(n)(\min\{n, m\} \log n)^{1/2})$, but it needs $1, 2, \cdots, n$ to be invertible over $\mathbb{A}$. Umans algorithm in 2008 [12] achieve $O(n^{1+o(1)})$ when $\mathbb{A}$ is a field with a small characteristic and $m = n$, and this idea is extended to $\mathbb{A} = \mathbb{F}_q$ in 2011 [4]. Kinoshita-Li algorithm in 2024 [6] is the SOTA of this problem now, with a time complexity of $O(\mathsf{M}(n) \log \min\{n, m\} + \mathsf{M}(m))$. After describing the algorithms, experiment for $\mathbb{A} = \mathbb{F}_{998244353}$ is done to compare the efficiency for some of the algorithms. The result shows that Kinoshita-Li is always the best, and Brent-Kung is slower than Chunking in the practical cases.

Contents

1 Introduction & Related Work

Suppose $F, G \in \mathbb{A}[z]$, where $\mathbb{A}$ is a commutative ring. The problem of polynomial composition, also known as power series composition in some contexts, involves calculating the composition of two polynomials $F(z) = \sum_{j=0}^{m-1} f_j z^j$ and $G(z) = \sum_{j=0}^{n-1} g_j z^j$ modulo z^n. Specifically, we seek to compute

$$F(G(z)) \bmod z^n = \sum_{j=0}^{m-1} f_j (G(z))^j \bmod z^n$$

A fundamental question is: how efficiently can we solve this problem?

One classic approach is the Brent-Kung algorithm [3] in 1978, which offers a time complexity of $O(\mathsf{M}(m) + \mathsf{M}(n)(\min\{n, m\} \log n)^{1/2})$, assuming polynomial multiplication in $\mathbb{A}[z]$ with degree less then n can be performed in $\mathsf{M}(n)$ time. This algorithm relies on the Taylor formula and requires that elements $1, 2, \ldots, n-1$ be invertible in $\mathbb{A}$.

In 2008, C.Umans introduced a novel algorithm [12] with a time complexity of $O(n^{1+o(1)})$ for certain cases, which was later extended to $\mathbb{A} = \mathbb{F}_q$ by K.S.Kedlaya and C.Umans [4] [5]. The algorithm's performance was further analyzed by Hoeven and Lecerf [13] in 2020, who showed a complexity of $O(n 2^{O(\sqrt{\log n \log\log n})} \log q)$.

Most recently, in 2024, Yasunori Kinoshita and Baitian Li presented a surprisingly simple algorithm [6] with a complexity of $O(\mathsf{M}(n) \log \min\{n, m\} + \mathsf{M}(m))$, based on Graeffe's method and bivariate rational polynomials.

Whether this problem can achieve its $\Omega(\mathsf{M}(n) + \mathsf{M}(m))$ tail bound is still open. If so, some problems like $\exp$ and $\ln$, which currently have $O(\mathsf{M}(n))$ algorithms, would have a uniform algorithm.

2 Preliminaries

2.1 Notation

In this survey, we always discuss under a commutative ring $\mathbb{A}$ that basic ring operation $(+, -, \times)$ takes constant time. We define the polynomial ring over $\mathbb{A}$ as $\mathbb{A}[z]$. For polynomial $F(z) = \sum_j f_j z^j$, we define $[z^t]F(z) = f_t$.

2.2 Basic Polynomial Operations

We need always to calculate some basic polynomial operations: modulo by z^n, addition, multication by number, multication in polynomials.

Modulo by z^n looks like $G(z) \leftarrow F(z) \bmod z^n$. This can be done in $O(\min\{\deg F, n\})$.

Addition looks like $H(z) \leftarrow F(z) + G(z)$. This can be done in $O(\deg F + \deg G)$.

Multication by number looks like $G(z) \leftarrow cF(z)$. This can be done in $O(\deg F)$.

Multication in polynomials looks like $H(z) \leftarrow F(z)G(z)$. This can be done in $O(\deg F \times \deg G)$ or $O(\mathsf{M}(\max\{\deg F, \deg G\}))$.

Here $\mathsf{M}(n)$ stands for the time limit we need to calculate two polynomials in length n. $\mathsf{M}(n)$ is always between $\Omega(n)$ and $O(n^2)$. In some fields, like $\mathbb{R}$, $\mathsf{M}(n) = O(n \log n)$ by FFT [10].

2.3 Polynomial Shift

Polynomial shift is a special case of polynomial composition: when $G(z) = z + a$, we need to calculate $F(z + a) \bmod z^n$.

A simple solution by D&C have a time complexity of $\Omega(\mathsf{M}(n)) \sim O(\mathsf{M}(n) \log n)$.

If $1, 2, \cdots, k - 1$ are invertible over $\mathbb{A}$, then

$$\begin{aligned} F(z+a) &= \sum_j f_j (z+a)^j \\ &= \sum_k k! z^k \sum_j \frac{f_j}{j!} \frac{a^{k-j}}{(k-j)!} \end{aligned}$$

So it can be computed by multication in $O(\mathsf{M}(n))$ time.

The meaning of polynomial shift is $F(G(z)) \bmod z^n = (F(z + g_0) \circ (G(z) - g_0)) \bmod z^n = ((F(z + g_0) \bmod z^n) \circ (G(z) - g_0)) \bmod z^n$.

That is, we can make sure $g_0 = 0$ and $m \le n$ after a polynomial shift.

So if $n \lll m$ would lead to a bad time complexity in an algorithm, we can use this idea.

3 Begin With Naive Algorithm

We have the following basic Naive algorithm:

1. Calculate $G^0 \bmod z^n, G^1 \bmod z^n, \cdots, G^{m-1} \bmod z^n$ by $G^k \bmod z^n = (G \times (G^{k-1} \bmod z^n)) \bmod z^n$.
2. Calculate $\sum_{j=0}^{m-1} f_j G^j \bmod z^n$.

The first step uses m times of multication with length n, so its time complexity is $O(m\mathsf{M}(n))$; and the second part is $O(mn)$ as it has m times of addition in length n. As $\mathsf{M}(n)$ is always $\Omega(n)$, the total complexity is $O(m\mathsf{M}(n))$ here.

In fact Horner's rule gives out a similar algorithm, with the same time complexity.

By using the polynomial shift in section 2.3, it's time complexity would become $O(n\mathsf{M}(n) + \mathsf{M}(m) \log m)$ when $n \lll m$.

4 A Better Naive Algorithm By Chunking

4.1 Basic Idea & Algorithm Steps

Can we try to optimize the Naive algorithm?

I find an idea from [11] (in Chinese). Suppose here's a block size B, then

$$\begin{aligned} F(G(z)) \bmod z^n &= \sum_{i=0}^{m-1} f_i G^i(z) \bmod z^n \\ &= \left(\sum_{i=0}^{\lfloor (m-1)/B \rfloor} (G^{iB} \bmod z^n) \sum_{j=0}^{\min\{B, m-iB\}-1} f_{iB+j} (G^j \bmod z^n) \right) \bmod z^n \end{aligned}$$

So we can have the following algorithm:

1. Calculate $G^0 \bmod z^n, G^1 \bmod z^n, \cdots, G^B \bmod z^n$ by $G^k \bmod z^n = (G \times (G^{k-1} \bmod z^n)) \bmod z^n$.
2. Calculate $G^0 \bmod z^n, G^B \bmod z^n, \cdots, G^{B\lfloor (m-1)/B \rfloor} \bmod z^n$ similarly.
3. Calculate $H_i(z) = \sum_{j=0}^{\min\{B, m-iB\}-1} f_{iB+j}(G^j \bmod z^n)$ by naive algorithm.
4. Calculate $F(G(z)) \bmod z^n = \left(\sum_{i=0}^{\lfloor (m-1)/B \rfloor} (G^{iB} \bmod z^n) H_i(z)\right) \bmod z^n$.

4.2 Time Analysis & Time Complexity

Time analysis for each part:

1. It uses B times of multication with length n, so it's $O(B\mathsf{M}(n))$.
2. Similarly it's $O(\frac{m}{B}\mathsf{M}(n))$.
3. It uses $O(nm)$ times of multication and addition.
4. It uses $O(m/B)$ times of multication and addition with length n, so it's $O(\frac{m}{B}\mathsf{M}(n))$.

Eventually the total time is $O((B + \frac{m}{B})\mathsf{M}(n) + nm)$.
So if we take $B = \Theta(\sqrt{m})$, the total complexity is $O(\sqrt{m}\mathsf{M}(n) + nm)$.
By using the polynomial shift in section 2.3, it's time complexity would become $O(\sqrt{n}\mathsf{M}(n) + n^2 + \mathsf{M}(m)\log m)$ when $n \lll m$.

5 Brent-Kung Algorithm

5.1 Introduction

In 1978, R.P.Brent and H.T.Kung gives a better algorithm [3]. This algorithm is based on Taylor formula and divide & conquer.

This algorithm uses the idea of BSGS, so its time complexity has a factor of $\sqrt{m}$.

The following gives a simple description for this algorithm by me, so it's a little bit different from the origin paper. I refer to the online article [14] (in Chinese) for this description.

5.2 Basic Idea

In this algorithm, $1, 2, \ldots, n-1$ should be invertible over $\mathbb{A}$.

It points out that, if we take $G_p(z) = \sum_{i=0}^{B-1} g_i z^i$ and $G_r(z) = G(z) - G_p(z)$, we'd have

$$\begin{aligned} F(G(z)) \bmod z^n &= F(G_p(z) + G_r(z)) \bmod z^n \\ &= \left(\sum_{j=0}^{\lfloor (n-1)/B \rfloor} \frac{F^{(j)}(G_p(z))}{j!} G_r^j(z)\right) \bmod z^n \\ &= \sum_{j=0}^{\lfloor (n-1)/B \rfloor} \left(\frac{F^{(j)}(G_p(z)) \bmod z^n}{j!} (G_r^j(z) \bmod z^n)\right) \bmod z^n \end{aligned}$$

Here $F^{(j)}(z)$ stands for the j-th order derivative of $F(z)$.

As $(F^{(j)}(G_p(z)))' = F^{(j+1)}(G_p(z)) \times G'_p(z)$, if we had known $F^{(j+1)}(G_p(z)) \bmod z^n$, we can get $(F^{(j)}(G_p(z)))' \bmod z^n$ in $O(\mathsf{M}(n))$ and then $F^{(j)}(G_p(z)) \bmod z^n$ in $O(n+m)$ (we can get $[z^0]F^{(j)}(G(z)) = F^{(j)}(g_0)$ in $O(m)$). Each $G_r^j \bmod z^n$ can be computed in $O(\mathsf{M}(n))$ in the same time.

So, if we had known $F^{(\lfloor (n-1)/B \rfloor)}(G_p(z)) \bmod z^n$, we can compute the answer in $O(\frac{n}{B}(m + \mathsf{M}(n)))$.

5.3 Divide & Conquer

Now we need only to get $F^{(\lfloor (n-1)/B \rfloor)}(G_p(z)) \bmod z^n$.

Divide & Conquer: for $H(z) = \sum_{j=0}^{m-1} h_j z^j$ that $\log_2 m \in \mathbb{N}$, we have $H(G(z)) \bmod z^n = H_1(G(z)) \bmod z^n + (G^{m/2}(z) \bmod z^n)(H_2(G(z)) \bmod z^n) \bmod z^n$. Here we take $H_1(z) = \sum_{j=0}^{m/2-1} h_j z^j$ and $H_2(z) = \sum_{j=0}^{m/2-1} h_{j+m/2} z^j$.

We can pre-calculate each $G^{2^t}(z) \bmod z^n$ in $O(\sum_{j=0}^{\log_2 m} \mathsf{M}(\min\{2^j B, n\}))$, which takes time of $O(\mathsf{M}(mB))$ when $mB < n$ and $O(\log_2(mB/n)\mathsf{M}(n))$ when $mB \geq n$.

And suppose this D&C has time complexity of $\mathsf{T}(m)$, then

$$\begin{aligned} \mathsf{T}(m) &= 2\mathsf{T}(m/2) + O(\mathsf{M}(\min(n, mB))) \\ &= \begin{cases} O(\sum_{j=0}^{\log_2 m} \frac{m}{2^j} \mathsf{M}(2^j B)) & mB < n \\ O(\frac{mB}{n} \sum_{j=0}^{\log_2(n/B)} \frac{n}{2^j B} \mathsf{M}(2^j B)) & mB \geq n \end{cases} \end{aligned}$$

So it's always $O(\frac{mB \log n}{n} \mathsf{M}(n))$ if $mB \geq n$.

5.4 Time Complexity

If we take $B = \Theta(n/\sqrt{m \log n})$, the total time complexity would be $O((m + \mathsf{M}(n))\sqrt{m \log n})$.

It'd be slow if $n \lll m$. By using the polynomial shift in section 2.3, it'd just become $O(\mathsf{M}(m) + \mathsf{M}(n)(\min\{n, m\} \log n)^{1/2})$.

5.5 Algorithm Steps & Time Analysis

Anyway, this algorithm can be done in following way:

1. $F(z) \leftarrow F(z + g_0) \bmod z^n$ and $G(z) \leftarrow G(z) - g_0$. It takes $O(\mathsf{M}(m))$ time, and the answer won't change. $m \leftarrow \min\{m, n\}$ after this.
2. Take $B = \lceil n/\sqrt{m(1 + \log_2 n)} \rceil$ in $O(1)$.
3. Take $G_p(z) = \sum_{j=0}^{B-1} g_j z^j$ and $G_r(z) = \sum_{j=B}^{n-1} g_j z^j$ in $O(n)$.
4. Calculate $F^{(\lfloor (n-1)/B \rfloor)}$ in $O(n)$.
5. Calculate $F^{(\lfloor (n-1)/B \rfloor)}(G_p) \bmod z^n$ by D&C in $O(\frac{m}{n} B \log n \mathsf{M}(n))$.
6. Calculate each $F^{(j)}(G_p) \bmod z^n$. It takes $O(\frac{n}{B}\mathsf{M}(n))$ in total.
7. Calculate each $\frac{1}{j!}\Big((F^{(j)}(G_p(z)) \bmod z^n)(G_r^j(z) \bmod z^n)\Big) \bmod z^n$. It takes $O(\frac{n}{B}\mathsf{M}(n))$ in total.
8. Sum them together in $O(\frac{n^2}{B})$. We get the answer.

The D&C part has been talked in section 5.3.

6 Umans Algorithm, and Kedlaya-Umans Algorithm

C.Umans finds such a fact in paper [12]: "Our insight is that this modular composition problem and the multipoint evaluation problem for multivariate polynomials are essentially equivalent in the sense that an algorithm for one achieving exponent α implies an algorithm for the other with exponent $\alpha + o(1)$, and vice versa". In fact, he gives out a $O(n^{1+o(1)})$ algorithm to calculate "multipoint evaluation of multivariate polynomials in small characteristic" in this paper.

K.S.Kedlaya and C.Umans extended this idea to finite field $\mathbb{F}_q$ within $O(n^{1+o(1)})$ time. [4] [5]

I don't decide to describe the technical part of these algorithms, as they're complex actually. One description can be found in Baitian Li's blog [8] (in Chinese).

7 Kinoshita-Li Algorithm

7.1 Introduction

It is the SOTA of this problem now. This algorithm is from Yasunori Kinoshita and Baitian Li in 2024 [6]. This algorithm is based on Graeffe's method and bivariate rational polynomial.

The following gives a simple description for this algorithm by me, so it's a little bit different from the origin paper. I refer to the online article [9] (in Chinese) for this description.

As this algorithm needs bivariate polynomial, we'd use the notations similar to single variable polynomial here.

7.2 Preknowledge

7.2.1 Transposition Principle

Transposition Principle is told by A.Bostan, G.Lecerf and E.Schost in 2003 [1].

Suppose here's a matrix $P_{n\times n} = (p_{ij})$. For any a vector $x = (x_1, x_2, \cdots, x_n)'$, we can calculate vector Px by linear algorithm (only apply elementary row transformation on x) in $\mathsf{T}(n)$ time. Then we can compute $P^T x$ in $\mathsf{T}(n) + O(n)$.

Why? It's because we need only to apply the transposition of each step in the inverse order to realize the answer, which we called as the transposed algorithm of the origin algorithm.

And this theorem is called as transposition principle.

Transposition principle means that $y_i = \sum_j P_{ij} x_j$ is as hard as $y_i = \sum_j P_{ji} x_j$. So if we want to compute $y_i = \sum_j P_{ji} x_j$, we need only to think about $y_i = \sum_j P_{ij} x_j$.

7.2.2 Polynomial Inversion

We define polynomials $F(z), G(z)$ over $\mathbb{A}$ with degree less than n are inversion(or reciprocal) of each other under mod z^n, when and only when $F(z)G(z) \bmod z^n = 1$.

If $[z^0]F = 1$, we can calculate $G(z) = F(z)^{-1} \bmod z^n$ in $O(\mathsf{M}(n))$. [7]

7.2.3 Bivariate Polynomial Multication

Here are bivariate polynomials $F(z,u), G(z,u)$ with $\deg_z < n$ and $\deg_u < m$. Then we can calculate $F(z,u)G(z,u)$ in $O(\mathsf{M}(nm))$.

Why? Because we can just turn each $z^a u^b$ into z^{2am+b} and do multication of length $2nm$ directly.

7.2.4 Transposed Multication

As the multication is a special part during the linear algorithm that we can't decompose to elementary matrix easily, let's think of a way to compute the transposed multication.

Now here's $F(z) = \sum_{j=0}^{n-1} f_j z^j$ and $G(z) = \sum_{j=0}^{n-1} g_j z^j$.

If in the origin algorithm, we have

$$F(z) \leftarrow F(z)G(z) = \sum_j z^j \sum_i f_i g_{j-i}$$

Then in the transposed algorithm, we need to transpose g, which gives

$$F(z) \leftarrow \sum_j z^j \sum_i f_i g_{i-j}$$

Suppose $H(z) = \sum_j f_{n-j-1} z^j$, then it's just $[z^j]F(z) \leftarrow [z^{n-j-1}]H(z)G(z)$.

So in the transposed algorithm, we can do the transposed multication in $O(\mathsf{M}(n))$ still easily.

We would call it as $F(z) \leftarrow F(z) \otimes G(z)$ since now.

7.3 Basic Idea

The basic idea for this problem is transposition principle: as $[z^k]F(G(z)) = \sum_{j=0}^{m-1} f_j [z^k](G(z))^j$, we need only to design an algorithm to calculate

$$h_k = \sum_{j=0}^{n-1} f_j [z^j](G(z))^k = [z^{n-1}u^k]\frac{F_T(z)}{1 - uG(z)}$$

Here $F_T(z) = \sum_{j=0}^{n-1} f_{n-j-1} z^j$. We can find that we need only to calculate

$$[z^{n-1}]\frac{F_T(z)}{1 - uG(z)} \bmod u^m$$

now.

7.4 Graeffe's method

We need to get one coefficient of rational polynomial about z now. The Bostan-Mori [2] sounds like a possible solution for such a problem. In fact, the following part is a generalization of the Graeffe's method in Bostan-Mori algorithm.

$$[z^n]\frac{F(u,z)}{G(u,z)} \bmod u^m = [z^n]\frac{F(u,z)G(u,-z)}{G(u,z)G(u,-z)} \bmod u^m$$

As $G(u,z)G(u,-z)$ has coefficients only on z^{2k}, we need only to leave the coefficients of $z^{2k+[2\neq m]}$ in $F(u,z)G(u,-z)$. Then we can turn $n \to \lfloor n/2 \rfloor$.

Eventually it looks like $[z^0](F(u,z)/G(u,z)) \bmod u^m$, which equals to $((F(u) \bmod u^m) \times (G^{-1}(u) \bmod u^m)) \bmod u^m$ in fact.

7.5 Transposed Algorithm

Now we need to give out a transposed algorithm, so that we can solve the origin problem.

Firstly we need to calculate each $G(u, z)$ during the Graeffe's method by the origin algorithm, and the $G^{-1}(u)$ eventually.

Now we need to transpose the part about F.

First $F(z) \leftarrow F(z) \otimes G(z)^{-1}$. Then $F(z, u) \leftarrow F(z, u) \otimes G(z, u)$ in the inverse order each time: this $\otimes$ can just be done one the single variate polynomials transformed from bivariate polynomials.

Eventually we need only to reverse the F, which would become the answer.

7.6 Time Analysis & Time Complexity

We can find that after k times, we'd have $\deg_u G(u, z) \leq \min\{2^k, m\}$ and $\deg_z G(u, z) \leq n/2^k$. So we need only to do the bivariate polynomial multication over $\min\{2^k, m\} \times (n/2^k)$ each time. After this, we need to calculate $(F(u)/G(u)) \bmod u^m$, which can be done in $O(\mathsf{M}(m))$.

So the total time is

$$O(\mathsf{M}(m)) + \sum_{k=0}^{\lfloor \log_2 n \rfloor} O(\mathsf{M}(\min\{2^k, m\}) \times (n/2^k)) = O(\mathsf{M}(n) \log \min\{n, m\} + \mathsf{M}(m))$$

So this algorithm would only take $O(\mathsf{M}(n) \log \min\{n, m\} + \mathsf{M}(m))$ time.

We can find that this algorithm need not the optimization of polynomial shift in section 2.3.

8 Experiment

In order to know the actual efficiency of these algorithms, I realize some of the algorithms and test them.

8.1 Platform & Data & Setting

I select Luogu Online Judge as the platform for testing the CPP codes with O2 optimization.

On that platform, this problem can be seen as compute $F(G(z)) \bmod z^{n+1}$ over $\mathbb{A} = \mathbb{F}_p$ that $p = 998244353 = 7 \times 17 \times 2^{23} + 1$. It keeps $\deg F = \deg G = n$ in fact.

It includes data that $n \in \{20000, 30000, 50000, 100000, 150000, 200000\}$.

We'd take the average time of the tests for each n. The cases for $n = 20000$ can only be tested to $4s$ limit, while others are $10s$.

In such cases, we can see $\mathsf{M}(n) = O(n \log n)$.

8.2 Coding Versions

The following algorithms would be tested: Naive algorithm (Section 3), Chunking (Section 4), Brent-Kung Algorithm (Section 5), Kinoshita-Li Algorithm (Section 7).

Each algorithm here has 3 versions when coding:

1. Calculate over $\mathbb{F}_p$ directly, and always take the polynomial multication as encapsulation.

2. Calculate over $\mathbb{F}_p$ directly, and may think about the multication from the NTT vision.

3. Can delay some modulo, and may think about the multication from the NTT vision.

Algorithm	Naive	Chunking	Brent-Kung	Kinoshita-Li
$n = 20000$	$> 4s$	$2.77s$	$> 4s$	$408ms$
$n = 30000$	$> 10s$	$4.88s$	$4.87s$	$538ms$
$n = 50000$	$> 10s$	$> 10s$	$> 10s$	$1.15s$
$n = 100000$	$> 10s$	$> 10s$	$> 10s$	$3.08s$
$n = 150000$	$> 10s$	$> 10s$	$> 10s$	$7.26s$
$n = 200000$	$> 10s$	$> 10s$	$> 10s$	$7.50s$

Table 1: efficiency for the 1st version of codes

Algorithm	Naive	Chunking	Brent-Kung	Kinoshita-Li
$n = 20000$	$> 4s$	$1.83s$	$> 4s$	$357ms$
$n = 30000$	$> 10s$	$2.94s$	$5.56s$	$522ms$
$n = 50000$	$> 10s$	$8.31s$	$> 10s$	$1.11s$
$n = 100000$	$> 10s$	$> 10s$	$> 10s$	$2.86s$
$n = 150000$	$> 10s$	$> 10s$	$> 10s$	$6.23s$
$n = 200000$	$> 10s$	$> 10s$	$> 10s$	$6.92s$

Table 2: efficiency for the 2nd version of codes

We can find that the 2nd and 3rd version may give some optimization to the algorithm.

- e.g. if we need to calculate $F \times G_1, F \times G_2, \cdots, F \times G_n$ that $\deg F \leq n \wedge \deg G_j \leq n$, we need only to give DFT to F once.
- e.g. The less modulo would make the code faster, as modulo is slow.

8.3 Experiment Result & Result Analysis

The Table 1, Table 2, Table 3 show the efficiency of my codes under the 1st, 2nd, 3rd versions.

It shows that

1. Naive is always the worst, and Kinoshita-Li is always the best.
2. Brent-Kung never runs better than Chunking when they both pass the time limit, even through it has a better time complexity.
3. The 2nd version of Brent-Kung is not very good, which is even slower than its 1st version.

Algorithm	Naive	Chunking	Brent-Kung	Kinoshita-Li
$n = 20000$	$> 4s$	$1.22s$	$3.22s$	$270ms$
$n = 30000$	$> 10s$	$1.98s$	$4.01s$	$400ms$
$n = 50000$	$> 10s$	$5.59s$	$> 10s$	$848ms$
$n = 100000$	$> 10s$	$> 10s$	$> 10s$	$2.25s$
$n = 150000$	$> 10s$	$> 10s$	$> 10s$	$5.07s$
$n = 200000$	$> 10s$	$> 10s$	$> 10s$	$5.56s$

Table 3: efficiency for the 3rd version of codes

9 Conclusion

In this survey, we have reviewed several algorithms for polynomial composition over a commutative ring $\mathbb{A}$.

We started with the basic Naive algorithm, which calculates $F(G(z)) \bmod z^n$ by directly calculating $G(z)^k \bmod z^n$ and then multiplying the results with each $[z^k]F(z)$. This method has a time complexity of $O(m\mathsf{M}(n))$, and can be optimized to $O(n\mathsf{M}(n) + \mathsf{M}(m)\log m)$ by polynomial shift.

By Chunking, we can optimize this Naive algorithm to $O(\sqrt{m}\mathsf{M}(n) + nm)$. When $n \lll m$, this algorithm can also be optimized to $O(\sqrt{n}\mathsf{M}(n) + n^2 + \mathsf{M}(m)\log m)$ by polynomial shift.

The Brent-Kung algorithm introduced in 1978 [3] uses Taylor series and D&C algorithm to achieve a time complexity of $O(\mathsf{M}(m) + \mathsf{M}(n)(\min\{n, m\}\log n)^{1/2})$. But this algorithm needs $1, 2, \cdots, n$ to be invertible over $\mathbb{A}$.

The Umans algorithm introduced in 2008 [12] provides an algorithm with a time complexity of $O(n^{1+o(1)})$ for the case where $\mathbb{A}$ is a field with a small characteristic. This algorithm is based on the equivalence between the polynomial modular composition and the multivariate multipoint evaluation. The Kedlaya-Umans algorithm extends this idea to the finite field $\mathbb{F}_q$. We don't describe this algorithm in detail.

The Kinoshita-Li algorithm introduced in 2024 [6] is the SOTA for this problem. It has a time complexity of $O(\mathsf{M}(n)\log\min\{n, m\} + \mathsf{M}(m))$. It's based on Graeffe's method and bivariate rational polynomials.

From the experimental results, we can see that the Naive algorithm is always the slowest, while the Kinoshita-Li algorithm is consistently the fastest. And the Brent-Kung algorithm is never faster than the Chunking algorithm.

The future work may focus on whether the problem can archive the theoretical tail bound $\Omega(\mathsf{M}(n) + \mathsf{M}(m))$.

References

[1] A. Bostan, G. Lecerf, and É. Schost. Tellegen's principle into practice. In Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation, ISSAC '03, page 37–44, New York, NY, USA, 2003. Association for Computing Machinery.

[2] Alin Bostan and Ryuhei Mori. A simple and fast algorithm for computing the n-th term of a linearly recurrent sequence, 2020.

[3] Richard P. Brent and H. T. Kung. Fast algorithms for manipulating formal power series. J. ACM, 25:581–595, 1978.

[4] Kiran S. Kedlaya and Christopher Umans. Fast modular composition in any characteristic. In 2008 49th Annual IEEE Symposium on Foundations of Computer Science, pages 146–155, 2008.

[5] Kiran S. Kedlaya and Christopher Umans. Fast polynomial factorization and modular composition. SIAM Journal on Computing, 40(6):1767–1802, 2011.

[6] Yasunori Kinoshita and Baitian Li. Power series composition in near-linear time, 2024.

[7] H. T. Kung. On computing reciprocals of power series. Numer. Math., 22(5):341–348, October 1974.

[8] Baitian Li. Almost linear algorithm for polynoimial modular composition, and data structure supporting multivariate multipoint evaluation. `https://www.cnblogs.com/Elegia/p/multivariate-evaluation.html`. (2023, June 23).

[9] Zhuocheng Lin. Solution for p5373 in luogu. `https://www.luogu.com/article/7joh5isi`. (2024, May 17).

[10] Harold S. Stone. R66-50 an algorithm for the machine calculation of complex fourier series. IEEE Transactions on Electronic Computers, EC-15(4):680–681, 1966.

[11] Ruize Sun. Solution for p5373 in luogu. `https://www.luogu.com/article/02jn7dne`. (2019, May 15).

[12] Christopher Umans. Fast polynomial factorization and modular composition in small characteristic. In Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing, STOC '08, page 481–490, New York, NY, USA, 2008. Association for Computing Machinery.

[13] Joris van der Hoeven and Grégoire Lecerf. Fast multivariate multi-point evaluation revisited. J. Complex., 56, 2020.

[14] yurzhang. Solution for p5373 in luogu. `https://www.luogu.com/article/tlu60zwy`. (2019, June 2).