The fast transposed Vandermonde solver and its implementation in C

by

Hyukho Kwon

B.Sc., Simon Fraser University, 2020

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Mathematics
Faculty of Science

SIMON FRASER UNIVERSITY
Spring 2024

Declaration of Committee

Name: **Hyukho Kwon**

Degree: **Master of Science**

Thesis title: **The fast transposed Vandermonde solver and its implementation in C**

Committee: **Chair:** Luis Goddyn
Professor, Mathematics

Michael Monagan
Supervisor
Professor, Mathematics

Amarpreet Rattan
Committee Member
Associate Professor, Mathematics

Marni Mishna
Examiner
Professor, Mathematics

Abstract

We study the algorithm of Kaltofen and Yagati [8], which solves an $n \times n$ transposed Vandermonde system of linear equations over a field F. This algorithm does $O(n \log^2 n)$ arithmetic operations in F. It assumes that the fast Fourier transform (FFT) is used for polynomial multiplication, polynomial division, and polynomial multipoint evaluation over F. We implemented this fast transposed Vandermonde solver in C. It uses the "product tree" of Borodin and Munro [2]. In our C implementation, we optimize the fast multiplication, the fast division, and the fast evaluation algorithms. To speed up fast division, we use Hanrot, Quercia, and Zimmerman's "middle product" [6]. Our fast solver beats Zippel's $O(n^2)$ solver [15] when $n \geq 128$ and $F = \mathbb{Z}_p$ with $p < 2^{63}$.

Keywords: transposed Vandermonde system; fast Fourier transform; polynomial multipoint evaluation; polynomial division; bit-reversal permutation

Dedication

For the two ladies I love: Byeongnam, my paternal grandmother, who raised her children and took care of her father-in-law by herself and Chunja, my maternal grandmother, who did not have the opportunity to showcase or develop her brilliant talents due to her gender — "History has failed us, but no matter."

Acknowledgements

I would like to thank my supervisor, Michael Monagan. Without his ceaseless support and constructive feedback, I would not have been able to face challenges and explore new opportunities at Simon Fraser University.

Also, I would like to thank Mahsa Ansari and Sophie Hoare for being my computer algebra buddies. It was great fun to take the Topics in Computer Algebra course together.

I would like to thank Eric Lam for being my friend, who likes my cynical jokes. You are the only one I can always share my honest feelings in Vancouver.

I would like to thank Bryce Haley for enriching my undergraduate life at Simon Fraser University. Without you, I could not get out of my comfort zone.

I would like to thank my parents, Oh-hyung and Misook, for their emotional support. Your limitless love always makes me strong enough to stay in Canada by myself.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Let F be a field. Let $a \in F[x]$ be a polynomial where $a(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$. For $u_1, u_2, \ldots, u_n \in F$, let $b_i = a(u_i)$ for $1 \leq i \leq n$. The polynomial interpolation problem is to find $a(x)$ given $(u_i$,$b_i)$ for $1 \leq i \leq n$. The following linear system of equations can be used to solve this problem.

$$\begin{aligned} b_1 = a(u_1) &= a_0 + a_1 u_1 + a_2 u_1^2 + \cdots + a_{n-1} u_1^{n-1} \\ b_2 = a(u_2) &= a_0 + a_1 u_2 + a_2 u_2^2 + \cdots + a_{n-1} u_2^{n-1} \\ &\vdots \\ b_n = a(u_n) &= a_0 + a_1 u_n + a_2 u_n^2 + \cdots + a_{n-1} u_n^{n-1} \end{aligned}$$

We can rewrite this in matrix representation:

$$\underbrace{\begin{bmatrix} 1 & u_1 & \cdots & u_1^{n-1} \\ 1 & u_2 & \cdots & u_2^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & u_n & \cdots & u_n^{n-1} \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}}_{\mathbf{b}}.$$

The matrix V is called a Vandermonde matrix and $V\mathbf{a} = \mathbf{b}$ is called a Vandermonde system of equations. It is known that $\det(V) = \prod_{1 \leq i < j \leq n}(u_i - u_j)$. Thus, if $u_1, u_2, \ldots, u_n$ are distinct, $V\mathbf{a} = \mathbf{b}$ has a unique solution.

Suppose we use Gaussian elimination to solve $V\mathbf{a} = \mathbf{b}$. In that case, we do $O(n^3)$ arithmetic operations in F and need space for $O(n^2)$ elements of F. Alternatively, Lagrange interpolation or Newton interpolation can be used to solve $V\mathbf{a} = \mathbf{b}$. Both methods do $O(n^2)$ arithmetic operations in F and need space for $O(n)$ elements of F. To speed up the evaluation of $a(u_i)$, various algorithms, including the fast Fourier transform (FFT), use geometric point sequences $u_i = \alpha^{i-1}$ for some $\alpha \in F$.

Let $u_i = \alpha^{i-1}$ where $\alpha^i \neq \alpha^j$ for all $i \neq j$. Then $a(u_i) = a(\alpha^{i-1}) = b_i$. It follows that, for $1 \leq i \leq n$,

$$b_i = a(u_i) = a(\alpha^{i-1}) = \sum_{j=0}^{n-1} a_j(\alpha^{i-1})^j = \sum_{j=0}^{n-1} a_j(\alpha^j)^{i-1} = \sum_{j=0}^{n-1} a_j(u_{j+1})^{i-1}.$$

That is

$$\begin{aligned}
b_1 &= a(u_1) = a_0 + a_1 + a_2 + \cdots + a_{n-1} \\
b_2 &= a(u_2) = a_0u_1 + a_1u_2 + a_2u_3 + \cdots + a_{n-1}u_n \\
b_3 &= a(u_3) = a_0u_1^2 + a_1u_2^2 + a_2u_3^2 + \cdots + a_{n-1}u_n^2 \\
&\vdots \\
b_n &= a(u_n) = a_0u_1^{n-1} + a_1u_2^{n-1} + a_2u_3^{n-1} + \cdots + a_{n-1}u_n^{n-1}
\end{aligned}$$

We can express this system of linear equations in matrix-vector form:

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ u_1 & u_2 & u_3 & \cdots & u_n \\ u_1^2 & u_2^2 & u_3^2 & \cdots & u_n^2 \\ \vdots & \vdots & \vdots & & \vdots \\ u_1^{n-1} & u_2^{n-1} & u_3^{n-1} & \cdots & u_n^{n-1} \end{bmatrix}}_{U} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}}_{\mathbf{b}}.$$

The matrix U is called a transposed Vandermonde matrix since $U = V^\top$. Also, $U\mathbf{a} = \mathbf{b}$ is called a transposed Vandermonde system of equations. Transposed Vandermonde systems of equations arise in sparse interpolation algorithms in computer algebra. Ben-Or and Tiwari's deterministic sparse interpolation algorithm [1] and Zippel's probabilistic sparse interpolation algorithm [15] have to solve transposed Vandermonde systems of equations. Hu and Monagan's sparse polynomial greatest common divisor (GCD) algorithm [7] needs to solve the transposed Vandermonde system of equations as well. The $n \times n$ transposed Vandermonde system is used to interpolate a sparse polynomial with n terms in all three algorithms.

In this work, we look at how Ben-Or and Tiwari algorithm creates a transposed Vandermonde system of equations. Let $f = \sum_{i=1}^{t} a_iM_i(x_1, x_2, \ldots, x_n) \in F[x_1, x_2, \ldots, x_n]$ be a sparse multivariate polynomial where $a_i \in F$ and $M_i(x_1, x_2, \ldots, x_n)$ is a monomial of f for

$1 \leq i \leq t$. Given a term bound $T \geq t$, Ben-Or and Tiwari algorithm uses the first n primes $2, 3, 5, \ldots, p_n$ to compute

$$b_j = f(2^j, 3^j, 5^j, \ldots, p_n^j) \text{ for } 0 \leq j \leq 2T - 1.$$

Let $m_i = M_i(2, 3, 5, \ldots, p_n)$. Ben-Or and Tiwari algorithm has two main steps. In the first step, it determines the m_i from the b_j. In the second step, it determines the M_i and a_i. By the fundamental theorem of arithmetic, every m_i is distinct. If $M_i = x_1^{d_1} x_2^{d_2} \cdots x_n^{d_n}$, then $m_i = 2^{d_1} 3^{d_2} \cdots p_n^{d_n}$. We can obtain $d_1, d_2, \ldots, d_n$ by dividing m_i by 2,3,5, etc.

Example 1.1. Suppose $n = 3$. If $m_1 = 60 = 2^2 \cdot 3 \cdot 5$, then $M_1(x_1, x_2, x_3) = x_1^2 x_2 x_3$.

Once all M_is are found, Ben-Or and Tiwari algorithm solves the $t \times t$ transposed Vandermonde system of equations for the unknown coefficients $a_1, a_2, \ldots, a_t$:

$$\underset{U}{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ m_1 & m_2 & m_3 & \cdots & m_t \\ m_1^2 & m_2^2 & m_3^2 & \cdots & m_t^2 \\ \vdots & \vdots & \vdots & & \vdots \\ m_1^{t-1} & m_2^{t-1} & m_3^{t-1} & \cdots & m_t^{t-1} \end{bmatrix}} \underset{\mathbf{a}}{\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_t \end{bmatrix}} = \underset{\mathbf{b}}{\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{t-1} \end{bmatrix}}.$$

Note that if we choose $F = \mathbb{Q}$, the fractions will get large while solving this transposed Vandermonde system. Instead of using $\mathbb{Q}$, we can work on $F = \mathbb{Z}_p$ where $p > m_i < p_n^d$ if it is known that $\deg(f) \leq d$. If we choose $p > m_i$, all m_i remain distinct in $\mathbb{Z}_p$.

In the following, let $M(n)$ be the number of arithmetic operations in F for polynomial multiplication of two polynomials where the sum of the degrees of these polynomials is less than $2n$. Zippel presented an algorithm which requires $O(n^2)$ arithmetic operations in F and space for $O(n)$ elements of F to solve $U\mathbf{a} = \mathbf{b}$ [15]. To improve Zippel's method, Kaltofen and Yagati presented an asymptotically fast transposed Vandermonde solver [8]. They managed to reduce the number of arithmetic operations in F to $O(M(n) \log n)$. If we use the fast Fourier transform (FFT) algorithm, $M(n) \in O(n \log n)$. Thus, we get an $O(n \log^2 n)$ solver.

Kaltofen and Yagati's algorithm needs to evaluate two polynomials of degree $n - 1$ at n points $u_1, u_2, \ldots, u_n$. Horner's method takes $O(n^2)$ arithmetic operations in F with n distinct evaluation points to evaluate a polynomial of degree $n - 1$. To make polynomial multipoint evaluation more efficient, Borodin and Munro developed a divide-and-conquer algorithm that constructs a product tree [2]. In Figure 1.1, we present the layout of a product tree.

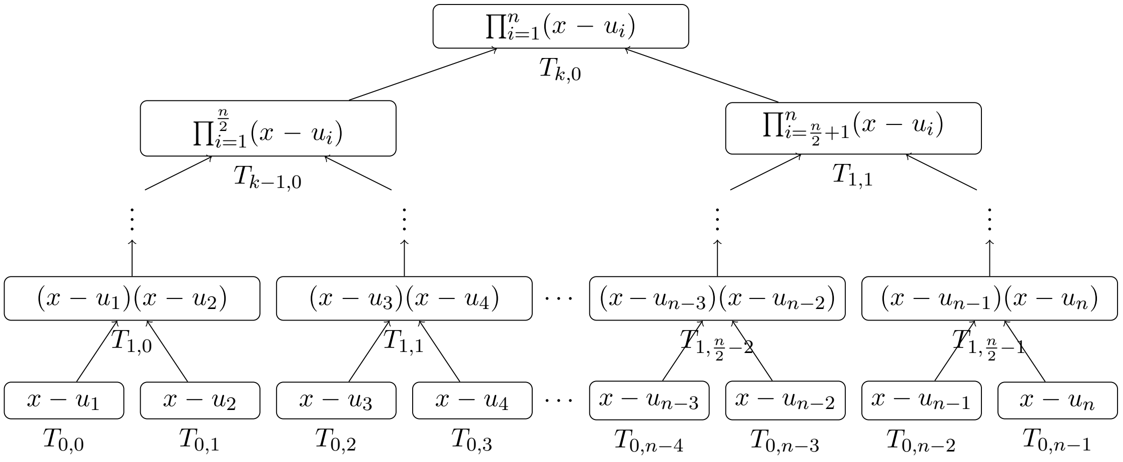

Figure 1.1: The layout of a product tree

The leaves of the product tree are the linear polynomials $x - u_1$, $x - u_2$, $\ldots$, $x - u_n$. This product tree consists of the product of $x - u_i$s as described in Figure 1.1 where u_is are evaluation points. Using this product tree, their algorithm takes $O(M(n) \log n)$ arithmetic operations in F to compute $a(u_i)$ for $1 \leq i \leq n$. Also, the product tree needs space for $O(n \log n)$ elements of F. In this thesis, we study, design and implement the fast transposed Vandermonde solver based on Kaltofen and Yagati's idea.

We first review the FFT algorithm, fast polynomial multiplication, and the permutation related to the FFT. Based on the FFT, we study three fast algorithms and present our optimized versions of them: fast polynomial division, fast polynomial multipoint evaluation, and the fast transposed Vandermonde solver. We have implemented the FFT and the three fast algorithms in C for $F = \mathbb{Z}_p$ with a prime $p < 2^{63}$. To show how well the algorithms perform, we time these fast algorithms and compare their time with the classical $O(n^2)$ algorithms.

In Chapter 2, we discuss the discrete Fourier transform (DFT), the FFT algorithm introduced by Cooley and Tuckey [3], and the inverse fast Fourier transform(inverse FFT) algorithm. Using the FFT and the inverse FFT, polynomial multiplication can be done in $O(n \log n)$ arithmetic operations in F where the sum of the degrees of two polynomials is less than $2n$. We can reduce the number of data moves by cancelling out the permutation from the FFT based on Law and Monagan's observation [9]. Also, we implement this fast multiplication in C and present an optimized version of fast polynomial multiplication. Additionally, the FFT permutes the elements in a way called the bit-reversal permutation. We present our own $O(n)$ algorithm for the bit-reversal permutation and an $O(n)$ algorithm for computing the DFT of the reciprocal polynomial $f^{(rec)}(x)$ given the DFT of $f(x)$.

In Chapter 3, we discuss classical polynomial division first. Using a Newton iteration, we can compute the inverse of the divisor as a power series. Then, using the reciprocal polynomial, we construct the fast polynomial division [14], which does at most $5M(n)+O(n)$ arithmetic operations in F. To speed up division, we have implemented the so-called "middle product" algorithm of Hanrot, Quercia, and Zimmerman [6]. This algorithm reduces the cost of fast division from at most $5M(n)+O(n)$ to at most $4M(n)+O(n)$ arithmetic operations in F. We further reduce this to $\frac{11}{3}M(n)+O(n)$. Then we optimize fast division by implementing these algorithms in C.

Next, we discuss the product tree and dividing down the product tree for fast polynomial multipoint evaluation [2]. Fast multipoint evaluation does $\frac{25}{6}M(n)\log_2 n + O(n \log n)$ arithmetic operations in F. We have optimized a C implementation. Lastly, we study Zippel's transposed Vandermonde solver and Kaltofen and Yagati's transposed Vanderemonde solver. After reviewing these methods, we present our optimized C implementation, which performs $\frac{53}{6}M(n)\log_2 n + O(n \log n)$ arithmetic operations in F.

Chapter 4 presents the execution time for our optimized fast division, fast evaluation, and fast transposed Vandermonde solver. We compare our timings with the timings of the classical algorithms. Our optimized fast transposed Vandermonde solver beats Zippel's $O(n^2)$ algorithm for $n \geq 128$, which is a good result.

In Chapter 5, we describe the polynomial representation in C and the underlying C library we use for the fast algorithms. Additionally, we discuss how we allocate memory to create a temporary array for each fast algorithm. Particularly, we optimize the structure of the product tree for fast multipoint evaluation due to the cutoff. To speed up our implementation, we have designed most of our algorithms so that they do not allocate memory. Instead, they run in the space of the input arrays, an output array, and a constant number of working arrays, where the size of the working arrays is linear in the size of the input arrays. This is important for recursive algorithms, which, otherwise, could spend a lot of time allocating and freeing many small arrays.

In this thesis, we have discovered the following which we believe to be original.

- The DFT of the reciprocal polynomial $F_\omega(f^{(rec)})$ can be obtained from the DFT of the original polynomial $F_\omega(f)$ without the FFT algorithms. Lemma 2.26 and Algorithm 10 give more details.
- We create the modified structure of the product tree for fast multipoint evaluation. Algorithm 21, Algorithm 22, and Algorithm 23 show more details.

Chapter 2

Tools

2.1 Fast Fourier Transform Algorithm

We start by recalling the details of the fast Fourier transform (FFT) algorithm and fast multiplication from [14]. Then we review Law and Monagan's fast multiplication [9] and implement our optimized fast multiplication algorithm in C. Moreover, we discuss the bit-reversal permutation in FFT and our algorithm for computing the discrete Fourier transform (DFT) of the reciprocal polynomial from the DFT of the original polynomial using linear work.

2.1.1 Fast Fourier Transform

Let F be a field. We need the definition of a primitive n-th root of unity in F. We set F to be finite fields of integers modulo p where p is a prime for examples and implementations.

Definition 2.1 (primitive n-th root of unity)**.** *An element $\omega \in F$ is a primitive n-th root of unity if $\omega^n = 1$ and $\omega^i \neq 1$ for all $1 \leq i < n$.*

Example 2.2. Let $F = \mathbb{Z}_{17}$. A primitive 4-th root of unity in $\mathbb{Z}_{17}$ is 13 because $13^4 \mod 17 = 1$, but $13^1 \mod 17 = 13$, $13^2 \mod 17 = 16$, and $13^3 \mod 17 = 4$.

With this definition, we can define the discrete Fourier transform.

Definition 2.3 (discrete Fourier transform)**.** *Suppose $a(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} \in F[x]$ is of degree less than n. Let ω be a primitive n-th root of unity in F. Also, let $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}] \in F^n$. Then the linear map $F_\omega : F^n \longrightarrow F^n$ which evaluates f at n powers of ω*

$$F_\omega(\mathbf{a}) = [a(\omega^i) : 0 \leq i \leq n-1]$$

is called the discrete Fourier transform (DFT).

Horner's evaluation method is one way to compute the $F_\omega(\mathbf{a})$. Horner's method does $n-1$ multiplications and $n-1$ additions to compute $a(\omega^i)$ for each i. Since there are n evaluation points, this method does $O(n^2)$ arithmetic operations in F. However, with the fast Fourier transform (FFT) algorithm, $F_\omega(\mathbf{a})$ can be computed in $O(n \log n)$ arithmetic operations in F.

In 1965, Cooley and Tuckey introduced the FFT algorithm to compute the discrete Fourier transform using a divide-and-conquer approach [3]. Before discussing the FFT algorithm, we list some properties of ω, a primitive n-th root of unity in F.

Lemma 2.4. *With even n, let ω be a primitive n-th root of unity in F. Then*

(i) $\omega^{\frac{n}{2}} = -1$

(ii) $\omega^{j+\frac{n}{2}} = -\omega^j$

(iii) ω^2 *is a primitive* $\frac{n}{2}$*-th root of unity in* F

(iv) $1 + \omega + \omega^2 + \cdots + \omega^{n-1} = 0$ *and*

(v) $\omega^{-1} = \omega^{n-1}$ *and* ω^{-1} *is a primitive* n*-th root of unity.*

(vi) ω^k *is a primitive* n*-th root of unity if and only if* $\gcd(k, n) = 1$.

Proof. (i) Since $\omega^n = 1$ and n is even, we have

$$\omega^n = \omega^{\frac{n}{2}} \cdot \omega^{\frac{n}{2}} = 1.$$

Then

$$(\omega^{\frac{n}{2}})^2 - 1 = (\omega^{\frac{n}{2}} - 1)(\omega^{\frac{n}{2}} + 1) = 0.$$

This implies that $\omega^{\frac{n}{2}} = \pm 1$. However, $\omega^{\frac{n}{2}} \neq 1$ by Definition 2.1. Thus, $\omega^{\frac{n}{2}} = -1$.

(ii) By (i), we have $\omega^{\frac{n}{2}} = -1$. Immediately it follows that

$$\omega^{j+\frac{n}{2}} = \omega^{\frac{n}{2}} \cdot \omega^j = (-1) \cdot \omega^j = -\omega^j.$$

(iii) Let $u = \omega^2$. We know that $\omega^i \neq 1$ for $0 < i < n$. Then

$$u^j = (\omega^2)^j \neq 1, \;\; 0 < j < \frac{n}{2}.$$

Also, $u^{\frac{n}{2}} = (\omega^2)^{\frac{n}{2}} = \omega^n = 1$. Therefore ω^2 is a primitive $\frac{n}{2}$-th root of unity.

(iv) From (ii), $\omega^{j+\frac{n}{2}} = -\omega^j$. Then we have

$$\begin{aligned}
1+\omega+\omega^2+\cdots+\omega^{n-1} &= (1+\omega+\cdots+\omega^{\frac{n}{2}-1}) + (\omega^{\frac{n}{2}}+\omega^{\frac{n}{2}+1}+\cdots+\omega^{n-1}) \\
&= \sum_{i=0}^{\frac{n}{2}-1}\omega^i + \sum_{i=0}^{\frac{n}{2}-1}\omega^{i+\frac{n}{2}} \\
&= \sum_{i=0}^{\frac{n}{2}-1}\omega^i + \sum_{i=0}^{\frac{n}{2}-1}-\omega^i \quad \text{by } (i) \\
&= 0.
\end{aligned}$$

(v) Since $\omega^n = 1$ by Definition 2.1, we have

$$\omega^n = \omega^{(n-1)+1} = \omega^{n-1}\cdot\omega = 1 \implies \omega^{-1} = \omega^{n-1}.$$

Next, towards a contradiction, we assume that ω^{-1} is not a primitive n-th root of unity. It follows that some m satisfying $(\omega^{-1})^m = 1$ exists where $1 \le m \le n-1$. Then

$$\omega^n \cdot (\omega^{-1})^m = 1\cdot 1 = 1.$$

This implies that $\omega^{n-m} = 1$. However, this contradicts that $\omega^i \neq 1$ for $1 \le i \le n-1$. Hence, ω^{-1} is a primitive n-th root of unity.

(vi) $(\Rightarrow)$ Assume $\gcd(k,n) \neq 1$. Let $\gcd(k,n) = g$. It follows that $k = q_k \cdot g$ and $n = q_n \cdot g$ where $\gcd(q_k, q_n) = 1$. This implies that

$$(\omega^k)^{q_n} = \omega^{k\cdot q_n} = \omega^{(q_k\cdot g)\cdot q_n} = \omega^{q_k\cdot n} = (\omega^n)^{q_k} = 1.$$

Since $q_n < n$, ω^k is not a primitive n-th root of unity.

$(\Leftarrow)$ Suppose $(\omega^k)^i = 1$ for some $1 \le i \le n-1$. Then $\omega^{k\cdot i} = 1$, which implies that $n | k\cdot i$ for $1 \le i \le n-1$. Thus, $\gcd(n,k) \neq 1$.

□

Now, we develop the FFT algorithm. This version of the FFT algorithm is called the decimation-in-time algorithm [12]. Suppose $a(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} \in F[x]$ is a polynomial of degree $n-1$ where n is even. Then we can rewrite $a(x)$ as

$$a(x) = (a_0 + a_2x^2 + a_4x^4 + \cdots + a_{n-2}x^{n-2}) + x(a_1 + a_3x^2 + a_5x^4 + \cdots + a_{n-1}x^{n-2}). \quad (2.1)$$

We define $b(x) = a_0 + a_2 x + a_4 x^2 + \cdots + a_{n-2} x^{\frac{n}{2}-1}$ and $c(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n-1} x^{\frac{n}{2}-1}$. Then (2.1) can be expressed as

$$a(x) = b(x^2) + x \cdot c(x^2).$$

Then, for $0 \leq i \leq \frac{n}{2} - 1$, we have

$$a(\omega^i) = b((\omega^2)^i) + \omega^i \cdot c((\omega^2)^i).$$

According to Lemma 2.4 (ii), $\omega^{i+\frac{n}{2}} = -w^i$. It follows that for $\frac{n}{2} \leq i \leq n-1$,

$$\begin{aligned} a(\omega^i) = a(\omega^{j+\frac{n}{2}}) &= b((\omega^{j+\frac{n}{2}})^2) + \omega^{j+\frac{n}{2}} \cdot c((\omega^{j+\frac{n}{2}})^2) \\ &= b((-\omega^j)^2) + (-\omega^j) \cdot c((-\omega^j)^2) \\ &= b((-\omega^2)^j) - \omega^j \cdot c((-\omega^2)^j). \end{aligned}$$

where $j = i - \frac{n}{2}$. Thus, for $0 \leq i \leq \frac{n}{2} - 1$, we have

$$\begin{cases} a(\omega^i) = b((\omega^2)^i) + \omega^i \cdot c((\omega^2)^i) \\ a(\omega^{i+\frac{n}{2}}) = b((\omega^2)^i) - \omega^i \cdot c((\omega^2)^i) \end{cases} \tag{2.2}$$

which differ by a sign. Now suppose $n = 2^k$ for some $k > 1$. Both $b(x)$ and $c(x)$ are polynomials of degree less than $\frac{n}{2}$. Also, by Lemma 2.4 (iii), ω^2 is a primitive $\frac{n}{2}$-th root of unity in F. We can evaluate $b(x)$ and $c(x)$ at the powers of ω^2 by two recursive calls. After evaluation, we can combine the results using (2.2) to obtain $a(\omega^i)$ for $0 \leq i \leq n-1$. The FFT in Algorithm 1 combines these ideas.

Theorem 2.5. *Algorithm 1 does $2n \log_2 n + n - 1$ arithmetic operations in F.*

Proof. Let $T(n)$ be the number of arithmetic operations in F for Algorithm 1 with a polynomial of degree less than n. Assume $n = 2^k$ for some $k \in \mathbb{N}$. When $n = 1$, no multiplication is performed. Thus, $T(1) = 0$. When $n \geq 2$, Algorithm 1 needs to compute ω^2 first by doing one multiplication. Then Algorithm 1 makes two recursive calls to compute $b((\omega^2)^i)$ and $c((\omega^2)^i)$ for $0 \leq i \leq \frac{n}{2} - 1$. In the for loop, one addition, one subtraction, and two multiplications occur at each iteration. This for loop iterates $\frac{n}{2}$ times. Thus, we have

$$\begin{cases} T(1) = 0 \\ T(n) = 2T\left(\frac{n}{2}\right) + 2n + 1 \;\text{ for } n \geq 2. \end{cases}$$

Algorithm 1 FFT(Fast Fourier Transform)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}] \in F^n$, and $\omega \in F$ is a primitive $n$-th
    root of unity
Output: $[a(1), a(\omega), \ldots, a(\omega^{n-1})] \in F^n$ where $a = \sum_{i=0}^{n-1} a_i x^i$,
if $n = 1$ then return $\mathbf{a}$ end if
$m \leftarrow \frac{n}{2}$
$\mathbf{b} \leftarrow [a_0, a_2, \ldots, a_{n-2}]$
$\mathbf{c} \leftarrow [a_1, a_3, \ldots, a_{n-1}]$
$B \leftarrow$ FFT($m$, $\mathbf{b}$, $\omega^2$)          // $B = [B_0, B_1, \ldots, B_{m-1}]$ where $B_i = a(\omega^{2i})$
$C \leftarrow$ FFT($m$, $\mathbf{c}$, $\omega^2$)          // $C = [C_0, C_1, \ldots, C_{m-1}]$ where $C_i = a(\omega^{2i+1})$
$y \leftarrow 1$
for $i$ from 0 to $m - 1$ do
    $t \leftarrow y \cdot C_i$          // $t = \omega^i \cdot C_i$
    $A_i \leftarrow B_i + t$          // $A_i = a(\omega^i)$
    $A_{i+m} \leftarrow B_i - t$          // $A_{i+m} = a(\omega^{i+\frac{n}{2}})$
    $y \leftarrow y \cdot \omega$          // $y = \omega^{i+1}$
end for
return $[A_0, A_1, \ldots, A_{n-1}]$
```

It follows that

$$\begin{aligned}
T(n) &= 2T(\tfrac{n}{2}) + 2n + 1 \\
&= 2(2T(\tfrac{n}{4}) + n + 1) + 2n + 1 = 4T(\tfrac{n}{4}) + 2n + 2 + 2n + 1 \\
&= 4(2T(\tfrac{n}{8}) + \tfrac{n}{2} + 1) + 2n + 2 + 2n + 1 \\
&= 8T(\tfrac{n}{8}) + 2n + 4 + 2n + 2 + 2n + 1 \\
&\vdots \\
&= 2n + 2^{k-1} + 2n + 2^{k-2} + \cdots + 2n + 2 + 2n + 1.
\end{aligned}$$

This expression simplifies:

$$\begin{aligned}
T(n) &= 2n + 2n + \cdots + 2n + 2^{k-1} + 2^{k-2} + \cdots + 1 \\
&= 2n + 2n + \cdots + 2n + 2^k - 1 \\
&= k(2n) + n - 1 \\
&= 2n \log_2 n + n - 1 \in O(n \log n).
\end{aligned}$$

□

Thus, Algorithm 1 does $O(n \log n)$ arithmetic operations in F.

2.1.2 Inverse Fast Fourier Transform

Let $\mathbf{a} \in F^n$ be a vector containing the coefficients of the polynomial $a = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} \in F[x]$ and $\omega \in F$ be a primitive n-th root of unity. Since F_ω is a linear transformation, we can express $F_\omega(\mathbf{a})$ as a matrix-vector multiplication. Suppose $n = 2^k$ for some $k \in \mathbb{N}$. The following Vandermonde linear system of equations shows $F_\omega(\mathbf{a})$.

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix}}_{V_\omega} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} a(1) \\ a(\omega) \\ a(\omega^2) \\ \vdots \\ a(\omega^{n-1}) \end{bmatrix}}_{\mathbf{b}} \tag{2.3}$$

Given $F_\omega(\mathbf{a})$, we can find the coefficients of $a(x)$ by interpolating $a(\omega^i)$ for $0 \leq i \leq n-1$. This interpolation is called the inverse discrete Fourier transform (IDFT). With $\mathbf{b} = [a(1), a(\omega), \ldots, a(\omega^{n-1})] \in F^n$, we can solve $V_\omega \mathbf{a} = \mathbf{b}$ for $\mathbf{a}$. Also, we can obtain $\mathbf{a}$ by computing V_ω^{-1} and multiplying V_ω^{-1} by $\mathbf{b}$. Both methods require $O(n^3)$ arithmetic operations in F to recover $\mathbf{a}$ with Gaussian elimination. To improve the cost of computing the IDFT, we use the following lemma.

Lemma 2.6 (Theorem 30.7 in [4]). *Let V_ω be a matrix of size $n \times n$ such that*

$$V_\omega = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix}$$

Then $V_\omega V_{\omega^{-1}} = nI$, which further implies that $(V_\omega)^{-1} = \frac{1}{n} V_{\omega^{-1}}$, where I is the $n \times n$ identity matrix.

Proof. Let N be the matrix computed by $V_\omega V_{\omega^{-1}}$. Then each entry $N_{i,j}$ is the dot product of the i-th row of V_ω and the j-th column of $V_{\omega^{-1}}$, so

$$N_{i,j} = 1 \cdot 1 + \omega^i \cdot \omega^{-j} + \cdots + \omega^{i(n-1)} \cdot \omega^{-j(n-1)}.$$

When $i \neq j$, we have

$$N_{i,j} = 1 + \omega^{i-j} + \cdots + \omega^{(i-j)(n-1)}.$$

For $1 \leq k \leq n-1$, we know that $1 - \omega^{kn} = 0$. It follows that

$$0 = 1 - \omega^{kn} = (1 - \omega^k)(1 + \omega^k + \omega^{2k} + \cdots + \omega^{(n-1)k}).$$

Since $1 - \omega^k \neq 0$ for $1 \leq k \leq n-1$,

$$1 + \omega^k + \omega^{2k} + \cdots + \omega^{(n-1)k} = 0.$$

Also, $1 - \omega^{-kn} = 0$. It follows that

$$0 = 1 - \omega^{-kn} = (1 - \omega^{-k})(1 + \omega^{-k} + \omega^{-2k} + \cdots + \omega^{-(n-1)k}).$$

Likewise, $1 - \omega^{-k} = 1 - (\omega^{-1})^k \neq 0$ for $1 \leq k \leq n-1$ by Lemma 2.4 (v). As a result,

$$1 + \omega^{-k} + \omega^{-2k} + \cdots + \omega^{-(n-1)k} = 0.$$

Therefore, for $i \neq j$,

$$N_{i,j} = 1 + \omega^{i-j} + \cdots + \omega^{(i-j)(n-1)} = 0.$$

On the other hand, when $i = j$,

$$N_{i,j} = 1 + \omega^{i-j} + \cdots + \omega^{(i-j)(n-1)} = 1 + 1 + \cdots + 1 = n.$$

It follows that

$$N = V_\omega V_{\omega^{-1}} = \begin{bmatrix} n & 0 & 0 & \cdots & 0 \\ 0 & n & 0 & \cdots & 0 \\ 0 & 0 & n & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & n \end{bmatrix} = n \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix} = nI.$$

Immediately, we have

$$(V_\omega)^{-1} V_\omega V_{\omega^{-1}} = (V_\omega)^{-1}(nI) \implies V_{\omega^{-1}} = n(V_\omega)^{-1} \implies (V_\omega)^{-1} = \frac{1}{n} V_{\omega^{-1}}.$$

□

Consequently, we can interpolate $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}]$ from $\mathbf{b} = [a(1), a(\omega), \ldots, a(\omega^{n-1})]$ using $F_{\omega^{-1}}(\mathbf{b})$. That is,

$$\mathbf{a} = (V_\omega)^{-1}\mathbf{b} = \frac{1}{n} V_{\omega^{-1}} \mathbf{b} = \frac{1}{n} F_{\omega^{-1}}(\mathbf{b}).$$

Lemma 2.4 (v) states that ω^{-1} is a primitive n-th root of unity. This implies that we are able to compute $F_{\omega^{-1}}(\mathbf{b})$ using Algorithm 1 with inputs n, $\mathbf{b}$, and ω^{-1}. Therefore, the IDFT can be computed using.

$$\mathbf{a} = \frac{1}{n} FFT(n, \mathbf{b}, \omega^{-1}) \tag{2.4}$$

To compute (2.4), we call Algorithm 1 with inputs n, $\mathbf{b}$, and ω^{-1} and multiply the output by $\frac{1}{n}$ to obtain $\mathbf{a}$. We call this the inverse fast Fourier transform (inverse FFT) algorithm.

2.1.3 Optimizing the Fast Fourier Transform

Law and Monagan presented an optimized version of Algorithm 1 that precomputes the powers of ω [9]. They created an array W such that

$$W = [1, \omega, \omega^2, \ldots, \omega^{\frac{n}{2}-1}, 1, \omega^2, \omega^4, \ldots, \omega^{\frac{n}{2}-2}, 1, \omega^4, \omega^8, \ldots, \omega^{\frac{n}{2}-4}, \ldots, 1, 0] \in F^n.$$

It costs $\frac{n}{2} - 1$ multiplications to build the array W.

Moreover, the output of the FFT can be returned in the input array $\mathbf{a}$, containing the coefficients of $a(x)$. For this in-place routine, they took a temporary array T of size n as input and removed the arrays $\mathbf{b}$ and $\mathbf{c}$ in Algorithm 1. They also used T for the two recursive calls to avoid allocating other memory space. This optimization is shown in Algorithm 2.

Algorithm 2 FFT1(Fast Fourier Transform 1)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}] \in F^n$, $W = [1, \omega, \omega^2, \ldots, \omega^{\frac{n}{2}-1}, 1, \omega^2, \omega^4, \ldots, \omega^{\frac{n}{2}-2}, 1, \omega^4, \omega^8, \ldots, \omega^{\frac{n}{2}-4}, \ldots, 1, 0]$ where $\omega \in F$ is a primitive $n$-th root of unity, and a temporary array $T$ of length $n$
Output: $[a(1), a(\omega), \ldots, a(\omega^{n-1})] \in F^n$ where $a = \sum_{i=0}^{n-1} a_i x^i$
if $n = 1$ then return $\mathbf{a}$ end if
$m \leftarrow \frac{n}{2}$
for $i$ from 0 to $m - 1$ do
    $T_i \leftarrow a_{2i}$
    $T_{m+i} \leftarrow a_{2i+1}$
end for
FFT1($m$, $T$, $W + m$, $\mathbf{a}$)    // $[T_0, T_1, \ldots, T_{m-1}]$ where $T_i = a(\omega^{2i})$
FFT1($m$, $T + m$, $W + m$, $\mathbf{a} + m$)    // $[T_m, T_{m+1}, \ldots, T_{n-1}]$ where $T_{m+i} = a(\omega^{2i+1})$
for $i$ from 0 to $m - 1$ do
    $s \leftarrow W_i \cdot T_{m+i}$    // $s = \omega^i \cdot a(\omega^{2i+1})$
    $a_i \leftarrow T_i + s$
    $a_{i+m} \leftarrow T_i - s$
end for
return
```

In Algorithm 2, the first recursive call takes an input $W + m$. The notation $W + m$ indicates that the array starts from the m-th element of W. This notation is based on pointer arithmetic in C. Likewise, $T + m$ means the array that starts from the m-th element of T, and $\mathbf{a} + m$ implies the array starts from the m-th element of $\mathbf{a}$ in the second recursive call.

Theorem 2.7. *Algorithm 2 does $\frac{3}{2}n\log_2 n$ arithmetic operations in F.*

Proof. Let $T(n)$ be the number of arithmetic operations in F for Algorithm 2 with a polynomial of degree less than n. When $n = 1$, no arithmetic operation is performed. For $n \geq 2$, each iteration does three arithmetic operations in the loop, which iterates $\frac{n}{2}$ times. Then we have

$$\begin{cases} T(1) = 0 \\ T(n) = 2T\left(\frac{n}{2}\right) + \frac{3n}{2} \text{ for } n \geq 2. \end{cases}$$

Thus, $T(n) = k(\frac{3n}{2}) = \frac{3}{2}n\log_2 n$ by the same type of computation as above. □

Remark 2.8. *There is one multiplication out of three arithmetic operations in the second for loop. We can refine the proof to show that Algorithm 2 does $\frac{1}{2}n\log_2 n$ multiplications in F.*

Remark 2.9. *Algorithm 2 is easily parallelized for large n by executing lines 7 and 8 in parallel for sufficiently large $n \geq 1024$ [9].*

2.2 Fast Multiplication

2.2.1 Fast Multiplication

Now, we look into how to use the FFT and inverse FFT algorithms to multiply two polynomials in $O(n\log n)$ arithmetic operations in F. Let $f, g \in F[x]$ be polynomials such that $f(x) = f_0 + f_1x + \cdots + f_{d_1}x^{d_1}$ and $g(x) = g_0 + g_1x + \cdots + g_{d_2}x^{d_2}$. Let $h \in F[x]$ be the product of f and g. The classical multiplication algorithm does at most $(d_1 + 1)(d_2 + 1)$ multiplications and $d_1 \cdot d_2$ additions in F. Thus, if $d = d_1 = d_2$, it costs at most $(d + 1)^2$ multiplications and d^2 additions in F to obtain h.

We choose n to be the smallest power of 2, satisfying $n > d_1 + d_2$. Suppose we can compute a primitive n-th root of unity $\omega \in F$. After computing ω, we create an array of W containing ω^i for $0 \leq i \leq n/2 - 1$ as described in Algorithm 2. To use the FFT, we define $A = [f_0, f_1, \ldots, f_{d_1}, 0, \ldots, 0] \in F^n$ by padding with zeros to length n. Similarly, we define $B = [g_0, g_1, \ldots, g_{d_2}, 0, \ldots, 0] \in F^n$. F_ω is a ring homomorphism. (See [14] Lemma 8.11.) It follows that, for $0 \leq i \leq n - 1$,

$$(f \cdot g)(\omega^i) = f(\omega^i) \cdot g(\omega^i).$$

Thus, point-wise multiplication of $F_\omega(A)$ and $F_\omega(B)$ gives $F_\omega(C)$ where $C \in F^n$ contains the coefficients of $h = f \cdot g$. Lastly, we use the inverse FFT to recover the polynomial h.

Algorithm 3 shows the fast multiplication algorithm with FFT1.

Algorithm 3 Fast multiplication

```
Input: Two polynomials $f, g \in F[x]$ such that $f(x) = \sum_{i=0}^{d_1} f_i x^i$ and $g(x) = \sum_{i=0}^{d_2} g_i x^i$
Output: $h(x) = f(x) \cdot g(x)$
$n \leftarrow$ The smallest power of 2 greater than $\deg(f) + \deg(g)$
Compute a primitive $n$-th root of unity $\omega$ in $F$
if $\omega$ does not exist then return FAIL  end if
$W \leftarrow [1, \omega, \omega^2, \dots, \omega^{\frac{n}{2}-1}, 1, \omega^2, \omega^4, \dots, \omega^{\frac{n}{2}-2}, 1, \omega^4, \omega^8, \dots, \omega^{\frac{n}{2}-4}, \dots, 1, 0]$
$T \leftarrow$ An array of length $n$
$A \leftarrow [f_0, f_1, \dots, f_{d_1}, 0, \dots, 0] \in F^n$
$B \leftarrow [g_0, g_1, \dots, g_{d_2}, 0, \dots, 0] \in F^n$
FFT1$(n, A, W, T)$        $//A = [A_0, A_1, \dots, A_{n-1}] = [f(1), f(\omega), \dots, f(\omega^{n-1})]$
FFT1$(n, B, W, T)$        $//B = [B_0, B_1, \dots, B_{n-1}] = [g(1), g(\omega), \dots, g(\omega^{n-1})]$
$C \leftarrow$ An array of length $n$
for $i$ from 0 to $n-1$ do $C_i \leftarrow A_i \cdot B_i$  end for        $// C_i = f(\omega^i) \cdot g(\omega^i)$
$V \leftarrow [1, \omega^{-1}, \omega^{-2}, \dots, \omega^{-\frac{n}{2}+1}, 1, \omega^{-2}, \omega^{-4}, \dots, \omega^{-\frac{n}{2}+2}, 1, \omega^{-4}, \omega^{-8}, \dots, \omega^{-\frac{n}{2}+4}, \dots, 1, 0]$
FFT1$(n, C, V, T)$
Compute $n^{-1}$ once
for $i$ from 0 to $n-1$ do $C_i \leftarrow n^{-1} \cdot C_i$  end for
return $\sum_{i=0}^{d_1+d_2} C_i x^i$
```

Example 2.10. Assume $F = \mathbb{Z}_{17}$. Let $f, g \in Z_{17}[x]$ be polynomials such that

$$f(x) = 1 + x + x^2$$
$$g(x) = 1 + 2x.$$

We want to compute $h(x) = f(x) \cdot g(x)$. Since $\deg(f) = 2$ and $\deg(g) = 1$, the smallest power of 2 greater than $\deg(f) + \deg(g)$ is 4, and then we set $n = 4$. We can obtain 13 as a primitive 4-th root of unity in $\mathbb{Z}_{17}$. We have the array W containing the powers of 13

$$W = [1, 13, 1, 0].$$

Next, we can define

$$A = [1, 1, 1, 0]$$
$$B = [1, 2, 0, 0]$$

By calling the FFT on A and B with an array T of length 4, we have

$$FFT1(4, A, W, T) \to A = [3, 13, 1, 4] = [f(1), f(\omega), f(\omega^2), f(\omega^3)]$$
$$FFT1(4, B, W, T) \to B = [3, 10, 16, 9] = [g(1), g(\omega), g(\omega^2), g(\omega^3)].$$

We compute point-wise multiplications of A and B. It follows that

$$C = [A_i \cdot B_i : \text{ for } i = 0, 1, 2, 3] = [9, 11, 16, 2].$$

Now, $13^{-1} \mod 17 = 4$, which is also a primitive 4-th root of unity in $\mathbb{Z}_{17}$ according to Lemma 2.4 (v). It follows that the array V is

$$V = [1, 4, 1, 0].$$

We can compute the inverse FFT on C with 4. With an array T of length 4, we have

$$FFT1(4, C, V, T) \to C = [4, 12, 12, 8].$$

By multiplying C by $4^{-1} \mod 17 = 13$, we have

$$C = [1, 3, 3, 2].$$

Thus, $h(x) = 1 + 3x + 3x^2 + 2x^3$.

Theorem 2.11. *Let $T(n)$ be the number of arithmetic operations in F that the FFT of size n. Then Algorithm 3 does $3T(2n) + O(n)$ arithmetic operations in F when the sum of the degrees of two polynomials is at most $2n - 1$.*

Proof. Let $M(n)$ be the number of arithmetic operations in F that Algorithm 3 does with two polynomials where the sum of the degrees of these polynomials is less than $2n$. Algorithm 3 calls three FFT1 of size $2n$, so it performs $3T(2n)$ arithmetic operations in F. It takes $2(n-1)$ multiplications to create arrays W and V. Also, it does $2n$ multiplications to compute the point-wise product of A and B. Moreover, Algorithm 3 does one inverse for n^{-1} and $2n$ multiplications to compute the product of C_i and n^{-1} for all i. Thus,

$$\begin{aligned} M(n) &= 3T(2n) + 2(n-1) + 2n + 1 + 2n \\ &= 3T(2n) + 6n - 1 \\ &= 3T(2n) + O(n). \end{aligned}$$

□

We know that Algorithm 2 FFT1 does $T(n) = \frac{3}{2}n \log_2 n$ arithmetic operations in F. Thus, $M(n) = 3T(2n) + O(n) = 3(\frac{3}{2}(2n)\log_2(2n)) + O(n) = 9n \log_2 n + O(n)$.

2.2.2 Another Fast Fourier Transform

In Algorithm 3, we allocate at least space for $6n$ elements of F in memory to store arrays A, B, W, V, T, and C. To reduce space, Law and Monagan used the other version of the FFT algorithm which is called a decimation-in-frequency algorithm [14], [12]. Let $a \in F[x]$ be a polynomial such that

$$a(x) = (a_0 + a_1 x + \cdots + a_{\frac{n}{2}-1} x^{\frac{n}{2}-1}) + (a_{\frac{n}{2}} x^{\frac{n}{2}} + a_{\frac{n}{2}+1} x^{\frac{n}{2}+1} + \cdots + a_{n-1} x^{n-1}).$$

First, we divide $a(x)$ by $(x^{\frac{n}{2}} - 1)$. Then we have

$$a(x) = q_1(x) \cdot (x^{\frac{n}{2}} - 1) + r_1(x) \tag{2.5}$$

where $r_1(x) = (a_0 + a_{\frac{n}{2}}) + (a_1 + a_{\frac{n}{2}+1})x + \cdots + (a_{\frac{n}{2}-1} + a_{n-1})x^{\frac{n}{2}-1}$. Additionally, when we divide $a(x)$ by $(x^{\frac{n}{2}} + 1)$, we have

$$a(x) = q_2(x) \cdot (x^{\frac{n}{2}} + 1) + r_2(x) \tag{2.6}$$

where $r_2(x) = (a_0 - a_{\frac{n}{2}}) + (a_1 - a_{\frac{n}{2}+1})x + \cdots + (a_{\frac{n}{2}-1} - a_{n-1})x^{\frac{n}{2}-1}$. We are able to compute r_1 in $\frac{n}{2}$ additions and r_2 in $\frac{n}{2}$ subtractions. Now, consider a evaluated at $x = \omega^{2i}$ where $\omega \in F$ is a primitive n-th root of unity. For $0 \leq i \leq \frac{n}{2} - 1$, (2.5) gives

$$a(\omega^{2i}) = q_1(\omega^{2i}) \cdot ((\omega^{2i})^{\frac{n}{2}} - 1) + r_1(\omega^{2i}) = r_1(\omega^{2i}).$$

Likewise, consider $a(x)$ evaluated at $x = \omega^{2i+1}$, for $0 \leq i \leq \frac{n}{2} - 1$. From (2.6), we have

$$\begin{aligned} a(\omega^{2i+1}) &= q_2(\omega^{2i+1}) \cdot ((\omega^{2i+1})^{\frac{n}{2}} + 1) + r_2(\omega^{2i+1}) \\ &= q_2(\omega^{2i+1}) \cdot (\omega^{\frac{2in}{2}+\frac{n}{2}} + 1) + r_2(\omega^{2i+1}) \\ &= r_2(\omega^{2i+1}). \end{aligned}$$

If we define $r_2^*(x) = r_2(\omega \cdot x) = \sum_{i=0}^{\frac{n}{2}-1} (a_i - a_{\frac{n}{2}+i})(\omega \cdot x)^i = \sum_{i=0}^{\frac{n}{2}-1} ((a_i - a_{\frac{n}{2}+i}) \cdot \omega^i) x^i$, it follows that

$$r_2^*(\omega^{2i}) = r_2(\omega^{2i+1}).$$

This decimation-in-frequency algorithm is given in Algorithm 4.

Theorem 2.12. *Algorithm 4 does $\frac{3}{2} n \log_2 n$ arithmetic operations in F.*

Proof. Let $T(n)$ be the number of arithmetic operations in F that Algorithm 4 performs with a polynomial of degree less than n. For $n = 1$, no arithmetic operation is performed.

```
Algorithm 4 FFT2(Fast Fourier Transform 2)
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}] \in F^n$ , $W = [1, \omega, \omega^2, \dots, \omega^{\frac{n}{2}-1}, 1, \omega^2, \omega^4, \dots, \omega^{\frac{n}{2}-2}, 1, \omega^4, \omega^8, \dots, \omega^{\frac{n}{2}-4}, \dots, 1, 0]$ where $\omega \in F$ is a primitive $n$-th root of unity, and a temporary array, $T$, of length $n$
Output: $[a(1), a(\omega), \dots, a(\omega^{n-1})] \in F^n$ where $a = \sum_{i=0}^{n-1} a_i x^i$
if $n = 1$ then return a  end if
$m \leftarrow \frac{n}{2}$
for  $i$ from 0 to $m - 1$ do
    $T_i \leftarrow a_i + a_{m+i}$
    $s \leftarrow a_i - a_{m+i}$
    $T_{m+i} \leftarrow s \cdot W_i$
end for
FFT2($m$, $T$, $W + m$, $\mathbf{a}$)      // $[T_0, T_1, \dots, T_{m-1}]$ where $T_i = a(\omega^{2i})$
FFT2($m$, $T + m$, $W + m$, $\mathbf{a} + m$)      // $[T_m, T_{m+1}, \dots, T_{n-1}]$ where $T_{m+i} = a(\omega^{2i+1})$
for $i$ from 0 to $m - 1$ do
    $a_{2i} \leftarrow T_i$
    $a_{2i+1} \leftarrow T_{m+i}$
end for
return
```

Otherwise, each iteration in the first loop does three arithmetic operations, and this loop iterates $\frac{n}{2}$ times. After the first loop, Algorithm 4 makes two recursive calls. Thus, we have

$$\begin{cases} T(1) = 0 \\ T(n) = 2T\left(\frac{n}{2}\right) + \frac{3n}{2} \text{ for } n \geq 2 \end{cases}$$

This recurrence relation is the same as Theorem 2.7, hence, $T(n) = \frac{3}{2} n \log_2 n$.

□

In [9], Law and Monagan presented an algorithm that computes $A = F_\omega(f)$ and $B = F_\omega(g)$ with FFT2 and $F_{\omega^{-1}}(A \times B)$ with FFT1, where × is a point-wise multiplication. We note that both Algorithm 2 and Algorithm 4 permute the order of elements in the input array **a**. In Algorithm 4, the permutation is performed at the end of the algorithm. On the other hand, Algorithm 2 permutes the elements of the input array in the beginning. The permutation related to FFT1 and FFT2 is known as the bit-reversal permutation. The bit-reversal permutation is an involution, that is, the bit-reversal permutation is its own functional inverse.

Example 2.13. Let σ be the bit-reversal permutation. Using Algorithm 2, we can observe the bit-reversal permutation with eight elements. Let $\mathbf{a} \in F^8$ be

$$\mathbf{a} = [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7].$$

which is one of the inputs for Algorithm 2. After the first for loop, we have

$$T = [a_0, a_2, a_4, a_6, a_1, a_3, a_5, a_7].$$

Then there are two recursive calls. In FFT1(4, T, $W+4$, $\mathbf{a}$), the first for loop permutes the elements in T. Then we have

$$\mathbf{a} = [a_0, a_4, a_2, a_6]$$

Likewise, FFT1(4, $T+4$, $W+4$, $\mathbf{a}+4$) permutes the elements of $T+4 = [a_1, a_3, a_5, a_7]$. After permuting elements,

$$\mathbf{a}+4 = [a_1, a_5, a_3, a_7]$$

Hence,

$$\mathbf{a} = [a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7]$$

After the for loop in FFT1(4, T, $W+4$, $\mathbf{a}$), two recursive calls with two elements result in

$$T = [a_0, a_4] \text{ and } T+2 = [a_2, a_6]$$

Similarly, two recursive calls with two elements after the for loop in FFT1(4, T, $W+4$, $\mathbf{a}$) yields

$$T+4 = [a_1, a_5] \text{ and } T+6 = [a_3, a_7]$$

Thus

$$T = [a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7]$$

Since FFT1 with one element does not perform the permutation, we end up with

$$\mathbf{a} = [a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7]$$

Hence, with eight elements, the bit-reversal permutation σ is

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 4 & 2 & 6 & 1 & 5 & 3 & 7 \end{pmatrix}.$$

In binary,

$$\sigma = \begin{pmatrix} 000_{(2)} & 001_{(2)} & 010_{(2)} & 011_{(2)} & 100_{(2)} & 101_{(2)} & 110_{(2)} & 111_{(2)} \\ 000_{(2)} & 100_{(2)} & 010_{(2)} & 110_{(2)} & 001_{(2)} & 101_{(2)} & 011_{(2)} & 111_{(2)} \end{pmatrix}.$$

Hence, $\sigma^{-1} = \sigma$.

Law and Michael observed that the bit-reversal permutation is cancelled out by applying FFT2 to compute F_ω of two polynomials first and FFT1 to compute $F_{\omega^{-1}}$ on the point-

Algorithm 5 FFT1(Fast Fourier Transform 1 with no permutation)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}] \in F^n$, and $W = [1, \omega, \omega^2, \dots, \omega^{\frac{n}{2}-1}, 1, \omega^2, \omega^4, \dots, \omega^{\frac{n}{2}-2}, 1, \omega^4, \omega^8, \dots, \omega^{\frac{n}{2}-4}, \dots, 1, 0]$ where $\omega \in F$ is a primitive $n$-th root of unity
Output: $[a(1), a(\omega), \dots, a(\omega^{n-1})] \in F^n$ where $a = \sum_{i=0}^{n-1} a_i x^i$
if $n = 1$ then return a end if
$m \leftarrow \frac{n}{2}$
FFT1($m$, $\mathbf{a}$, $W + m$ )      // $a = [a_1, a_2, \dots, a_{m-1}]$
FFT1($m$, $\mathbf{a} + m$, $W + m$ )      // $a + m = [a_m, a_{m+1}, \dots, a_{n-1}]$
for $i$ from 0 to $m - 1$ do
    $s \leftarrow a_i$
    $t \leftarrow W_i \cdot a_{m+i}$          // $W_i = \omega^i$ for $0 \leq i \leq n - 1$
    $a_i \leftarrow s + t$
    $a_{m+i} \leftarrow s - t$
end for
return
```

wise product [9]. Thus, we do not have to permute the elements. Since data movement is expensive on a modern computer, Law and Monagan suggested removing the permutation steps from both Algorithm 2 and Algorithm 4 [9]. By doing so, we also do not have to allocate a temporary array T since no space is required to store the permutation. Permutation-eliminated FFT1 and FFT2 are presented as Algorithm 5 and Algorithm 6. The number of arithmetic operations in F for both algorithms is the same as for the previous versions with the permutation.

Algorithm 6 FFT2(Fast Fourier Transform 2 with no permutation)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}] \in F^n$, and $W = [1, \omega, \omega^2, \dots, \omega^{\frac{n}{2}-1}, 1, \omega^2, \omega^4, \dots, \omega^{\frac{n}{2}-2}, 1, \omega^4, \omega^8, \dots, \omega^{\frac{n}{2}-4}, \dots, 1, 0]$ where $\omega \in F$ is a primitive $n$-th root of unity
Output: $[a(1), a(\omega), \dots, a(\omega^{n-1})] \in F^n$ where $a = \sum_{i=0}^{n-1} a_i x^i$,
if $n = 1$ then return a end if
$m \leftarrow \frac{n}{2}$
for $i$ from 0 to $m - 1$ do          // $a_i = f_i$ for $0 \leq i \leq n - 1$
    $s \leftarrow a_i + a_{m+i}$
    $t \leftarrow a_i - a_{m+i}$
    $a_i \leftarrow s$
    $a_{m+i} \leftarrow t \cdot W_i$          // $W_i = \omega^i$ for $0 \leq i \leq n - 1$
end for
FFT2($m$, $\mathbf{a}$, $W + m$)
FFT2($m$, $\mathbf{a} + m$, $W + m$)
return
```

2.2.3 Optimizing Fast Multiplication

In Algorithm 3, the arrays W and V are created independently. However, we can create the array V containing powers of ω^{-1} by permuting the elements of W. According to Lemma 2.4 (ii), $\omega^{\frac{n}{2}-i} = \omega^{-i}$ for $1 \leq i \leq \frac{n}{2} - 1$. When we compute W, the first $\frac{n}{2}$ elements are

$$W = [1, \omega, \omega^2, \ldots, \omega^{\frac{n}{2}-1}, \ldots].$$

Since the first $\frac{n}{2}$ elements of V are

$$V = [1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(\frac{n}{2}-1)}, \ldots] = [1, \omega^{\frac{n}{2}-1}, \omega^{\frac{n}{2}-2}, \ldots, \omega, \ldots],$$

we can obtain V by setting $V_i = W_{\frac{n}{2}-i}$ for $1 \leq i \leq \frac{n}{2} - 1$. Then we construct the remaining part of V from the first $\frac{n}{2}$ elements in V.

We also consider the boundary case. Let $f, g \in F[x]$ be polynomials such that $f = f_0 + f_1 x + \cdots + f_{d_1} x^{d_1}$ and $g = g_0 + g_1 x + \cdots + g_{d_2} x^{d_2}$. Assume $\deg(f) + \deg(g) = d_1 + d_2 = n = 2^k$ for some $k \in \mathbb{N}$. That is, the degree of $f \cdot g$ is n. To compute this product, with Algorithm 3, we need to use the FFT of size $2n$. In Chapter 3, we will encounter this type of polynomial multiplication when constructing a product tree.

To reduce the size of the FFT to n, we compute

$$f \cdot g = f \cdot (g - g_{d_2} x^{d_2}) + f \cdot g_{d_2} x^{d_2}.$$

Since f is of degree d_1 and $g - g_{d_2} x^{d_2}$ is of degree $d_2 - 1$, $d_1 + (d_2 - 1) = n - 1 < n$. Thus $f \cdot (g - g_{d_2} x^{d_2})$ can be done with the FFT of size n. After this multiplication, it takes $d_1 + 1$ multiplications to compute $f \cdot g_{d_2} x^{d_2}$, and we add this result to $f \cdot (g - g_{d_2} x^{d_2})$, which does at most d_1 additions. We present Algorithm 7 based on these improvements.

Theorem 2.14. *Assume that $T(n)$ is the number of arithmetic operations in F to compute FFT of size n. It follows that Algorithm 7 does $3T(2n) + O(n)$ arithmetic operations in F when the sum of the degrees of two polynomials is at most $2n - 1$.*

Proof. Let $M(n)$ be the number of arithmetic operations in F that Algorithm 7 does with two polynomials where the sum of the degrees of these polynomials is less than $2n$. Algorithm 7 executes three FFTs of size $2n$. It takes $n - 1$ multiplications to compute W. Then Algorithm 7 does $2n$ multiplications for point-wise multiplication. During the inverse FFT, it takes one inverse to get n^{-1} and $2n$ multiplications to multiply the result of the inverse

Algorithm 7 Fast multiplication (using FFT1 and FFT2 with no permutation) [9]

```
Input: Two polynomials $f(x), g(x) \in F[x]$ where $F$ is a field such that $f(x) = \sum_{i=0}^{d_1} f_i x^i$
    and $g(x) = \sum_{i=0}^{d_2} g_i x^i$
Output: $h(x) = f(x) \cdot g(x)$
$d \leftarrow \deg(f) + \deg(g)$
$n \leftarrow$ the smallest power of 2 greater than $d - 1$
if $d = n$ then
    $h_1 \leftarrow$ Fast multiplication($f$,$g - g_{d_2} x^{d_2}$)        //$h_1 = f \cdot (g - g_{d_2} x^{d_2})$
    $h_2 \leftarrow f \cdot g_{d_2} x^{d_2}$
    $h \leftarrow h_1 + h_2$
    return $h$
end if
Compute a primitive $n$-th root of unity $\omega$ in $F$
if $\omega$ does not exist then return FAIL  end if
$W \leftarrow [1, \omega, \omega^2, \dots, \omega^{\frac{n}{2}-1}, 1, \omega^2, \omega^4 \dots, \omega^{\frac{n}{2}-2}, 1, \omega^4, \omega^8, \dots, \omega^{\frac{n}{2}-4}, \dots, 1, 0]$
$A \leftarrow [f_0, f_1, \dots, f_{d_1}, 0, \dots, 0] \in F^n$
$B \leftarrow [g_0, g_1, \dots, g_{d_2}, 0, \dots, 0] \in F^n$
FFT2($n, A, W$)    //$A = [A_0, A_1, \dots, A_{n-1}] = [f(1), f(\omega), \dots, f(\omega^{n-1})]$ ......... $T(n)$
FFT2($n, B, W$)    //$B = [B_0, B_1, \dots, B_{n-1}] = [g(1), g(\omega), \dots, g(\omega^{n-1})]$ ......... $T(n)$
$C \leftarrow$ An array of length $n$
for $i$ from 0 to $n - 1$ do $C_i \leftarrow A_i \cdot B_i$  end for     //$C_i = f(\omega^i) \cdot g(\omega^i)$
$V \leftarrow [1, \omega^{-1}, \omega^{-2}, \dots, \omega^{-\frac{n}{2}+1}, 1, \omega^{-2}, \omega^{-4}, \dots, \omega^{-\frac{n}{2}+2}, 1, \omega^{-4}, \omega^{-8}, \dots, \omega^{-\frac{n}{2}+4}, \dots, 1, 0]$
FFT1($n, C, V$) ............................................................ $T(n)$
Compute $n^{-1}$ once
for $i$ from 0 to $n-1$ do $C_i \leftarrow n^{-1} \cdot C_i$  end for        //$C = [C_0, C_1, \dots, C_{d_1+d_2}, 0, \dots, 0]$
return $\sum_{i=0}^{d_1+d_2} C_i x^i$
```

FFT by n^{-1}. Therefore,

$$\begin{aligned} M(n) &= 3T(2n) + (n - 1) + 2n + 1 + 2n \\ &= 3T(2n) + 5n = 3T(2n) + O(n) \end{aligned}$$

□

Both Algorithm 5 FFT1 and Algorithm 6 FFT2 do $T(n) = \frac{3}{2} n \log_2 n$ arithmetic operations in F. It follows that $M(n) = 9n \log_2 n + 14n \in O(n \log n)$ arithmetic operations in F. Also, Algorithm 7 performs $3n \log_2 n + O(n)$ multiplications in F.

Corollary 2.15. *Let $M(n)$ be the number of arithmetic operations in F that Algorithm 7 does with two polynomials where the sum of the degrees of these polynomials is less than $2n$. Then*

$$M(n) \geq 2M(\tfrac{n}{2}).$$

Proof. From the proof of Theorem 2.14, we note that if $n \geq m$ for some $n, m \in \mathbb{N}$,

$$\frac{M(n)}{n} = 9\log_2 n + 14 \geq 9\log_2 m + 14 = \frac{M(m)}{m}.$$

If $m = \frac{n}{2}$, we have

$$\frac{M(n)}{n} \geq \frac{M\left(\frac{n}{2}\right)}{\frac{n}{2}} \implies M(n) \geq 2M\left(\frac{n}{2}\right).$$

□

In the remainder of this thesis, we assume that $2M\left(\frac{n}{2}\right) \leq M(n)$ if Algorithm 7 Fast multiplication is used. Basically, we assume polynomial multiplication is at least linear complexity.

Now, we compare Algorithm 7 to the classical polynomial multiplication algorithm. In Monagan's C library for polynomial arithmetic in $\mathbb{Z}_p[x]$ with a prime $p < 2^{63}$, `polmul64s` implements the classical $O(n^2)$ polynomial multiplication. We worked over the field $F = \mathbb{Z}_p$ where $p = 3 \cdot 2^{30} + 1$, a 32-bit prime. We use two polynomials of degree $n = 2^k$ for some $k \in \mathbb{N}$. These two polynomials are generated by choosing their coefficients at random from $[0, p)$.

n	Classical multiplication (ms)	Fast multiplication (ms)
2^6	0.0075	0.0146
2^7	0.0249	0.0322
2^8	0.0890	0.0713
2^9	0.3348	0.1612
2^{10}	1.327	0.3391
2^{11}	5.217	0.7205
2^{12}	20.84	1.608

Table 2.1: Timings in ms for the classical multiplication of two polynomials of degree n and Algorithm 7 Fast multiplication

From Table 2.1, the classical multiplication algorithm is faster than Algorithm 7 for $n < 2^8 = 256$. Contrarily, Algorithm 7 is faster than the classical one for $n \geq 256$. This implies that when the degrees of two polynomials are at least 256, Algorithm 7 Fast multiplication takes less time. From Table 2.1, we present an optimized fast multiplication in Algorithm 8.

2.3 FFT Permutation

2.3.1 Bit-reversal Permutation

In the previous section, we mentioned that Cooley and Tuckey's FFT algorithm involves the bit-reversal permutation. Now, we define the bit-reversal permutation.

Algorithm 8 Optimized fast multiplication (FastMul)

```
Input: Polynomials $f, g \in F[x]$ where $F$ is a field
Output: $h(x) = f(x) \cdot g(x)$
if $\deg(f) \cdot \deg(g) < 2^{16}$ then
    $h \leftarrow$ Call classical multiplication with inputs $f$ and $g$
    return $h$
end if
$h \leftarrow$ Call Algorithm 7 Fast multiplication with inputs $f$ and $g$
return $h$
```

Definition 2.16. *Let $\{0, 1, \ldots, n-1\}$ be a finite set of size $n = 2^k$ for some $k \in \mathbb{N}$. The bit-reversal permutation σ is*

$$\sigma(i) = j$$

where $i = (b_{k-1}b_{k-2}\ldots b_0)_2$ and $j = (b_0 b_1 \ldots b_{k-1})_2$ in binary.

Remark 2.17. *The bit-reversal permutation is an involution. In other words, $\sigma(\sigma(i)) = i$ for all $i \in \{0, 1, \ldots, n-1\}$.*

Example 2.18. Let $n = 8 = 2^3$. Then

i	i in binary	$\sigma(i)$ in binary	$\sigma(i)$
0	000_2	000_2	0
1	001_2	100_2	4
2	010_2	010_2	2
3	011_2	110_2	6
4	100_2	001_2	1
5	101_2	101_2	5
6	110_2	011_2	3
7	111_2	111_2	7

Table 2.2: The bit-reversal permutation of {0,1,...,7}

Let $P(n)$ be the number of moves of elements in Algorithm 1 where $n = 2^k$ for some $k \in \mathbb{N}$. If $n = 1$, no permutation is performed. Otherwise, there are $\frac{n}{2}$ moves to $\mathbf{b}$ and $\frac{n}{2}$ moves to $\mathbf{c}$. Also, two recursive calls exist in Algorithm 1. Thus, we have

$$\begin{cases} P(1) = 0 \\ P(n) = 2\left(\frac{n}{2}\right) + 2P(\frac{n}{2}) \text{ for } n \geq 2. \end{cases}$$

This implies that $P(n) = n + 2(\frac{n}{2}) + \cdots + 2^{k-1}(2) = kn = n\log_2 n$.

We found that the bit-reversal permutation σ can be done with linear moves based on the following lemma.

Lemma 2.19. *Let $\sigma : \{0, 1, \ldots, n-1\} \to \{0, 1, \ldots, n-1\}$ be the bit-reversal permutation where $n = 2^k$ for some $k \in \mathbb{N}$. Then, for $0 \le j \le 2^i - 1$,*

$$\sigma(2^i + j) = \sigma(j) + 2^{k-(i+1)}.$$

Example 2.20. Continuing from Example 2.18, $k = 3$ since $8 = 2^3$. Then we have

$$\sigma(6) = \sigma(2^2 + 2) = \sigma(2) + 2^{3-(2+1)} = 2 + 2^0 = 3.$$

Proof (Lemma 2.19). It is obvious that $\sigma(0) = 0$. For all $0 \le i \le k-1$, assume we know $\sigma(0), \sigma(1), \ldots, \sigma(2^i - 1)$. If we permute 2^i to $2^{i+1} - 1$, 1 is in the i-th bit in binary. In other words, we permute

$$2^i + j = (00\ldots\overbrace{1}^{i\text{-th}}b_{i-1}b_{i-2}\ldots b_0)_2$$

for $0 \le j \le 2^i - 1$. Then applying σ on $2^i + j$

$$\begin{aligned}\sigma(2^i + j) &= (b_0 b_1 \ldots b_{i-1} 10 \ldots 0)_2 \\ &= (b_0 b_1 \ldots b_{i-1} 0 \ldots 0)_2 + (00\ldots 0 \quad \overbrace{1}^{k-(i+1)\text{-th}} \quad 0 \ldots 0)_2 \\ &= \sigma(j) + 2^{k-(i+1)}\end{aligned}$$

since every j is at most i digit binary number for $0 \le j \le 2^i - 1$.

We know each $\sigma(j)$ for $0 \le j \le 2^i - 1$. It follows that we only need to add $2^{k-(i+1)}$ to all $\sigma(j)$ to compute $\sigma(2^i + j)$. Thus,

$$\sigma(2^i + j) = \sigma(j) + 2^{k-(i+1)}$$

for $0 \le j \le 2^i - 1$. □

Using the identity of Lemma 2.19, we present our bit-reversal permutation algorithm in Algorithm 9.

Theorem 2.21. *Algorithm 9 does $n + O(\log n)$ arithmetic operations in $\mathbb{Z}$.*

Proof. Let $P(n)$ be the number of arithmetic operations in $\mathbb{Z}$ that Algorithm 9 does. One division is performed to compute δ. In the outer loop, one division and one multiplication are performed at each iteration. Also, in the inner for loop, every iteration does one addition and this loop iterates i times. Hence, i additions exist at each iteration in the outer loop. This outer loop iterates $\log_2 n$ times since i doubles each iteration in the outer while loop.

Algorithm 9 Bit-reversal permutation

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$
Output: $\pi = [\sigma(0), \sigma(1), \ldots, \sigma(n-1)]$ where $\sigma$ is a bit-reversal permutation.
$\pi_0 \leftarrow 0$          $//\pi_0 = \sigma(0) = 0$
$\delta \leftarrow \frac{n}{2}$
$i \leftarrow 1$
while $i < n$ do
    for $j$ from 0 to $i - 1$ do $\pi_{i+j} \leftarrow \pi_j + \delta$ end for
    $\delta \leftarrow \frac{\delta}{2}$
    $i \leftarrow i \cdot 2$
end while
return $\pi$
```

This implies that

$$\begin{aligned} P(n) &= 1 + \sum_{j=0}^{k-1} (1 + 1 + 2^j) \\ &= 1 + 2k + 2^k - 1 \\ &= n + 2\log_2 n \\ &= n + O(\log n). \end{aligned}$$

□

2.3.2 FFT on the Reciprocal Polynomials

Definition 2.22. *Let $f \in F[x]$ be a polynomial such that $f = f_d x^d + f_{d-1} x^{d-1} + \cdots f_1 x + f_0$ with $f_d \neq 0$. The reciprocal polynomial of f is defined*

$$f^{(rec)}(x) = x^d f\left(\frac{1}{x}\right) = f_0 x^n + f_1 x^{d-1} + \cdots + f_{d-1} x + f_d.$$

Therefore, $f^{(rec)}$ has coefficients in reverse order of the coefficients of f.

Example 2.23. Assume $f = 1 - 2x + 4x^2 + 3x^3$. Then the reciprocal polynomial of f is

$$f^{(rec)} = x^3 f\left(\frac{1}{x}\right) = x^3 \left(1 - 2\left(\frac{1}{x}\right) + 4\left(\frac{1}{x}\right)^2 + 3\left(\frac{1}{x}\right)^3\right) = x^3 - 2x^2 + 4x + 3.$$

Suppose $\omega \in F$ is a primitive n-th root of unity and $\deg(f) < n$. Then

$$F_\omega(f) = [f(1), f(\omega), \ldots, f(\omega^{n-1})]$$

Now, we consider computing $F_\omega(f^{(rec)})$ from $F_\omega(f)$. One way to compute this is the following. We use the inverse FFT on $F_\omega(f)$ to obtain f and then compute $f^{(rec)}$ from f. Then we obtain $F_\omega(f^{(rec)})$ by applying the FFT on $f^{(rec)}$. It takes two FFTs of size n, both of which perform $O(n \log n)$ arithmetic operations in F. However, we can compute $F_\omega(f^{(rec)})$ from $F_\omega(f)$ with no FFT but some linear work.

Remark 2.24. *Let $f \in F[x]$ be a polynomial such that $f = f_d x^d + f_{d-1}x^{d-1} + \cdots f_1 x + f_0$ with $f_d \neq 0$. Assume $\omega \in F$ is a primitive n-th root of unity. Then*

$$f^{(rec)}(\omega^i) = (\omega^i)^d f(\omega^{n-i}).$$

Algorithm 6 FFT2 returns the array $\mathbf{a}$ containing the values of $f(\omega^i)$ in the order of the bit-reversal permutation σ:

$$\mathbf{a} = F_\omega(f) = [f(\omega^{\sigma(0)}), f(\omega^{\sigma(1)}), \ldots, f(\omega^{\sigma(n-1)})].$$

Now, we can obtain $f^{(rec)}(\omega^i)$ for $0 \leq i \leq n-1$ from $\mathbf{a}$ using Lemma 2.26 where $d = \deg(f)$:

$$F_\omega(f^{(rec)}) = [(\omega^{\sigma(0)})^d \cdot f(\omega^{n-\sigma(0)}), (\omega^{\sigma(1)})^d \cdot f(\omega^{n-\sigma(1)}), \ldots, (\omega^{\sigma(n-1)})^d \cdot f(\omega^{n-\sigma(n-1)})]$$

To compute $f(\omega^{n-\sigma(i)})$ for $0 \leq i \leq n-1$, we observe a property of $n - \sigma(i)$.

Example 2.25. Let $n = 16 = 2^4$. For every i, we define $m = 2^i$ and notice that $n-\sigma(m+j)$ lists the $\sigma(m+j)$ in reverse order for all $0 \leq j \leq m-1$ from Table 2.3.

m	j	$m+j$	$\sigma(m+j)$	$n-\sigma(m+j)$
		0	0	16
2^0	0	1	8	8
2^1	0	2	4	12
	1	3	12	4
2^2	0	4	2	14
	1	5	10	6
	2	6	6	10
	3	7	14	2
2^3	0	8	1	15
	1	9	9	7
	2	10	5	11
	3	11	13	3
	4	12	3	13
	5	13	11	5
	6	14	7	9
	7	15	15	1

Table 2.3: $\sigma(m+j)$ and $n-\sigma(m+j)$ with the bit-reversal permutation σ for $n = 16$

From Example 2.25, for each m, $n - \sigma(m+j)$ where j is from 0 to $m-1$ equals $\sigma(m+j^*)$ where j^* is from $m-1$ to 0. We set $j = m-1-j^*$ and then $j^* = m-1-j$. This implies that

$$\sigma(m+j^*) = \sigma(m+m-1-j) = \sigma(2m-1-j).$$

Thus, $n - \sigma(m+j) = \sigma(2m-1-j)$.

Lemma 2.26. *Let $n = 2^k$ for some $k \in \mathbb{N}$ and σ be the bit-reversal permutation. Assume $m = 2^i$ for some $0 \leq i \leq k-1$. Then*

$$\sigma(m+j) + \sigma(2m-1-j) = n$$

for any $0 \leq j \leq m-1$.

Proof. Let j be an arbitrarily chosen value from $0 \leq j \leq m-1$. Then, in binary,

$$j = (b_{i-1}b_{i-2}\dots b_1b_0)_2$$

where $b_d \in \{0,1\}$ for all d. Since $m = 2^i$, for $0 \leq j \leq m-1$

$$m + j = 2^i + j = (00\dots0\overbrace{\ 1\ }^{i\text{th}}b_{i-1}b_{i-2}\dots b_0)_2.$$

It follows that

$$\sigma(m+j) = (b_0b_1\dots b_{i-1}\quad\overbrace{\ 1\ }^{k-(i+1)\text{th}}\quad 0\dots0)_2.$$

Also, in binary, $2m-1-j$ can be written as

$$(2m-1)-j = (00\dots0\overbrace{\ 1\ }^{ith}1\dots1)_2 - (00\dots0b_{i-1}b_{i-2}\dots b_0)_2 = (00\dots0\overbrace{\ 1\ }^{i\text{th}}c_{i-1}c_{i-2}\dots c_0)_2$$

where $c_d = 1 - b_d$ for $0 \leq d \leq i-1$. This implies that

$$\sigma(2m-1-j) = (c_0c_1\dots c_{i-1}\quad\overbrace{\ 1\ }^{k-(i+1)\text{th}}\quad 0\dots0)_2.$$

Using $c_d = 1 - b_d$,

$$\begin{aligned}
\sigma(m+j) + \sigma(2m-1-j) &= (b_0 b_1 \dots b_{i-1} \overbrace{1}^{k-(i+1)\text{th}} 0\dots0)_2 + (c_0 c_1 \dots c_{i-1} \overbrace{1}^{k-(i+1)\text{th}} 0\dots0)_2 \\
&= (b_0 b_1 \dots b_{i-1} 10\dots0)_2 + (11\dots1 \overbrace{1}^{k-(i+1)\text{th}} 0\dots0)_2 - (b_0 b_1 \dots b_{i-1} 0\dots0)_2 \\
&= (00\dots0 \overbrace{1}^{k-(i+1)\text{th}} 0\dots0)_2 + (11\dots1 \overbrace{1}^{k-(i+1)\text{th}} 0\dots0)_2 \\
&= (\overbrace{1}^{k\text{th}} 000\dots0)_2 \\
&= 2^k = n.
\end{aligned}$$

□

Using this property, we present Algorithm 10 that obtains $F_\omega(f^{(rec)})$ from $F_\omega(f)$ with no FFTs.

Algorithm 10 FFT of the reciprocal polynomial

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $A = [f(\omega^{\sigma(0)}), f(\omega^{\sigma(1)}), \dots, f(\omega^{\sigma(n-1)})]$ where $\sigma$ is a bit-reversal permutation and $A = F_\omega(f)$, and $\omega \in F$, a primitive $n$-th root of unity
Output: $F_\omega(f^{(rec)}) = [f^{(rec)}(\omega^{\sigma(0)}), f^{(rec)}(\omega^{\sigma(1)}), \dots, f^{(rec)}(\omega^{\sigma(n-1)})]$
 1: $I \leftarrow$ Call Algorithm 9 with $n$        // $I = [\sigma(0), \sigma(1), \dots, \sigma(n-1)] = [I_0, I_1, \dots, I_{n-1}]$
 2: $u \leftarrow \omega^d$
 3: $W_0 \leftarrow 1$
 4: for $i$ from 1 to $n-1$ do $W_{\sigma(i)} = W_{\sigma(i-1)} \cdot u$ end for
 5: $i \leftarrow 1$
 6: $B_0 \leftarrow A_0$
 7: while $i < n$ do
 8:     $l \leftarrow i \cdot 2 - 1$
 9:     for $j$ from 0 to $i-1$ do $B_{i+j} \leftarrow A_{l-j}$ end for
10:     $i \leftarrow l + 1$
11: end while
12: for $i$ from 1 to $n-1$ do $B_i \leftarrow B_i \cdot W_i$ end for
13: return $B$
```

Theorem 2.27. *Assume Algorithm 10 works on the field $F = \mathbb{Z}_p$, where p is a prime. Algorithm 10 does at most $3n + O(\log n)$ arithmetic operations in F.*

Proof. Let $R(n)$ be the number of arithmetic operations in F that Algorithm 10 does. Line 2 takes at most $2\lceil \log_2 d \rceil$ multiplications to compute $\omega^d \mod p$ using the Square-and-multiply algorithm [13]. Since $d \leq n$, computing $\omega^d \mod p$ does at most $2 \log_2 n$ arithmetic operations in F. After computing ω^d, the first for loop does $n-1$ multiplications. Then the while loop iterates $\log_2 n$ times, and each iteration does one multiplication, one subtraction, and one addition. The inner for loop iterates i times, and it takes one addition and one subtraction to compute indices of A and B. Hence, the inner for loop costs $2i$ arithmetic operations in F. The last for loop does $n-1$ multiplications in line 12. This implies that

$$\begin{aligned}
R(n) &\leq 2\lceil \log_2 d \rceil + n - 1 + \sum_{i=1}^{k} (3 + 2 \cdot 2^{i-1}) + n - 1 \\
&\leq 2\log_2 n + 3\log_2 n + 2\left(\tfrac{2^k-1}{2-1}\right) + 2n - 2 \\
&= 5\log_2 n + 2(n-1) + 2n - 2 \\
&= 4n + 5\log_2 n - 4 \\
&= 4n + O(\log n).
\end{aligned}$$

□

Chapter 3

Fast Algorithms

We need two fast algorithms to construct the fast Vandermonde solver algorithm: the fast division algorithm and the fast multipoint evaluation algorithm. These two fast algorithms use the fast multiplication in Chapter 2. The fast division algorithm uses Newton inversion [14] and the middle product [6]. Then we optimize Newton inversion by computing repeated F_ω once. The fast multipoint evaluation algorithm uses the product tree [2]. We modify the product tree by optimizing F_ω in this chapter. For comparison, we review Zippel's transposed Vandermonde solver [15]. Then we present Kaltofen and Yagati's fast transposed Vandermonde solver [8].

3.1 Fast Division

3.1.1 Classical Division Algorithm

Let F be a field. Suppose $f, g \in F[x]$ are polynomials such that $f = \sum_{i=0}^{m} f_i x^i$ and $g = \sum_{i=0}^{n} g_i x^i$. It follows that the degree of f is m, and the degree of g is n. Let $LT(f)$ denote the leading term of f, i.e., $LT(f) = f_m x^m$. To divide f by g, we want to obtain the quotient and remainder $q, r \in F[x]$ such that

$$f = g \cdot q + r \tag{3.1}$$

where $r = 0$ or $\deg(r) < n$. Algorithm 11 presents the classical division algorithm for $F[x]$.

Theorem 3.1. *Let $f, g \in F[x]$ be such that f is a dividend and g is a divisor. Assume* $\deg(f) = m$ *and* $\deg(g) = n$ *and $m \geq n$. Then Algorithm 11 performs at most $O((m - n + 1)n)$ arithmetic operations in F.*

Proof. Algorithm 11 computes a new term $q^{(*)}$ of the quotient q in each iteration. Since $\deg(q) = m - n$, q has at most $m - n + 1$ terms. Thus, lines 4, 5, and 6 are executed at most

Algorithm 11 Classical division

```
Input: $f, g \in F[x]$ where $g \neq 0$.
Output: $q, r \in F[x]$ such that $f = g \cdot q + r$ where $r = 0$ or $\deg(r) < \deg(g)$.
$q \leftarrow 0$
$r \leftarrow f$
while $r \neq 0$ and $\deg(r) \geq \deg(g)$ do
    $q^{(*)} \leftarrow LT(r)/LT(g)$
    $q \leftarrow q + q^{(*)}$
    $r \leftarrow r - (g \cdot q^{(*)})$
end while
return $q, r$
```

$m-n+1$ times. Each iteration does one division in F to compute $q^{(*)}$. Since g is a polynomial of degree n, g has at most $n+1$ terms. We know that $g_n x^n \cdot q^{(*)} = f_m x^m$, so it is unnecessary to compute this again. Therefore, $g \cdot q^{(*)}$ requires at most n multiplications in F. Additionally, the difference between $LT(f)$ and $LT(g \cdot q^{(*)})$ is zero. It follows that $f - (g \cdot q^{(*)})$ needs at most n subtractions in F. Thus, the total number of arithmetic operations in F for the classical division is at most $(m-n+1)(1+n+n) = (m-n+1)(2n+1) \in O((m-n+1)n)$. □

Corollary 3.2. *If $m = 2n$, this division does $(2n - n + 1)(1 + n + n) = (n + 1)(2n + 1) = 2n^2 + 3n + 1 \in O(n^2)$ arithmetic operations in F.*

Corollary 3.3. *If $n = 1$, this division does $(m - 1 + 1)(1 + 1 + 1) = 3m \in O(m)$ arithmetic operations in F.*

Corollary 3.3 will be used in Zippel's transposed Vandermonde solver.

3.1.2 Newton Inversion

To obtain a faster division algorithm, we need to look into how to find the inverse of the divisor polynomial. To discuss the inverse, let $\hat{f} = f(x) \mod x^n$ be the remainder when f is divided by x^n. This is equivalent to simply truncating f at the x^{n-1} term.

Definition 3.4. *Let $f \in R[x]$ where R is any ring. The order n approximation of f is $\hat{f} = f(x) \mod x^n$.*

Let $f \in F[x]$ be a polynomial where $f = \sum_{i=0}^{n-1} f_i x^i$ with $f_0 \neq 0$. We can compute a polynomial $a(x) \mod x^n \in F[x]$ by equating the coefficients $f \cdot a = 1$ where a is the inverse of f. It follows that

$$\begin{cases} 1 = f_0 \cdot a_0 \\ 0 = \sum_{i+j=k} f_i \cdot a_j \quad \text{for } k \geq 1. \end{cases}$$

Then we have $a_0 = f_0^{-1}$ since $f_0 \neq 0$, which does one inverse. Also, for $k \geq 1$, $f_0 \cdot a_k = -\sum_{i+j=k, i\neq 0} f_i \cdot a_j$. This implies that

$$a_k = f_0^{-1} \cdot (-\sum_{i+j=k, i\neq 0} f_i \cdot a_j) = -a_0 \cdot (\sum_{i+j=k, i\neq 0} f_i \cdot a_j). \tag{3.2}$$

For $k \geq 1$, it takes k multiplications and $k-1$ additions to compute the sum in (3.2) and one multiplication and one negation to multiply $-a_0$ by the sum. Thus computing a_k does $k + (k-1) + 1 + 1 = 2k + 1$ arithmetic operations in F. When we compute the polynomial a of degree $n-1$, this method does

$$1 + \sum_{i=1}^{n-1} (2k+1) = 1 + 2\frac{n(n-1)}{2} + n - 1 = 1 + n^2 - n + n - 1 = n^2$$

arithmetic operations in F. Therefore equating the coefficients of $f \cdot a = 1$ costs $O(n^2)$.

To speed up finding the inverse, recall that Newton's method is a root-finding algorithm which successively approximates the roots of equation $f(x) = 0$ [14]. Newton's method starts with an initial approximation α_0. Subsequent approximations are computed using

$$\alpha_i = \alpha_{i-1} - \frac{f(\alpha_{i-1})}{f'(\alpha_{i-1})}$$

for $i \geq 1$. With a suitable initial point α_0, this iteration generates a sequence $\{\alpha_i\}_{i=0}^{\infty}$ converging to one of the roots of the equation $f(x) = 0$.

Newton's iteration generates a sequence of polynomials approximating f^{-1}. Denote $y = f^{-1}$. Then we can obtain y by solving the following equation:

$$f(x) - \frac{1}{y(x)} = 0.$$

We define $a(y) = f - \frac{1}{y}$. We can obtain f^{-1} from solving $a(y) = 0$ by Newton's iteration. Assume $y_0 \in F$ is the initial approximation. The derivative of $a(y)$ is $a'(y) = \frac{1}{y^2}$. To compute the root of $a(y)$, Newton iteration with y_0 results in

$$y_i = y_{i-1} - \frac{a(y_{i-1})}{a'(y_{i-1})} = y_{i-1} - \frac{f - \frac{1}{y_{i-1}}}{\frac{1}{y_{i-1}^2}} = y_{i-1} - f \cdot y_{i-1}^2 + y_{i-1} = 2y_{i-1} - f \cdot y_{i-1}^2 \tag{3.3}$$

for $i \geq 1$. We call this method Newton inversion. With Newton inversion, we can compute the inverse of a polynomial as a power series to an order n [14].

Theorem 3.5. *Let F be a field. Assume $f = f_0 + f_1 x + f_2 x^2 + \cdots \in F[x]$ and $f_0 \neq 0$. Let $y_0 = f_0^{-1}$ and $y_i = 2y_{i-1} - f \cdot y_{i-1}^2 \mod x^{2^i}$ for $i \geq 1$. For all $i \geq 0$,*

$$f \cdot y_i \mod x^{2^i} = 1.$$

Proof. See the proof of Theorem 9.2. in [14]. □

We note that we use $y_i = 2y_{i-1} - (f \mod x^{2^i}) \cdot y_{i-1}^2 \mod x^{2^i}$ for the recurrence.

Example 3.6. We want to compute the inverse of $f = 1 + 3x + 5x^2$ to an order 4 approximation. That is, we compute $(1+3x+5x^2)^{-1} \mod x^4$. According to Theorem 3.5, we start with

$$y_0 = 1^{-1} = 1 \mod x^1$$

When $i = 1$, we have

$$\begin{aligned} y_1 &= 2y_0 - f \cdot y_0^2 = 2 \cdot 1 - (1 + 3x + 5x^2 \mod x^2) \cdot 1^2 = 2 - 1 - 3x \\ &= 1 - 3x \mod x^2. \end{aligned}$$

When $i = 2$, we have

$$\begin{aligned} y_2 &= 2y_1 - f \cdot y_1^2 = 2 \cdot (1 - 3x) - (1 + 3x + 5x^2)(1 - 3x)^2 \\ &= 2 - 6x - (1 - 3x - 4x^2 - 3x^3 + 45x^4) = 2 - 6x - 1 + 3x + 4x^2 + 3x^3 - 45x^4 \\ &= 1 - 3x + 4x^2 + 3x^3 \mod x^4. \end{aligned}$$

Then we have $f^{-1} \mod x^4 = 1 - 3x + 4x^2 + 3x^3$.

Applying Theorem 3.5, we can create a recursive algorithm as described in Algorithm 12.

Algorithm 12 Newton inversion

Input: A polynomial $f = f_0 + f_1 x + \cdots \in F[x]$ with $f_0 \neq 0$ and $n \in \mathbb{N}$

Output: An order n approximation of f^{-1}

```
if $n = 1$ then
    $y \leftarrow f_0^{-1} \in F$
    return $y$
end if
$m \leftarrow \lceil \frac{n}{2} \rceil$
$g \leftarrow f \mod x^n$
$g^{-1} \leftarrow$ Newton inversion$(g, m)$
$b \leftarrow g^{-1} \cdot g^{-1} \mod x^n$
$c \leftarrow g \cdot b \mod x^n$
$y \leftarrow 2 \cdot g^{-1} - c$
return $y$
```

Theorem 3.7. *Let $n = 2^k$ for some $k \in \mathbb{N}$. Let $M(n)$ be the number of arithmetic operations in F for polynomial multiplication where the sum of the degrees of two polynomials is less than $2n$. Then Algorithm 12 does at most $3M(n) + O(n)$ arithmetic operations in F.*

Proof. Let $I(n)$ be the number of arithmetic operations for computing $f^{-1} \bmod x^n$ by Algorithm 12. When $n = 1$, Algorithm 12 only computes f_0^{-1} in line 2, and hence, $I(1) = 1$. For $n > 1$, Algorithm 12 makes one recursive call to compute g^{-1} to an order $m = \frac{n}{2}$, which costs $I(\frac{n}{2})$. Then, in line 8, since the degree of g^{-1} is $\frac{n}{2} - 1$, $g^{-1} \cdot g^{-1}$ is a polynomial multiplication and costs less than $M(\frac{n}{2})$. In line 9, there is another multiplication of two polynomials, b of degree $n - 2$ and g of degree $n - 1$, which performs less than $M(n)$ arithmetic operations in F. Moreover, it takes $O(n)$ arithmetic operations in F to compute $2 \cdot g^{-1} - c$ in line 9. Hence, we have

$$\begin{cases} I(1) = 1 \\ I(n) < I\left(\frac{n}{2}\right) + M\left(\frac{n}{2}\right) + M\left(n\right) + O(n) \text{ for } n > 1 \end{cases}$$

By Corollary 2.15, we observe that $M(\frac{n}{2}) \leq \frac{1}{2}M(n)$ using fast multiplication. In other words, polynomial multiplication is not sub-linear. Then we have

$$\begin{aligned} I(n) &< I(\tfrac{n}{4}) + M(\tfrac{n}{4}) + M(\tfrac{n}{2}) + O(\tfrac{n}{2}) + M(\tfrac{n}{2}) + M(n) + O(n) \\ &\leq I(\tfrac{n}{8}) + M(\tfrac{n}{8}) + M(\tfrac{n}{4}) + O(\tfrac{n}{4}) + \big(M(\tfrac{n}{2}) + M(\tfrac{n}{4})\big) + \big(M(n) + M(\tfrac{n}{2})\big) + O(n) \\ &\ \vdots \\ &\leq \big(M\left(\tfrac{n}{2}\right) + M\left(\tfrac{n}{4}\right) + \cdots + M(1)\big) + \big(M(n) + M\left(\tfrac{n}{2}\right) + \cdots + M(2)\big) + O(n) \\ &\leq \tfrac{1}{2}M(n) + \tfrac{1}{4}M(n) + \cdots + \tfrac{1}{n}M(n) + M(n) + \tfrac{1}{2}M(n) + \cdots + \tfrac{1}{n/2}M(n) + O(n) \\ &= \left(\tfrac{1}{2} + \tfrac{1}{4} + \cdots + \tfrac{1}{n}\right) M(n) + \left(1 + \tfrac{1}{2} + \cdots + \tfrac{1}{n/2}\right) M(n) + O(n) \\ &\leq M(n) + 2M(n) + O(n) \in 3M(n) + O(n). \end{aligned}$$

□

If a primitive n-th root of unity in F exists, $M(n) = 9n \log_2 n + O(n)$ using fast multiplication and Algorithm 12 does at most $27n \log_2 n + O(n) \in O(n \log n)$ arithmetic operations in F.

3.1.3 The Middle Product

In 2004, Hanrot, Quercia, and Zimmerman [6] reduced the constant 3 in Theorem 3.7 to 2, that is the complexity of their Newton inversion is $2M(n) + O(n)$. From (3.3), they considered an alternative formula for Newton's method.

$$y_k = 2y_{k-1} - f \cdot y_{k-1}^2 = y_{k-1} + y_{k-1} - f \cdot y_{k-1}^2 = y_{k-1} + y_{k-1} \cdot (1 - f \cdot y_{k-1})$$

We know that $f \cdot y_{k-1} \mod x^{2^{k-1}} = 1$. Assume $n = 2^k$ and $y_{k-1} = h_0 + h_1 x + \cdots + h_{\frac{n}{2}-1} x^{\frac{n}{2}-1}$. Let f^* denote $f \mod x^n$. After $f \mod x^n$ is computed,

$$\begin{aligned} f^* \cdot y_{k-1} &= (f_0 + f_1 x + \cdots + f_{n-1} x^{n-1})(h_0 + h_1 x + \cdots + h_{\frac{n}{2}-1} x^{\frac{n}{2}-1}) \\ &= 1 + 0 \cdot x + \cdots + 0 \cdot x^{\frac{n}{2}-1} + m_0 x^{\frac{n}{2}} + \cdots + m_{\frac{n}{2}-1} x^{n-1} + a_0 x^n + \cdots + a_{\frac{n}{2}-2} x^{\frac{3n}{2}-2} \end{aligned}$$

for some $m_i, a_i \in F$. Let $mp = \sum_{i=0}^{\frac{n}{2}-1} m_i x^i$ and $a = \sum_{i=0}^{\frac{n}{2}-2} a_i x^i$. It follows that

$$f^* \cdot y_{k-1} \mod x^n = 1 + mp \cdot x^{\frac{n}{2}} + a \cdot x^n \mod x^n = 1 + mp \cdot x^{\frac{n}{2}}.$$

The polynomial mp is called the middle product.

Since $\deg(f^*) = n - 1$ and $\deg(y_{k-1}) = \frac{n}{2} - 1$, the degree of $f^* \cdot y_{2k-1} = \frac{3n}{2} - 2$. Algorithm 7 Fast multiplication needs an FFT of size $2n$, which is the smallest power of 2 greater than $\frac{3n}{2} - 2$. However, instead of using the FFT of size $2n$ to multiply f^* by y_{k-1}, we will use an FFT of size n.

Let $A = [f_0, f_1, \ldots, f_{n-1}] \in F^n$ where every f_i is the coefficient of f^* at degree i. Also, let $B = [h_0, h_1, \ldots, h_{\frac{n}{2}-1}, 0, \ldots, 0] \in F^n$ where each h_i is the coefficient of y_{k-1} at degree i. Assume $\omega \in F$ is a primitive n-th root of unity. We define $C = F_w(A) \times F_w(B) \in F^n$, where $\times$ is a point-wise multiplication. This implies that

$$C = [f^* \cdot y_{k-1}(1), f^* \cdot y_{k-1}(\omega), \ldots, f^* \cdot y_{k-1}(\omega^{n-1})].$$

We note that for all $0 \leq i \leq n - 1$

$$\begin{aligned} f^* \cdot y_{k-1}(\omega^i) &= 1 + mp(\omega^i) \cdot (\omega^i)^{\frac{n}{2}} + a(\omega^i) \cdot (\omega^i)^n \\ &= 1 + mp(\omega^i) \cdot (\omega^i)^{\frac{n}{2}} + a(\omega^i) \cdot (\omega^n)^i \\ &= 1 + mp(\omega^i) \cdot (\omega^i)^{\frac{n}{2}} + a(\omega^i) \quad \text{(since } \omega^n = 1) \end{aligned}$$

Thus, C equals $F_w(D)$ where $D = [1 + a_0, a_1, \ldots, a_{\frac{n}{2}-2}, 0, m_0, m_1, \ldots, m_{\frac{n}{2}-1}] \in F^n$ containing the coefficients of $1 + mp \cdot x^{\frac{n}{2}} + a$.

Thus, applying the inverse FFT of size n on C yields

$$\frac{1}{n} F_{\omega^{-1}}(C) = D.$$

This enables us to extract the coefficients of mp. With this middle product, we can rewrite Newton's method as

$$y_k = y_{k-1} + y_{k-1} \cdot (1 - f \cdot y_{k-1}) = y_{k-1} + (y_{k-1} \cdot (-mp) \mod x^{\frac{n}{2}}) \cdot x^{\frac{n}{2}} \tag{3.4}$$

Based on (3.4), we can use the middle product to improve Newton inversion algorithm. This is presented in Algorithm 13.

Algorithm 13 Newton inversion with the middle product

```
Input: A polynomial $f = f_0 + f_1 x + \cdots \in F[x]$ with $f_0 \neq 0$ and $n \in \mathbb{N}$
Output: An order $n$ approximation of $f^{-1}$
 1: if $n = 1$ then
 2:     $y \leftarrow f_0^{-1} \in F$
 3:     return $y$
 4: end if
 5: $m \leftarrow \lceil \frac{n}{2} \rceil$
 6: $g \leftarrow f \mod x^n$
 7: $y \leftarrow$ Algorithm 13 Newton inversion with the middle product($g$,$m$) ............. $I(m)$
 8: $b \leftarrow y \cdot g$ using fast multiplication with the FFT of size $n$   // $b = \sum_{i=0}^{n-1} b_i x^i$ .... $M(\frac{n}{2})$
 9: $mp \leftarrow -\sum_{i=0}^{n-m-1} b_{i+m} x^i$
10: $h \leftarrow y \cdot mp$ using fast multiplication   // $h = \sum_{i=0}^{n-2} h_i x^i$ ........................ $M(\frac{n}{2})$
11: $y \leftarrow y + x^m \cdot \sum_{i=0}^{n-m-1} h_i x^i$
12: return $y$
```

Theorem 3.8. *Let $n = 2^k$ for some $k \in \mathbb{N}$. Let $M(n)$ be the number of arithmetic operations in F for multiplying two polynomials where the sum of the degrees of these polynomials is less than $2n$. Then Algorithm 13 does at most $2M(n) + O(n)$ arithmetic operations in F.*

Proof. Let $I(n)$ be the total number of arithmetic operations in F for Algorithm 13. When $n = 1$, it only needs to compute the inverse of $f_0 \in F$. This implies that $I(1) = 1$. Otherwise, the algorithm calls itself with size m once. Since $n = 2^k$, $m = \lceil \frac{n}{2} \rceil = \frac{n}{2}$, it costs $I(\frac{n}{2})$. In Chapter 2, we observe that polynomial multiplication where the sum of the degrees of two polynomials is less than n uses the FFT of size n. However, in line 8, fast multiplication with the FFT of size n computes $y \cdot g$ although $\deg(y) + \deg(g) = \frac{3n}{2} - 2$ to read off the coefficients of mp. This implies that computing $y \cdot g$ costs $M(\frac{n}{2})$. Also, computing $y \cdot mp$ where $\deg(y) = \frac{n}{2} - 1$ and $\deg(mp) = \frac{n}{2} - 1$ does $M(\frac{n}{2})$ arithmetic operations in F. In line 9, it takes $n - m$ arithmetic operations in F to negate coefficients. No additions are performed in line 11. Thus, we have

$$\begin{cases} I(1) = 1 \\ I(n) < I\left(\frac{n}{2}\right) + 2M\left(\frac{n}{2}\right) + O(n) \ \text{ for } n > 1 \end{cases}$$

By Corollary 2.15, $2M(\frac{n}{2}) \leq M(n)$. Then

$$\begin{aligned} I(n) &< 2M\left(\tfrac{n}{2}\right) + 2M\left(\tfrac{n}{4}\right) + \cdots + 2M(1) + O(n) \\ &\leq M\left(n\right) + M\left(\tfrac{n}{2}\right) + \cdots + M(2) + O(n) \\ &\leq M\left(n\right) + \tfrac{1}{2}M\left(n\right) + \cdots + \tfrac{1}{n/2}M(n) + O(n) \\ &= \left(1 + \tfrac{1}{2} + \tfrac{1}{4} + \cdots + \tfrac{1}{n/2}\right)M(n) + O(n) \\ &< 2M(n) + O(n) \end{aligned}$$

□

Compared to Algorithm 12, Algorithm 13 saves one polynomial multiplication $M(n)$ out of three.

3.1.4 Optimizing Newton Inversion

In Algorithm 13, fast multiplication is executed twice to get the inverse of $f \mod x^n$. Each fast multiplication requires three FFTs of size n. We can observe that Algorithm 13 computes $F_\omega(y)$ twice in lines 8 and 10 for polynomial multiplications. Also, the same list of powers of a primitive n-th root of unity is created redundantly in these two fast multiplications. To reduce the number of arithmetic operations, we can compute $F_\omega(y)$ once and reuse the result during the second multiplication. To do so, we break apart Algorithm 7 Fast multiplication. An optimized Newton inversion with the middle product is shown in Algorithm 14.

Theorem 3.9. *Assume $d = n = 2^k$ for some $k \in \mathbb{N}$. Let $T(n)$ be the number of arithmetic operations in F that FFT of size n does. Let $M(n)$ be the number of arithmetic operations in F for polynomial multiplications where the sum of the degrees of two polynomials is less than $2n$. Then Algorithm 14 does at most $\frac{5}{3}M(n) + O(n)$ arithmetic operations in F.*

Proof. Let $I(n)$ be the number of arithmetic operations in F for Algorithm 14. When $n = 1$, $I(1) = 1$ for one negation. For $n > 1$, $m = \frac{d}{2} = \frac{n}{2}$ because we assume n is the power of 2. It follows that one recursive call does $I(\frac{n}{2})$ arithmetic operations in F. Moreover, Algorithm 14 calls two FFT1s and three FFT2s, which cost $5T(n)$ since $d = n$. It takes $O(n)$ arithmetic operations in F to create W and V. Also, point-wise multiplications and scalar multiplications in Algorithm 14 do $O(n)$ arithmetic operations in F. Thus,

$$\begin{cases} I(1) = 1 \\ I(n) \leq I(\frac{n}{2}) + 5T(n) + O(n) & \text{for } n > 1. \end{cases}$$

Fast multiplication does $M(n) = 3T(2n) + O(n)$ arithmetic operations in F by Theorem 2.14. Then we have $3T(n) \leq M(\frac{n}{2})$. This further implies that $T(n) \leq \frac{1}{3}M(\frac{n}{2}) \leq \frac{1}{6}M(n)$ from

Algorithm 14 Optimized Newton inversion with the middle product (NIwithMP)

```
Input: A polynomial $f = f_0 + f_1x + \cdots \in F[x]$ with $f_0 \neq 0$ and $d \in \mathbb{N}$
Output: An order $d$ approximation of $f^{-1}$
if $d = 1$ then
    $y \leftarrow f_0^{-1} \in F$
    return $y$
end if
$m \leftarrow \lceil \frac{d}{2} \rceil$
$a \leftarrow f \mod x^d$    // $a = \sum_{i=0}^{n-1} f_i x^i$
$y \leftarrow$ NIwithMP $(a, m)$    // $y = \sum_{i=0}^{m-1} y_i x^i = f^{-1} \mod x^m$ .................... $I(m)$
$n \leftarrow$ The smallest power of 2 greater than $d - 1$
$\omega \leftarrow$ Compute a primitive $n$-th root of unity in $F$
$W \leftarrow [1, \omega, \omega^2, \dots, \omega^{\frac{n}{2}-1}, 1, \omega^2, \omega^4, \dots, \omega^{\frac{n}{2}-2}, 1, \omega^4, \omega^8, \dots, \omega^{\frac{n}{2}-4}, \dots, 1, 0]$
$A \leftarrow [f_0, f_1, \dots, f_{d-1}, 0, \dots, 0] \in F^n$
$Y \leftarrow [y_0, y_1, \dots, y_{m-1}, 0, \dots, 0] \in F^n$
FFT2($n$,$A$,$W$)    // $F_\omega(A) = [F_0, F_1, \dots, F_{n-1}]$ .................... $T(n)$
FFT2($n$,$Y$,$W$)    // $F_\omega(Y) = [Y_0, Y_1, \dots, Y_{n-1}]$ .................... $T(n)$
$C \leftarrow$ An array of length $n$ //    $C = [C_0, C_1, \dots, C_{n-1}]$
for $i$ from 0 to $n - 1$ do $C_i \leftarrow F_i \cdot Y_i$ end for
$V \leftarrow [1, \omega^{-1}, \omega^{-2}, \dots, \omega^{-\frac{n}{2}+1}, 1, \omega^{-2}, \omega^{-4}, \dots, \omega^{-\frac{n}{2}+2}, 1, \omega^{-4}, \omega^{-8}, \dots, \omega^{-\frac{n}{2}+4}, \dots, 1, 0]$
FFT1($n$,$C$,$V$) .................... $T(n)$
$t \leftarrow n^{-1}$
for $i$ from 0 to $n - 1$ do $C_i \leftarrow t \cdot C_i$ end for
$M \leftarrow [-C_m, -C_{m+1}, \dots, -C_{d-1}, 0, \dots, 0] \in F^n$    // $mp = -\sum_{i=0}^{d-m-1} C_{i+m} x^i$
FFT2($n$,$M$,$W$) //    $F_\omega(M) = [M_0, M_1, \dots, M_{n-1}]$ .................... $T(n)$
$H \leftarrow$ An array of length $n$ //    $H = [H_0, H_1, \dots, H_{n-1}]$
for $i$ from 0 to $n - 1$ do $H_i \leftarrow M_i \cdot Y_i$ end for
FFT1($n$,$H$,$V$) .................... $T(n)$
for $i$ from 0 to $n - 1$ do $H_i \leftarrow t \cdot H_i$ end for
$y \leftarrow y + x^m \cdot \sum_{i=0}^{d-m-1} H_i x^i$
return $y$
```

Corollary 2.15. Therefore,

$$
\begin{aligned}
I(n) &\leq I(\tfrac{n}{2}) + \tfrac{5}{6}M(n) + O(n) \\
&\leq I(\tfrac{n}{4}) + \tfrac{5}{6}M(\tfrac{n}{2}) + \tfrac{5}{6}M(n) + O(n) \\
&\vdots \\
&\leq 1 + \tfrac{5}{6}M(2) + \cdots + \tfrac{5}{6}M(\tfrac{n}{2}) + \tfrac{5}{6}M(n) + O(n) \\
&\leq \tfrac{5}{6}M(n) + \tfrac{5}{12}M(n) + \cdots + \tfrac{5}{6(n/2)}M(n) + O(n) \quad (\text{since } M(\tfrac{n}{2}) \leq \tfrac{1}{2}M(n)) \\
&= (1 + \tfrac{1}{2} + \cdots + \tfrac{1}{n/2})\tfrac{5}{6}M(n) + O(n) \\
&< 2(\tfrac{5}{6}M(n)) + O(n) \in \tfrac{5}{3}M(n) + O(n).
\end{aligned}
$$

□

This optimization reduces the constant 2 in Algorithm 13 to $\frac{5}{3}$.

3.1.5 Fast Division

Let $f \in F[x]$ be a dividend polynomial of degree m and $g \in F[x]$ be a divisor polynomial of degree n with a nonzero constant term. Then g^{-1} exists and we can compute $g^{-1} \mod x^{m-n+1}$ by applying Algorithm 14. Let c denote $g^{-1} \mod x^{m-n+1}$. Assume $r, q \in F[x]$ are polynomials satisfying $f = q \cdot g + r$ where $r = 0$ or $\deg(r) < \deg(g)$. It follows that

$$f \cdot c = q + r \cdot c \implies q = f \cdot c - r \cdot c.$$

Since the polynomial r is unknown, we are unable to compute $r \cdot c$ which might contain some terms of degree less than $m - n + 1$. Thus, we cannot obtain the quotient polynomial q.

To avoid this problem, we use the reciprocal polynomial introduced in Chapter 2. Let $f \in F[x]$ be a polynomial such that $f = f_n x^n + f_{n-1}x^{n-1} + \cdots + f_1 x + f_0$ with $f_n \neq 0$. According to Definition 2.22, the reciprocal of f is

$$f^{(rec)}(x) = x^n f\left(\frac{1}{x}\right) = f_0 x^n + f_1 x^{n-1} + \cdots + f_{n-1}x + f_n.$$

Thus, $f^{(rec)}$ has coefficients in reverse order of the coefficients of f. We know that $(f^{(rec)})^{-1}$ exists since $f_n \neq 0$, which implies that $f^{(rec)}$ has a nonzero constant term.

We note that if $f_0 \neq 0$, $\deg(f) = \deg(f^{(rec)})$. Furthermore, assuming $\deg(f^{(rec)}) = k$, we have

$$(f^{(rec)})^{(rec)}(x) = x^k f^{(rec)}\left(\frac{1}{x}\right) = x^k \left(\frac{1}{x}\right)^n f\left(\frac{1}{1/x}\right) = x^{k-n} f(x)$$

Thus we conclude that $(f^{(rec)})^{(rec)} = f$ if and only if $f_0 \neq 0$.

Theorem 3.10. *Let F be a field. Suppose $f, g \in F[x]$ are polynomials such that $\deg(f) = m$ and $\deg(g) = n$, where $m \geq n$. Let $q, r \in F[x]$ satisfy $f = g \cdot q + r$ where $r = 0$ or $\deg(r) < \deg(g)$. Then*

$$q^{(rec)} = f^{(rec)} \cdot (g^{(rec)})^{-1} \mod x^{m-n+1}.$$

Proof. Since $f = g \cdot q + r$, we have

$$\begin{aligned}
& f\left(\frac{1}{x}\right) = g\left(\frac{1}{x}\right) \cdot q\left(\frac{1}{x}\right) + r\left(\frac{1}{x}\right) \\
\implies & x^m f\left(\frac{1}{x}\right) = x^m \left(g\left(\frac{1}{x}\right) \cdot q\left(\frac{1}{x}\right) + r\left(\frac{1}{x}\right)\right) \\
\implies & x^m f\left(\frac{1}{x}\right) = x^n g\left(\frac{1}{x}\right) \cdot x^{m-n} q\left(\frac{1}{x}\right) + x^m r\left(\frac{1}{x}\right) \quad (\deg(q) = m - n) \\
\implies & f^{(rec)}(x) = g^{(rec)}(x) \cdot q^{(rec)}(x) + x^m r\left(\frac{1}{x}\right).
\end{aligned}$$

We consider two cases: $r(x) = 0$ and $r(x) \neq 0$. If $r(x) = 0$, it follows that $r(\frac{1}{x}) = 0$. Then we have

$$f^{(rec)}(x) = g^{(rec)}(x) \cdot q^{(rec)}(x).$$

Since $(g^{(rec)})^{-1}$ exists and $\deg(q^{(rec)}) \leq \deg(q) = m - n$,

$$q^{(rec)}(x) = f^{(rec)}(x) \cdot (g^{(rec)}(x))^{-1} \mod x^{m-n+1}.$$

If $r(x) \neq 0$, assume $\deg(r)$ is l. Then $0 \leq l < n$. We have

$$f^{(rec)}(x) = g^{(rec)}(x) \cdot q^{(rec)}(x) + x^{m-l}x^l r\left(\frac{1}{x}\right) = g^{(rec)}(x) \cdot q^{(rec)}(x) + x^{m-l}r^{(rec)}(x).$$

$(g^{(rec)})^{-1}$ exists, so it follows that

$$q^{(rec)}(x) = f^{(rec)}(x) \cdot (g^{(rec)}(x))^{-1} - x^{m-l}r^{(rec)}(x) \cdot (g^{(rec)}(x))^{-1}.$$

As $l < n$, this implies that $m - l > m - n$. Also, $r^{(rec)}(x) \neq 0$ as well. With Newton's method, $(g^{(rec)})^{-1}$ can be expressed as a power series to an order $m - n + 1$. Therefore, $\deg((g^{(rec)})^{-1}) > 0$. Then

$$\deg\left(x^{m-l}r^{(rec)}(x) \cdot (g^{(rec)}(x))^{-1}\right) \geq m - l > m - n.$$

Since $\deg(q^{(rec)}) \leq \deg(q) = m - n$, we can conclude that

$$\begin{aligned} q^{(rec)}(x) &= f^{(rec)}(x) \cdot (g^{(rec)}(x))^{-1} - x^{m-l}r^{(rec)}(x) \cdot (g^{(rec)}(x))^{-1} \mod x^{m-n+1} \\ &= f^{(rec)}(x) \cdot (g^{(rec)}(x))^{-1} \mod x^{m-n+1}. \end{aligned}$$

□

Based on Theorem 3.10, we can compute $q^{(rec)}$ by multiplying $f^{(rec)}$ by $(g^{(rec)})^{-1}$ and truncate this product to order $m - n + 1$. Since $\deg(q^{(rec)}) \leq m - n$, we reverse the order of coefficients in $q^{(rec)}$ to obtain q. In other words, $q = \sum_{i=0}^{m-n} q^*_{m-n-i}x^i$ when $q^{(rec)} = \sum_{i=0}^{m-n} q^*_i x^i$. Once we have obtained q, the remainder r can be computed using $r = f - g \cdot q$.

Example 3.11. Let $F = \mathbb{Z}_{17}$ and $f, g \in F[x]$ be a polynomial such that

$$\begin{aligned} f &= 1 + 2x + 3x^2 + 4x^3 \\ g &= 5 + 6x + 7x^2. \end{aligned}$$

We want to compute polynomials $r, q \in F[x]$ satisfying $f = g \cdot q + r$ where $r = 0$ or $\deg(r) < \deg(g)$. The reciprocals of f and g are

$$\begin{aligned} f^{(rec)} &= 4 + 3x + 2x^2 + x^3 \\ g^{(rec)} &= 7 + 6x + 5x^2. \end{aligned}$$

Since $\deg(q) = \deg(f) - \deg(g) = 3 - 2 = 1$, we truncate $f^{(rec)}$ and $g^{(rec)}$ to order 2. We have

$$\begin{aligned} f^{(rec)} \mod x^2 &= 4 + 3x \\ g^{(rec)} \mod x^2 &= 7 + 6x. \end{aligned}$$

Using Newton inversion, we compute $(g^{(rec)})^{-1}$ to order x^2 such that

$$(g^{(rec)})^{-1} \mod x^2 = 5 + 3x.$$

Then we multiply $f^{(rec)}$ by $(g^{(rec)})^{-1}$.

$$f^{(rec)} \cdot (g^{(rec)})^{-1} = 3 + 10x + 9x^2$$

According to Theorem 3.10, the order 2 approximation of this product is $q^{(rec)}$.

$$q^{(rec)} = 3 + 10x + 9x^2 \mod x^2 = 3 + 10x$$

Thus,

$$q = 10 + 3x.$$

Then r can be computed by

$$r = f - gq = 1 + 2x + 3x^2 + 4x^3 - (5 + 6x + 7x^2)(10 + 3x) = 2 + 12x.$$

Hence, $r = 2 + 12x$ and $q = 10 + 3x$.

Now, we present the fast division algorithm in Algorithm 15 based on Theorem 3.10.

Theorem 3.12. *Let $M(n)$ be the number of arithmetic operations in F for multiplying two polynomials where the sum of the degrees of these polynomials is less than $2n$. Assume $f \in F[x]$ is a dividend polynomial of degree $2n - 1$ and $g \in F[x]$ is a divisor polynomial of degree n. Then Algorithm 15 executes at most $\frac{11}{3}M(n) + O(n)$ arithmetic operations in F.*

Proof. Let $D(n)$ be the number of arithmetic operations in F that Algorithm 15 does to divide f by g. We can compute reciprocal polynomials by shifting coefficients and truncate polynomials to order n by reading off the coefficients up to degree $n - 1$. Thus, no arith-

Algorithm 15 Fast division

```
Input: Polynomials $f, g \in F[x]$ where $g \neq 0$.
Output: The remainder and quotient $r, q \in F[x]$ of $f \div g$ satisfying $f = g \cdot q + r$ where
    $r = 0$ or $\deg(r) < \deg(g)$
 1: $m \leftarrow \deg(f)$
 2: $n \leftarrow \deg(g)$
 3: if $m < n$ then return $f, 0$ end if
 4: $s \leftarrow m - n + 1$
 5: $a \leftarrow g^{(rec)} \mod x^s$
 6: $b \leftarrow f^{(rec)} \mod x^s$
 7: $c \leftarrow \text{NIwithMP}(a,\ s)$    // $c = a^{-1} \mod x^{m-n+1}$ .............................. $I(n)$
 8: $e \leftarrow b \cdot c$ using fast multiplication ........................................ $M(n)$
 9: $q^{(rec)} \leftarrow e \mod x^s$    // $q^{(rec)} = \sum_{i=0}^{m-n} q_i^* x^i$
10: $q \leftarrow \sum_{i=0}^{m-n} q_{m-n-i}^* x^i$
11: $M \leftarrow g \cdot q$ using fast multiplication ....................................... $M(n)$
12: $r \leftarrow f - M$
13: return $r, q$
```

metic operations are performed. We observe that Algorithm 14 Newton inversion with the middle product does $I(n)$ arithmetic operations in F to get an order $2n - 1 - n + 1 = n$ approximation. After executing Newton inversion, Algorithm 15 does one multiplication to compute $q^{(rec)} = b \cdot c$ in line 8. Since $\deg(b) = n - 1$ and $\deg(c) = n - 1$, it takes less than $M(n)$ arithmetic operations in F to compute $b \cdot c$. In line 11, there is another multiplication to compute $g \cdot q$, which does at most $M(n)$ arithmetic operations in F as well because $\deg(g) = n$ and $\deg(q) = n - 1$. Line 12 costs at most $n + 1$ subtractions in F. Thus, the cost of Algorithm 15 is

$$\begin{aligned} D(n) &= I\,(n) + M\,(n) + M\,(n) + O(n) \\ &\leq \tfrac{5}{3} M\,(n) + 2M(n) + O(n) \quad \text{(by Theorem 3.9)} \\ &= \tfrac{11}{3} M(n) + O(n). \end{aligned}$$

□

When we use fast multiplication, $M(n) = 9n \log_2 n + O(n)$, and hence, $D(n) = 33n \log_2 n + O(n) \in O(n \log n)$.

3.1.6 Optimizing Fast Division

We compare Algorithm 15 to Algorithm 11 in terms of timing. We will work over the field $F = \mathbb{Z}_p$ where $p = 3 \cdot 2^{30} + 1$ to implement both algorithms. Let n be the degree of divisor such that $n = 2^k$ for some $k \in \mathbb{N}$. Assume that the degree of dividend polynomial is $2n - 1$. We randomly chose the coefficients of the dividend and divisor polynomials from $[0, p)$.

n	Classical division	ratio	Fast division	ratio
2^6	0.010	-	0.037	-
2^7	0.027	2.70	0.093	2.51
2^8	0.086	3.19	0.186	2.00
2^9	0.291	3.38	0.403	2.17
2^{10}	1.05	3.60	0.809	2.01
2^{11}	4.27	4.07	1.76	2.18
2^{12}	18.5	4.33	3.92	2.23
2^{13}	64.5	3.49	9.30	2.37
2^{14}	262.0	4.06	20.9	2.24
2^{15}	1175.8	4.48	40.3	1.93
2^{16}	4493.5	3.82	89.3	2.22

Table 3.1: Timings in ms for classical division and fast division with a dividend polynomial of degree $2n-1$ and a divisor polynomial of degree n

The columns labelled "ratio" are the values of execution time with n divided by execution time with $\frac{n}{2}$. Table 3.1 shows that the execution time for the classical algorithm increases by a factor of 4 as n increases by a factor of 2. This is because the cost of the classical algorithm is $O(n^2)$.

Table 3.1 also shows that classical division is faster than fast division when $n < 1024$. However, for $n \geq 1024$, fast division is getting faster as n increases. It follows that it is better to use classical division for the degree of divisor is less than or equal to 512. Contrarily, fast division performs better than classical division for $n \geq 1024$. We adopt this result in our Algorithm 16 FastDiv.

Algorithm 16 Optimized fast division (FastDiv)

```
Input: Polynomials $f, g \in F[x]$ where $g \neq 0$.
Output: The remainder and quotient $r, q \in F[x]$ of $f \div g$ satisfying $f = g \cdot q + r$ where
    $r = 0$ or $\deg(r) < \deg(g)$
if $\deg(g) \cdot (\deg(f) - \deg(g)) \leq 2^{18}$ then
    $r, q \leftarrow$ Algorithm 11 Classical division ($f$,$g$)
    return $r, q$
end if
$r, q \leftarrow$ Algorithm 15 Fast division ($f$,$g$)
return $r, q$
```

3.2 Fast Multipoint Evaluation

3.2.1 Classical Evaluation Algorithm

Let F be a field. Let $f = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1} \in F[x]$ and $u_0, u_1, \ldots, u_{n-1}$ be distinct elements in F. To compute $f(u_i)$ for all $0 \leq i \leq n-1$, we can rewrite f using Horner's method as

$$f = f_0 + x(f_1 + x(f_2 + \cdots + x(f_{n-1})) \cdots)).$$

Then, for arbitrary i, computing $f(u_i) = f_0 + u_i(f_1 + u_i(f_2 + \cdots + u_i(f_{n-1})) \cdots))$ requires $n-1$ multiplications and $n-1$ additions. This implies that evaluating a polynomial of degree $n-1$ with Horner's method at one point takes $O(n)$ arithmetic operations in F. Thus, the cost of evaluating this polynomial at $u_0, u_1, \ldots, u_{n-1}$ is $O(n^2)$.

3.2.2 The Product Tree

In 1971, Borodin and Munro introduced a fast algorithm for polynomial evaluation at multipoints [2]. To implement this fast algorithm, a product tree should be constructed first. Suppose $u_0, u_1, \ldots, u_{n-1} \in F$ are distinct evaluation points where $n = 2^k$ for some $k \in \mathbb{N}$. We construct a complete binary tree T with these evaluation points. See Figure 3.1. In T, every leaf is a linear polynomial $x - u_i$ for $0 \leq i \leq n-1$. Each parent node is the product of their two children.

Let $T_{i,j}$ be a polynomial in $j-$th node from the left at height i in the tree T. Then $T_{k,0} = \prod_{i=0}^{n-1}(x - u_i)$, which is the root of T. Since F is a field and $u_0, u_1, \ldots, u_{n-1}$ are pairwise distinct, every $T_{i,j}$ is a monic square-free polynomial. Moreover, building up this product tree does not depend on the polynomial to be evaluated but on the evaluation points only. Thus, once we construct a product tree, we can use this product tree again for other polynomials to be evaluated at the same points.

Example 3.13. In $\mathbb{Z}_{97}$, suppose four evaluation points are $u_0 = 9, u_1 = 7, u_2 = 5$, and $u_3 = 3$. Then we can build up a product tree like the one below.

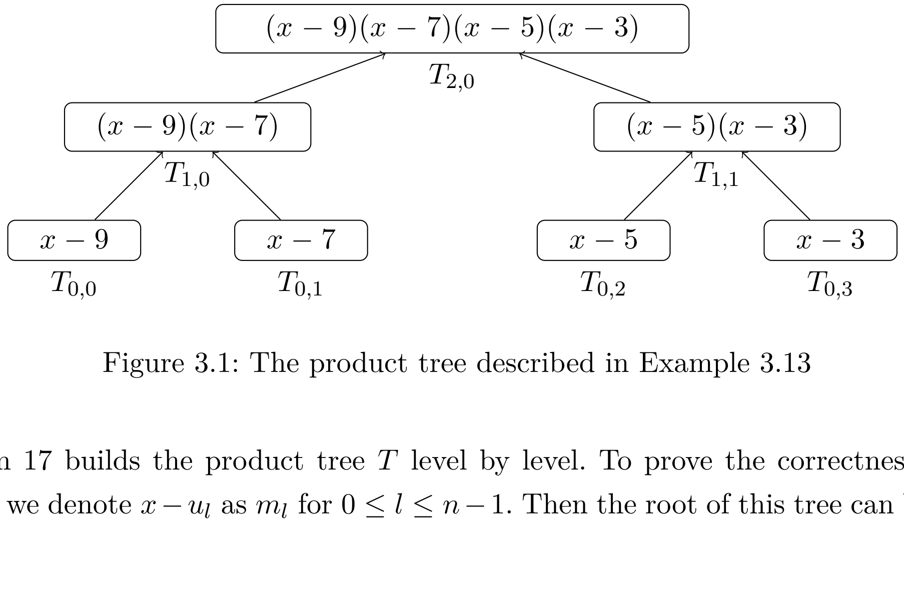

Figure 3.1: The product tree described in Example 3.13

Algorithm 17 builds the product tree T level by level. To prove the correctness of Algorithm 17, we denote $x - u_l$ as m_l for $0 \leq l \leq n-1$. Then the root of this tree can be written

Algorithm 17 Building up a product tree

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$ and $u_0, u_1, \ldots, u_{n-1} \in F$.
Output: A product tree $T$ containing polynomials $T_{i,j}$ for $0 \leq i \leq k$ and $0 \leq j < 2^{k-i}$.
for $j$ from 0 to $n-1$ do $T_{0,j} \leftarrow x - u_j$ end for
for $i$ from 1 to $k$ do    // $k = \log_2 n$
    for $j$ from 0 to $2^{k-i} - 1$ do
        $T_{i,j} \leftarrow T_{i-1,2j} \cdot T_{i-1,2j+1}$
    end for
end for
return $T$
```

as $T_{k,0} = \prod_{l=0}^{n-1} m_l$. Also, each $T_{i,j}$ can be expressed as

$$\begin{aligned} T_{i,j} &= (x - u_{2^i \cdot j})(x - u_{2^i \cdot j+1}) \cdots (x - u_{2^i \cdot j+(2^i-1)}) \\ &= m_{2^i \cdot j} \cdot m_{2^i \cdot j+1} \cdots m_{2^i \cdot j+(2^i-1)}. \end{aligned}$$

This implies that every $T_{i,j}$ is a product of 2^i subsequent linear polynomials from $m_{2^i \cdot j}$ to $m_{2^i \cdot j+(2^i-1)}$, which satisfies the recurrence with $T_{0,j} = m_j$ and $T_{i,j} = T_{i-1,2j} \cdot T_{i-1,2j+1}$ for $1 \leq i \leq k$ and $0 \leq j \leq 2^{k-i} - 1$.

Theorem 3.14. *Let $M(n)$ be the number of arithmetic operations in F for multiplying two polynomials where the sum of the degrees of these polynomials is less than $2n$. Then Algorithm 17 does $M(n) \log_2 n + O(n)$ arithmetic operations in F.*

Proof. Let $B(n)$ be the number of arithmetic operations to construct a product tree with n evaluation points. The first for loop does n negations. After the first for loop, Algorithm 17 runs a nested loop. The outer for loop iterates $k = \log_2 n$ times and the inner for loop iterates 2^{k-i} times. In each iteration of the inner for loop, Algorithm 17 multiplies two polynomials of degree 2^{i-1}. The sum of the degrees of these two polynomials is not less than 2^i but less than 2^{i+1}. It follows that each iteration of the inner loop does at most $M(2^i)$ arithmetic operations in F. Thus,

$$\begin{aligned} B(n) &\leq n + \sum_{i=1}^{k} 2^{k-i} M(2^i) \\ &= M\left(n\right) + 2M\left(\tfrac{n}{2}\right) + 2^2 M\left(\tfrac{n}{4}\right) + \cdots + 2^{k-1} M(2) + n. \end{aligned}$$

By Corollary 2.15, $2M(\frac{n}{2}) \leq M(n)$. Then we have

$$\begin{aligned} B(n) &\leq M(n) + M(n) + \cdots + M(n) + n \\ &= k \cdot M(n) + n \\ &= M(n) \log_2 n + O(n). \end{aligned}$$

□

By Theorem 2.14, Algorithm 7 does $M(n) = 9n\log_2 n + O(n)$ arithmetic operations in F. With fast multiplication, Algorithm 17 does $M(n)\log_2 n + O(n) = 9n\log_2^2 n + O(n\log n) \in O(n\log^2 n)$ arithmetic operations in F.

3.2.3 Dividing Down the Product Tree

Let $f \in F[x]$ be a polynomial of degree at most $n-1$. Using the product tree, we can evaluate f based on the Chinese Remainder Theorem over $F[x]$. Let $m_i = x - u_i$ for all $0 \leq i \leq n-1$. Let $f \mod m_i$ be the remainder of f divided by m_i. Since m_i is a linear polynomial, $f \mod m_i$ is a constant. When we evaluate f at $x = u_i$, we have

$$f(u_i) = q(u_i)m_i(u_i) + r(u_i) = r(u_i) = q(u_i)(u_i - u_i) + r(u_i) = r(u_i)$$

This implies that $f \mod m_i = f(u_i)$. Thus we can evaluate a polynomial at n distinct points by dividing the polynomial by n linear polynomials. The cost of these divisions is $O(n^2)$ since each division does $3n-3$ arithmetic operations in F by Corollary 3.3. We can use the product tree instead of n linear divisions because of the following lemma.

Lemma 3.15. *Let $f, g, h \in F[x]$ where $g|h$. Then $f \mod g = (f \mod h) \mod g$.*

Proof. Dividing f by h gives

$$f = q \cdot h + r$$

for some $q, r \in F[x]$ where $r = 0$ or $\deg(r) < \deg(h)$. It follows that $f \mod h = r$. Moreover, let $s \in F[x]$ be a polynomial such that $h = g \cdot s$. Then we have

$$\begin{aligned} f \mod g &= (q \cdot h + r) \mod g \\ &= (q \cdot (g \cdot s) + r) \mod g \\ &= r \mod g \\ &= (f \mod h) \mod g. \end{aligned}$$

□

We note that each node in T is a factor of its parent node in T. In other words,

$$T_{i,j} = T_{i-1,2j} \cdot T_{i-1,2j+1} \implies T_{i-1,2j}|T_{i,j} \text{ and } T_{i-1,2j+1}|T_{i,j}.$$

Using Lemma 3.15, we can divide down the product tree to compute polynomial multipoint evaluation. This division down the product tree is implemented with a divide-and-conquer approach.

Example 3.16. Assume $n = 4$. Let $f(x) = 1 + 2x + 3x^2 + 4x^3 \in \mathbb{Z}_{97}[x]$. Given the product tree in Example 3.13, we want to evaluate $f(x)$ at $u_0 = 9$, $u_1 = 7$, $u_2 = 5$, and $u_3 = 3$. This algorithm computes $f \mod T_{1,0}$ and $f \mod T_{1,1}$. At this step, we have

$$\begin{aligned} r_0 &= f \mod T_{1,0} = f \mod (x-9)(x-7) = 46x + 48 \text{ and} \\ r_1 &= f \mod T_{1,1} = f \mod (x-5)(x-3) = 28x + 58. \end{aligned}$$

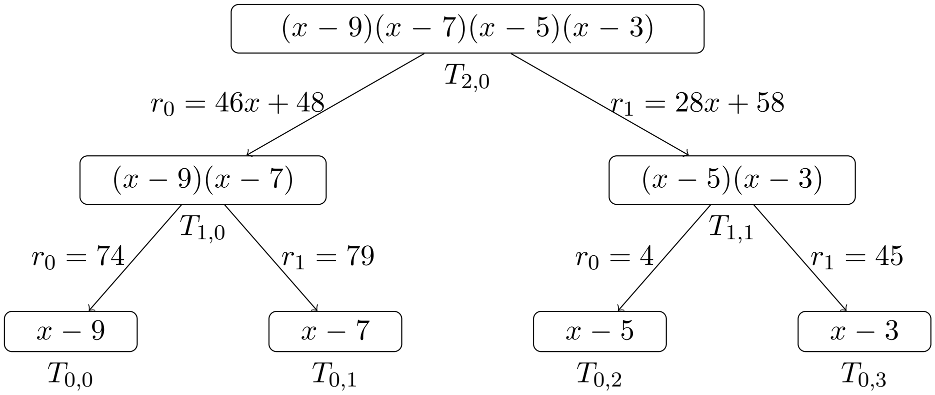

Figure 3.2: Dividing down the product tree described in Example 3.16

After computing two remainders, this algorithm makes two recursive calls. One is with input $\frac{n}{2} = 2$, r_0, and the tree rooted at $T_{1,0}$. This recursive call outputs

$$\begin{aligned} r_0 &= 46x + 48 \mod T_{0,0} = 46x + 48 \mod (x-9) = 74 \text{ and} \\ r_1 &= 46x + 48 \mod T_{0,1} = 46x + 48 \mod (x-7) = 79. \end{aligned}$$

The other recursive call is with input $\frac{n}{2} = 2$, r_0, and the tree rooted at $T_{1,1}$. It outputs

$$\begin{aligned} r_0 &= 28x + 58 \mod T_{0,2} = 28x + 58 \mod (x-5) = 4 \text{ and} \\ r_1 &= 28x + 58 \mod T_{0,3} = 28x + 58 \mod (x-3) = 45. \end{aligned}$$

Thus, we have $f(9) = 74$, $f(7) = 79$, $f(5) = 4$, and $f(3) = 45$.

Algorithm 18 computes all $f(u_i)$ for $0 \leq i \leq n-1$.

The correctness of Algorithm 18 can be shown by induction on $k = \log_2 n$. For $k = 0$, f is a constant since the input polynomial f is of degree less than n and $n = 1$. For an inductive hypothesis, we assume that Algorithm 18 computes the correct values for $k = m - 1 \geq 0$. Now consider when $k = m$. Assume $f \in F[x]$ is a polynomial of degree at most $n-1$ where $n = 2^m$. Let q_0 be the quotient of f divided by $T_{m-1,0}$. Since $T_{m-1,0} = \prod_{i=0}^{\frac{n}{2}-1}(x - u_i)$,

Algorithm 18 Dividing down the product tree

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $f \in F[x]$ of degree less than $n$, and the product tree $T$ built
    up with $u_0, u_1, \ldots, u_{n-1} \in F$.
Output: $f(u_0), f(u_1), \ldots, f(u_{n-1}) \in F$
if $n = 1$ then return $f \in F$ end if
$r_0 \leftarrow f \mod T_{k-1,0}$    // divide $f$ by $\prod_{i=0}^{\frac{n}{2}-1}(x - u_i)$
$r_1 \leftarrow f \mod T_{k-1,1}$    // divide $f$ by $\prod_{i=\frac{n}{2}}^{n-1}(x - u_i)$
$r_0(u_0), \ldots, r_0(u_{\frac{n}{2}-1}) \leftarrow$ Algorithm 18 Dividing down the product tree ($\frac{n}{2}$, $r_0$, the tree
    rooted at $T_{k-1,0}$)
$r_1(u_{\frac{n}{2}}), \ldots, r_1(u_{n-1}) \leftarrow$ Algorithm 18 Dividing down the product tree ($\frac{n}{2}$, $r_1$, and the
    tree rooted at $T_{k-1,1}$)
return $r_0(u_0), \ldots, r_0(u_{\frac{n}{2}-1}), r_1(u_{\frac{n}{2}}), \ldots, r_1(u_{n-1})$
```

evaluating f at u_i for $0 \leq i \leq \frac{n}{2} - 1$ gives

$$f(u_i) = q_0(u_i)T_{m-1,0} + r_0(u_i) = q_0(u_i) \cdot 0 + r_0(u_i) = r_0(u_i).$$

Also, let q_1 be the quotient of f divided by $T_{m-1,1} = \prod_{i=\frac{n}{2}}^{n-1}(x - u_i)$. Similarly,

$$f(u_i) = q_1(u_i)T_{m-1,1} + r_1(u_i) = q_1(u_i) \cdot 0 + r_1(u_i) = r_1(u_i)$$

for $\frac{n}{2} \leq i \leq n - 1$. Then we evaluate r_0 and r_1 at $\frac{n}{2} = 2^{m-1}$ distinct evaluation points using recursive calls. Correctness follows immediately.

Theorem 3.17. *Let $M(n)$ be the number of arithmetic operations in F for polynomial multiplication of two polynomials where the sum of the degrees of these polynomials less than $2n$. Also, let $D(n)$ be the number of arithmetic operations in F that polynomial division does with a dividend polynomial of degree $2n - 1$ and a divisor polynomial of degree n. Then Algorithm 18 does $\frac{11}{3}M(n)\log_2 n + O(n \log n)$ arithmetic operations in F with input $f \in F[x]$ of degree less than n.*

Proof. Let $C(n)$ be the number of arithmetic operations in F that Algorithm 18 does with input polynomial f of degree at most $n - 1$. When $n = 1$, no operations occur, and hence, $C(1) = 0$. Otherwise, Algorithm 18 does two divisions by $T_{k-1,0}$ and $T_{k-1,1}$ first. Both $T_{k-1,0}$ and $T_{k-1,1}$ are of degree $\frac{n}{2}$, so each division does at most $D(\frac{n}{2})$ arithmetic operations in F. Then Algorithm 18 makes two recursive calls of size $\frac{n}{2}$ with r_0 and r_1 of degree at most $\frac{n}{2} - 1$. This implies that

$$\begin{cases} C(1) = 0 \\ C(n) \leq 2D(\frac{n}{2}) + 2C(\frac{n}{2}) \text{ for } n > 1. \end{cases}$$

It follows that

$$C(n) \leq 2D\left(\tfrac{n}{2}\right) + 4D\left(\tfrac{n}{4}\right) + 8D\left(\tfrac{n}{8}\right) + \cdots + 2^k D(1).$$

Since $2M(\frac{n}{2}) \leq M(n)$ by Corollary 2.15, it follows that $2D(\frac{n}{2}) \leq D(n)$. As a result,

$$\begin{aligned} C(n) &\leq D(n) + D(n) + \cdots + D(n) \\ &= kD(n) = D(n)\log_2 n. \end{aligned}$$

From Theorem 3.12, fast division does $D(n) = \frac{11}{3}M(n) + O(n)$ arithmetic operations in F with a dividend of degree $2n-1$ and a divisor of degree n. Thus,

$$C(n) \leq D(n)\log_2 n = \frac{11}{3}M(n)\log_2 n + O(n\log n).$$

□

With fast multiplication, $M(n) = 9n\log_2 n + O(n)$. Thus, $C(n) = 33n\log_2^2 n + O(n\log n) \in O(n\log^2 n)$ arithmetic operations in F.

Remark 3.18. *The cost of dividing down the product tree is $\frac{11}{3}$ times as expensive as the cost of building up the product tree even with the optimizations to polynomial divisions.*

3.2.4 Fast Multipoint Evaluation

We can combine the two procedures above: constructing a product tree and dividing down the product tree to obtain Borodin and Munro's fast multipoint evaluation algorithm [2].

Algorithm 19 Fast multipoint evaluation

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $f \in F[x]$ of degree less than $n$, and $u_0, u_1, \ldots, u_{n-1} \in F$.
Output: $f(u_0), f(u_1), \ldots, f(u_{n-1}) \in F$
Construct the product tree $T$ using Algorithm 17 with inputs $n$, and $u_0, u_1, \ldots, u_{n-1}$
Compute the evaluations $f(u_0), f(u_1), \ldots, f(u_{n-1})$ using Algorithm 18 Dividing down
   the product tree with inputs $n$, $f$, and $T$
return $f(u_0), f(u_1), \ldots, f(u_{n-1})$
```

We note that the root $T_{k,0} = \prod_{i=0}^{n-1}(x - u_i)$ of the product tree has degree n. Since Algorithm 19 takes the input f of degree at most $n-1$, the root $T_{k,0}$ is not used for polynomial divisions. However, this root polynomial will later be used for the fast transposed Vandermonde solver algorithm. Therefore, we build the entire product tree with the root polynomial $T_{k,0}$ in this algorithm.

Theorem 3.19. *Let $M(n)$ be the number of arithmetic operations in F for multiplying two polynomials where the sum of the degrees of these polynomials is less than $2n$. Also, let $D(n)$ be the number of arithmetic operations in F that polynomial division does with a dividend* *polynomial of degree $2n-1$ and a divisor polynomial of degree n. Then Algorithm 19 does $\frac{14}{3}M(n)\log_2 n + O(n\log n)$ arithmetic operations in F.*

Proof. Let $E(n)$ be the number of arithmetic operations in F for Algorithm 19. Algorithm 19 calls both Algorithm 17 and Algorithm 18 once. Let $B(n)$ be the cost of building a product tree with n evaluation points and $C(n)$ be the cost of dividing down the product tree with a polynomial degree less than n. It follows that $E(n) = B(n) + C(n)$.

Using fast multiplication and fast division, $B(n) = M(n)\log_2 n + O(n)$ by Theorem 3.14. Likewise, by Theorem 3.17, $C(n) = \frac{11}{3}M(n)\log_2 n + O(n\log n)$. Then we have

$$\begin{aligned} E(n) &= B(n) + C(n) \\ &\le M(n)\log_2 n + O(n) + \frac{11}{3}M(n)\log_2 n + O(n\log n) \\ &= \frac{14}{3}M(n)\log_2 n + O(n\log n). \end{aligned}$$

□

By Theorem 2.14, fast multiplication with two polynomials where the sum of the degrees of these polynomials is at most $2n-1$ does $M(n) = 9n\log_2 n + O(n)$ arithmetic operations in F. This implies that Algorithm 19 does $42n\log_2^2 n + O(n\log n) \in O(n\log^2 n)$ arithmetic operations in F.

3.2.5 Optimizing Fast Multipoint Evaluation

We discuss our optimized fast multipoint evaluation algorithm based on Algorithm 19. Algorithm 19 calls Algorithm 17 Building up a product tree first. Algorithm 17 starts from the leaves of the tree. At the i-th step, this algorithm multiplies a pair of polynomials of degree 2^{i-1}, and there exist $\frac{n}{2^i} = 2^{k-i}$ pairs of such polynomials. Since the product of two polynomials is a polynomial of degree 2^i, fast multiplication uses the FFT of size 2^{i+1}. To improve efficiency, we cut down the size of the FFTs. The following simple optimization will speed up the construction of the product tree by approximately a factor of 2.

Consider $T_{i-1,2j}$ and $T_{i-1,2j+1}$ for some j. We can write these two polynomials as

$$\begin{aligned} T_{i-1,2j} &= a_0 + a_1x + \cdots + a_{2^{i-1}-1}x^{2^{i-1}-1} + x^{2^{i-1}} \\ T_{i-1,2j+1} &= b_0 + b_1x + \cdots + b_{2^{i-1}-1}x^{2^{i-1}-1} + x^{2^{i-1}} \end{aligned}$$

Then we can compute $T_{i,j}$ by separating the leading term and the remaining terms.

$$\begin{aligned} T_{i,j} &= T_{i-1,2j} \cdot T_{i-1,2j+1} \\ &= \left(a_0 + a_1 x + \cdots + a_{2^{i-1}-1} x^{2^{i-1}-1} + x^{2^{i-1}}\right) \cdot \left(b_0 + b_1 x + \cdots + b_{2^{i-1}-1} x^{2^{i-1}-1} + x^{2^{i-1}}\right) \\ &= \left((a_0 + a_1 x + \cdots + a_{2^{i-1}-1} x^{2^{i-1}-1}) + x^{2^{i-1}}\right) \cdot \left((b_0 + b_1 x + \cdots + b_{2^{i-1}-1} x^{2^{i-1}-1}) + x^{2^{i-1}}\right) \end{aligned}$$

Let $A, B \in F[x]$ be a polynomial such that

$$\begin{aligned} A &= a_0 + a_1 x + \cdots + a_{2^{i-1}-1} x^{2^{i-1}-1}, \\ B &= b_0 + b_1 x + \cdots + b_{2^{i-1}-1} x^{2^{i-1}-1}. \end{aligned}$$

It follows that

$$T_{i,j} = (A + x^{2^{i-1}}) \cdot (B + x^{2^{i-1}}) = A \cdot B + (A + B) \cdot x^{2^{i-1}} + x^{2^i}.$$

The leading term of every polynomial $T_{i,j}$ in the product tree will be x^{2^i} for $0 \le i \le k$, and its coefficient is 1. We do not have to compute the leading term of $T_{i,j}$, and hence, we do not store all the leading terms of polynomials in the product tree. Also, we are able to compute $(A + B) \cdot x^{2^{i-1}}$ by adding and shifting coefficients in A and B, which only takes $O(n)$ arithmetic operations in F. Then we have one polynomial multiplication of $A \cdot B$ where A and B are of degree $2^{i-1} - 1$. Thus, we can use one fast multiplication with the FFT of size 2^i instead of 2^{i+1}.

Algorithm 20 modifies Algorithm 17 to be recursive.

Algorithm 20 Recursive building up a product tree

Input: $n = 2^k$ for some $k \in \mathbb{N}$ and $u_0, u_1, \ldots, u_{n-1} \in F$.
Output: A product tree T containing polynomials $T_{i,j}$ for $0 \le i \le k$ and $0 \le j < 2^{k-i}$.

```
if $n = 1$ then return $-u_j$ end if
A tree rooted at $T_{k-1,0} \leftarrow$ Algorithm 20 with inputs $2^{k-1}$ and $u_0, \ldots, u_{\frac{n}{2}-1}$ ..... $B(\frac{n}{2})$
A tree rooted at $T_{k-1,1} \leftarrow$ Algorithm 20 with inputs $2^{k-1}$ and $u_{\frac{n}{2}}, \ldots, u_{n-1}$ ..... $B(\frac{n}{2})$
$A \leftarrow T_{k-1,0}$     // $A = \sum_{i=0}^{\frac{n}{2}-1} a_i x^i$
$B \leftarrow T_{k-1,1}$     // $B = \sum_{i=0}^{\frac{n}{2}-1} b_i x^i$
$C \leftarrow A \cdot B$ using fast multiplication ............................................ $M(\frac{n}{2})$
$D \leftarrow A + B$
$T_{k,0} \leftarrow C + D \cdot x^{2^{k-1}}$
return $T$
```

Theorem 3.20. *Algorithm 20 does $\frac{1}{2}M(n)\log_2 n + O(n \log n)$ arithmetic operations in F, where $M(n)$ is the number of arithmetic operations in F for multiplying two polynomials where the sum of the degrees of these polynomials is less than $2n$.*

Proof. Let $B(n)$ be the number of arithmetic operations in F for Algorithm 20 with n distinct evaluation points. $B(1) = 1$ for the negation in line 1. For $n > 1$, two recursive calls in lines 2 and 3 cost $2B(\frac{n}{2})$. In line 6, Algorithm 20 does polynomial multiplication where the sum of the degrees of two polynomials is at most $n-2$. It takes at most $M(\frac{n}{2})$ arithmetic operations in F to multiply two polynomials. Also, Algorithm 20 does $O(n)$ additions in lines 7 and 8. Therefore,

$$\begin{cases} B(1) = 1 \\ B(n) = 2B(\frac{n}{2}) + M(\frac{n}{2}) + O(n) \text{ for } n > 1. \end{cases}$$

By Corollary 2.15, this recurrence relation results in

$$\begin{aligned} B(n) &= M\left(\tfrac{n}{2}\right) + 2M\left(\tfrac{n}{4}\right) + \cdots + 2^{k-1}M\left(1\right) + kO(n) + n \\ &\leq \frac{1}{2}\left(M(n) + M(n) + \cdots + M(n)\right) + O(n \log n) \\ &= \frac{1}{2}\left(k \cdot M(n)\right) + O(n \log n) \\ &= \frac{1}{2}M(n)\log_2 n + O(n \log n) \in O(M(n) \log n). \end{aligned}$$

□

Now we compare the execution time for the classical evaluation algorithm and Algorithm 19 Fast evaluation. We will work over the field $F = \mathbb{Z}_p$ where $p = 3 \cdot 2^{30} + 1$. Let n be the number of evaluation points where $n = 2^k$ for some $k \in \mathbb{N}$. We generated a polynomial $f \in F[x]$ of degree $n-1$ to be evaluated by randomly choosing its coefficients from $[0, p)$. Also, the distinct evaluation points $u_0, u_1, \ldots, u_{n-1}$ are randomly chosen from $[0, p)$.

n	Classical evaluation	ratio	Fast evaluation	ratio
2^6	0.043	-	0.054	-
2^7	0.161	3.74	0.130	2.41
2^8	0.498	3.09	0.350	2.69
2^9	2.003	4.02	0.734	2.10
2^{10}	8.808	4.39	2.149	2.93
2^{11}	34.94	3.96	6.223	2.90
2^{12}	132.1	3.78	16.16	2.60
2^{13}	509.8	3.85	43.36	2.68
2^{14}	2098.9	4.11	98.06	2.26
2^{15}	7942.0	3.78	252.8	2.58
2^{16}	34001.6	4.28	562.8	2.23

Table 3.2: Timings in ms for classical evaluation and fast evaluation with a polynomial of degree $n-1$ at n distinct evaluation points

The columns labelled "ratio" contain the values of execution time with n divided by execution time with $\frac{n}{2}$. Table 3.2 shows that the execution time for the classical $O(n^2)$ algorithm increases by a factor of 4 as n is doubled. Moreover, Table 3.2 shows that for $n \leq 64$, classical evaluation using Horner's method is faster than Algorithm 19 Fast evaluation. When $n > 64$, Algorithm 19 is much faster, and the gap between two polynomial multipoint evaluation algorithms increases as n grows. By reflecting on this result, we change all the required algorithms for fast multipoint evaluation, as shown below.

Algorithm 21 Optimized building up a product tree (BUPT)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$ and $[u_0, u_1, \ldots, u_{n-1}] \in F^n$.
Output: A modified product tree $T$ containing polynomials $T_{i,j}$ for $6 \leq i \leq k$ and $0 \leq j < 2^{k-i}$
if $n \leq 64$ then        // $T_{k,0} = \prod_{i=0}^{n-1}(x - u_i) - x^n$
    $M \leftarrow x - u_0$
    for $i$ from 1 to $n - 1$ do ............................ $O(n^2)$
        $M \leftarrow M \cdot (x - u_i)$
    end for
    $T_{k,0} \leftarrow M - x^n$
    return $T_{k,0}$
end if
A tree rooted at $T_{k-1,0} \leftarrow$ BUPT($2^{k-1}$, $[u_0, \ldots, u_{\frac{n}{2}-1}]$) ............ $B(\frac{n}{2})$
A tree rooted at $T_{k-1,1} \leftarrow$ BUPT($2^{k-1}$, $[u_{\frac{n}{2}}, \ldots, u_{n-1}]$) ............ $B(\frac{n}{2})$
$A \leftarrow T_{k-1,0}$        // $A = \sum_{i=0}^{\frac{n}{2}-1} a_i x^i$
$B \leftarrow T_{k-1,1}$        // $B = \sum_{i=0}^{\frac{n}{2}-1} b_i x^i$
$C \leftarrow A \cdot B$ using fast multiplication ............................ $M(\frac{n}{2})$
$D \leftarrow A + B$
$T_{k,0} \leftarrow C + D \cdot x^{2^{k-1}}$
return $T$
```

Algorithm 22 Optimized Dividing down the product tree (DDPT)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $f \in F[x]$ of degree less than $n$, and the product tree $T$ built up with $u_i$ for all $0 \leq i \leq n - 1$.
Output: $f(u_0), f(u_1), \ldots, f(u_{n-1}) \in F$
if  $n \leq 64$ then
    for $i$ from 0 to $n - 1$ do
        $f(u_i) \leftarrow$ Call classical polynomial evaluation with inputs $f$ and $u_i$ ........ $O(n)$
    end for
    return $f(u_0), f(u_1), \ldots, f(u_{n-1})$
end if
$r_0 \leftarrow f$  mod $T_{k-1,0}$ ............................................ $D(\frac{n}{2})$
$r_1 \leftarrow f$  mod $T_{k-1,1}$ ............................................ $D(\frac{n}{2})$
$C \leftarrow$ DDPT($\frac{n}{2}$, $r_0$, the tree rooted at $T_{k-1,0}$)  // $C = [r_0(u_0), \ldots, r_0(u_{\frac{n}{2}-1})]$ ..... $C(\frac{n}{2})$
$D \leftarrow$ DDPT($\frac{n}{2}$, $r_1$, the tree rooted at $T_{k-1,1}$)  // $D = [r_1(u_{\frac{n}{2}}), \ldots, r_1(u_{n-1})]$ .... $C(\frac{n}{2})$
return $r_0(u_0), \ldots, r_0(u_{\frac{n}{2}-1}), r_1(u_{\frac{n}{2}}), \ldots, r_1(u_{n-1})$
```

Algorithm 21 computes $\prod_{i=0}^{n-1}(x-u_i)-x^n$ for $n \le 64$. In line 2, it takes one negation to initialize M. In the for loop, each iteration does one negation, $2i$ multiplications, and $i-1$ additions in F. Hence, each iteration does $3i$ arithmetic operations in F. Then it takes

$$1+\sum_{i=1}^{n-1} 3i = 3\left(\frac{n(n-1)}{2}\right)+1 = \frac{3}{2}n^2-\frac{3}{2}n+1 \in O(n^2)$$

to compute $T_{k,0}$ when $n \le 64$. On the other hand, for $n > 64$, Algorithm 21 constructs the modified product tree using two recursive calls like Algorithm 20.

Similarly, in Algorithm 22 DDPT, we implement the classical evaluation algorithm using Horner's method for $n \le 64$. As a result, our DDPT does not divide down further. Horner's method does $O(n^2)$ arithmetic operations in F for polynomial multipoint evaluation with a polynomial of degree $n-1$ at n distinct points. When $n > 64$, Algorithm 22 DDPT makes two recursive calls after computing two fast divisions as Algorithm 18 behaves.

Algorithm 23 Optimized fast multipoint evaluation (FastEval)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $f \in F[x]$ of degree less than $n$, and $u_0, u_1, \ldots, u_{n-1} \in F$.
Output: $f(u_0), f(u_1), \ldots, f(u_{n-1}) \in F$
if $n \le 64$ then
    for $i$ from 0 to $n-1$ do ........................................ $O(n^2)$
        $f(u_i) \leftarrow$ Call classical polynomial evaluation with inputs $f$ and $u_i$
    end for
    return $f(u_0), f(u_1), \ldots, f(u_{n-1})$
end if
$T \leftarrow$ BUPT($n$, $u_0, u_1, \ldots, u_{n-1}$) ........................................ $B(n)$
$f(u_0), f(u_1), \ldots, f(u_{n-1}) \leftarrow$ DDPT($n$,$f$, $T$) ........................................ $C(n)$
return $f(u_0), f(u_1), \ldots, f(u_{n-1})$
```

Algorithm 23 executes the classical evaluation algorithm using Horner's method for $n \le 64$. Otherwise, Algorithm 23 calls our BUPT and DDPT algorithms to evaluate a polynomial of degree $n-1$ at n distinct points.

Theorem 3.21. *Let $M(n)$ be the number of arithmetic operations in F for multiplying two polynomials where the sum of the degrees of these polynomials is less than $2n$. Also, let $D(n)$ be the number of arithmetic operations in F that polynomial division does with a dividend polynomial of degree $2n-1$ and a divisor polynomial of degree n. For $n > 64$, Algorithm 23 does $\frac{25}{6}M(n)\log_2 n + O(n \log n)$ arithmetic operations in F.*

Proof. Let $E(n)$ be the number of arithmetic operations in F for Algorithm 23 when $n > 64$. Algorithm 23 calls both Algorithm 21 which costs $B(n)$ and Algorithm 22 which costs $C(n)$ once. It follows that $E(n) = B(n) + C(n)$. $B(n) = \frac{1}{2}M(n)\log_2 n + O(n)$ by Theorem 3.14

and $C(n) = \frac{11}{3}M(n)\log_2 n + O(n \log n)$ by Theorem 3.17. Then we have

$$E(n) = \frac{25}{6}M(n)\log_2 n + O(n \log n).$$

□

Since fast multiplication does $M(n) = 9n\log_2 n + O(n)$ arithmetic operations in F by Theorem 2.14, Algorithm 23 does $\frac{75}{2}n\log_2^2 n + O(n \log n) \in O(n \log^2 n)$ arithmetic operations in F.

3.3 Fast Transposed Vandermonde Solver

3.3.1 Zippel's Transposed Vandermonde Solver

Let $a \in F[x]$ be a polynomial such that $a(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ with unknown coefficients $a_0, a_1, \ldots, a_{n-1}$. Let $\alpha \in F$ be $\alpha^i \neq \alpha^j$ for all $i \neq j$ and satisfy $b_i = a(\alpha^{i-1})$ for $1 \leq i \leq n$. Let $u_i = \alpha^{i-1}$ for $1 \leq i \leq n$. Then we have

$$b_i = a(\alpha^{i-1}) = \sum_{j=0}^{n-1} a_j(\alpha^{i-1})^j = \sum_{j=0}^{n-1} a_j(\alpha^j)^{i-1} = \sum_{j=1}^{n} a_j(\alpha^{j-1})^{i-1} = \sum_{j=1}^{n} a_j(u_j)^{i-1}.$$

We express these equations as $U\mathbf{a} = \mathbf{b}$, where

$$U = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ u_1 & u_2 & \cdots & u_n \\ \vdots & \vdots & & \vdots \\ u_1^{n-1} & u_2^{n-1} & \cdots & u_n^{n-1} \end{bmatrix}, \quad \mathbf{a} = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}, \quad \text{and } \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$U\mathbf{a} = \mathbf{b}$ is a transposed Vandermonde system of equations. In 1990, Zippel presented an $O(n^2)$ algorithm to solve a transposed Vandermonde system of equations [15]. We present Zippel's method and construct Kaltofen and Yagati's fast method from Zippel's method.

Lemma 3.22. *Let V be a $n \times n$ Vandermonde matrix such that*

$$V = \begin{bmatrix} 1 & u_1 & \cdots & u_1^{n-1} \\ 1 & u_2 & \cdots & u_2^{n-1} \\ \vdots & \vdots & & \vdots \\ 1 & u_n & \cdots & u_n^{n-1} \end{bmatrix}.$$

Then

$$\det(V) = \prod_{1 \leq i < j \leq n} (u_j - u_i).$$

Proof. See the proof in [5]. □

Corollary 3.23. *Let U be a transposed Vandermonde matrix of size $n \times n$. Then*

$$\det(U) \neq 0 \textit{ if and only if } u_i \neq u_j, \quad \forall i \neq j.$$

Proof. ($\Leftarrow$) Assume all $u_1, u_2, \ldots, u_n$ are distinct. Consider $V = U^\top$, which is a Vandermonde matrix. According to Lemma 3.22, $\det(V) = \prod_{1 \leq i < j \leq k}(u_j - u_i)$. Since $u_i \neq u_j$ for all $i \neq j$, $\det(V) \neq 0$. Due to the fact that $\det(V) = \det(V^\top)$, $\det(U)$ cannot be zero as well.

($\Rightarrow$) Assume $\det(U) \neq 0$. Then $\det(U) = \det(U^\top) = \det(V) \neq 0$. It follows that $\det(V) = \prod_{1 \leq i < j \leq k}(u_j - u_i) \neq 0$. Therefore, $u_i \neq u_j$ for all $i \neq j$. □

By Corollary 3.23, a transposed Vandermonde matrix U must have a unique inverse. Let

$$U^{-1} = \begin{bmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,n} \\ \vdots & \vdots & & \vdots \\ s_{n,1} & s_{n,2} & \cdots & s_{n,n} \end{bmatrix}$$

Then we can compute $\mathbf{a}$ by solving

$$\mathbf{a} = U^{-1}\mathbf{b}.$$

To compute U^{-1}, let $p_i \in F[x]$ be a polynomial such that $p_i(x) = s_{i,1} + s_{i,2}x + \cdots + s_{i,n}x^{n-1}$ for $1 \leq i \leq n$. From the fact that $U^{-1}U = I$,

$$\underbrace{\begin{bmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,n} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,n} \\ \vdots & \vdots & & \vdots \\ s_{n,1} & s_{n,2} & \cdots & s_{n,n} \end{bmatrix}}_{U^{-1}} \underbrace{\begin{bmatrix} 1 & 1 & \cdots & 1 \\ u_1 & u_2 & \cdots & u_n \\ \vdots & \vdots & & \vdots \\ u_1^{n-1} & u_2^{n-1} & \cdots & u_n^{n-1} \end{bmatrix}}_{U} = \underbrace{\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}}_{I}$$

Each entry $I_{i,j}$ in I is computed by multiplying the i-th row of U^{-1} by the j-th column of U. Observe that

$$I_{i,j} = \begin{bmatrix} s_{i,1} & s_{i,2} & \cdots & s_{i,n} \end{bmatrix} \begin{bmatrix} 1 \\ u_j \\ \vdots \\ u_j^{n-1} \end{bmatrix} = s_{i,1} \cdot 1 + s_{i,2} \cdot u_j + \cdots + s_{i,n} \cdot u_j^{n-1} = p_i(u_j).$$

Since I is the $n \times n$ identity matrix,

$$p_i(u_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases}$$

To obtain the polynomials $p_i(x)$, we compute $M(x) \in F[x]$ of degree n by

$$M(x) = (x - u_1)(x - u_2) \cdots (x - u_n).$$

Zippel calls $M(x)$ the master polynomial. Let $q_i(x) \in F[x]$ of degree $n - 1$ be computed by

$$q_i(x) = \frac{M(x)}{x - u_i} = \prod_{j \neq i} (x - u_j)$$

for all $1 \leq i \leq n$. Each division to obtain q_i costs $3n$ arithmetic operations in F by Corollary 3.3. Then we have

$$\begin{cases} q_i(u_j) = 0 & \text{if } i \neq j \\ q_i(u_j) \neq 0 & \text{otherwise} \end{cases}$$

Since $p_i(u_i) = 1$, we can set $p_i(x)$ as

$$p_i(x) = q_i(u_i)^{-1} \cdot q_i(x)$$

for $1 \leq i \leq n$. Now, we have all the coefficients of $p_i(x)$. Consequently, we obtain U^{-1}. Then we can solve the transposed Vandermonde system of equations $U\mathbf{a} = \mathbf{b}$.

Example 3.24. Assume $F = \mathbb{Z}_{11}$. Let U be a transposed Vandermonde matrix made up of $u_1 = 1$, $u_2 = 2$, and $u_3 = 3$, and $\mathbf{b} = [4, 5, 6]$.

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 4 & 9 \end{bmatrix}}_{U} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix}}_{\mathbf{a}} = \underbrace{\begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}}_{\mathbf{b}}$$

To solve this transposed Vandermonde system of equations, we compute the master polynomial M first.

$$M(x) = (x - 1)(x - 2)(x - 3) \mod 11 = x^3 + 5x^2 + 5.$$

Now, all q_is are obtained by polynomial division.

$$\begin{aligned} q_1(x) &= \frac{M(x)}{x-1} = x^2+6x+6 \\ q_2(x) &= \frac{M(x)}{x-2} = x^2+7x+3 \\ q_3(x) &= \frac{M(x)}{x-3} = x^2+8x+2 \end{aligned}$$

Then we evaluate q_is at $x = u_i$ for all $i = 1, 2, 3$.

$$q_1(1) = 2,\ q_2(2) = 10,\ q_3(3) = 2$$

Computing the inverse of $q_i(u_i)$ gives

$$c_1 = q_1(1)^{-1} = 6,\ c_2 = q_2(2)^{-1} = 10,\ c_3 = q_3(3)^{-1} = 6$$

Now we can obtain p_i by computing $c_i \cdot q_i$ for $i = 1, 2, 3$.

$$\begin{aligned} p_1 &= c_1 \cdot q_1 = 6(x^2+6x+6) = 3+3x+6x^2 \\ p_2 &= c_2 \cdot q_2 = 10(x^2+7x+3) = 8+4x+10x^2 \\ p_3 &= c_3 \cdot q_3 = 6(x^2+8x+2) = 1+4x+6x^2 \end{aligned}$$

Thus,

$$U^{-1} = \begin{bmatrix} 3 & 3 & 6 \\ 8 & 4 & 10 \\ 1 & 4 & 6 \end{bmatrix}$$

The dot product of the coefficients of p_i and $\mathbf{b}$ gives a_0, a_1, and a_2.

$$\begin{aligned} a_0 &= 3\cdot 4 + 3 \cdot 5 + 6 \cdot 6 \mod 11 = 8 \\ a_1 &= 8 \cdot 4 + 4 \cdot 5 + 10 \cdot 6 \mod 11 = 2 \\ a_2 &= 1 \cdot 4 + 4 \cdot 5 + 6 \cdot 6 \mod 11 = 5 \end{aligned}$$

Therefore, $\mathbf{a} = [8, 2, 5]$.

Zippel's transposed Vandermonde solver is presented in Algorithm 24.

Theorem 3.25. *Algorithm 24 does $O(n^2)$ arithmetic operations in F*

Proof. Let $V(n)$ be the number of arithmetic operations in F for Algorithm 24 Zippel's transposed Vandermonde solver. There are n negations to compute $-u_1, -u_2, \ldots, -u_n$. In the for loop in line 2, M is the polynomial of degree i at the $i-$th iteration. Then M is multiplied by a linear polynomial at each iteration, which does i multiplications and i

Algorithm 24 Zippel's transposed Vandermonde solver

```
Input: $n$, $u_1, u_2, \dots, u_n \in F$ which compose the transposed Vandermonde matrix $U$, and
    $\mathbf{b} = [b_1, b_2, \dots, b_n] \in F^n$
Output: $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}] \in F^n$ satisfying $V\mathbf{a} = \mathbf{b}$
$M \leftarrow x - u_1$
for $i$ from 1 to $n-1$ do   // $M = \prod_{i=1}^{n}(x - u_i)$ .............................. $O(n^2)$
    $M \leftarrow M \cdot (x - u_{i+1})$
end for
for $i$ from 1 to $n$ do
    $q \leftarrow M/(x - u_i)$ .............................................................. $O(n)$
    $c \leftarrow q(u_i)$ ................................................................... $O(n)$
    if $c = 0$ then return ERROR "$u_i$'s are not distinct"  end if
    $c \leftarrow c^{-1}$
    $p \leftarrow c \cdot q$      // $p = \sum_{j=0}^{n-1} p_j x^j$ ....................................... $O(n)$
    $P \leftarrow [p_0, p_1, \dots, p_{n-1}]$
    $a_{i-1} \leftarrow$ Compute the dot product of $P \cdot \mathbf{b}$ ................................ $O(n)$
end for
return $[a_0, a_1, \dots, a_{n-1}]$
```

additions. Since this for loop iterates $n-1$ times, it takes $2 + 4 + \cdots + 2(n-1) = n^2 - n$ arithmetic operations to compute M.

After obtaining M, in the second for loop, there is one division of M by $(x - u_i)$ to compute q at each iteration, which does $2n$ arithmetic operations in F by Corollary 3.3. Also, using Horner's method, it takes $n-1$ additions and $n-1$ multiplications to evaluate q at $x = u_i$ since $\deg(q) = n-1$. Only one arithmetic operation is performed to compute the inverse of $q(u_i)$. Moreover, $q(u_i)^{-1} \cdot q(x)$ takes n multiplications. Lastly, the dot product does n multiplications and $n-1$ additions. Hence, each iteration of the second loop costs $2n + 2n - 2 + 1 + n + 2n - 1 = 7n - 2$ arithmetic operations in F. This for loop iterates n times. As a consequence,

$$V(n) = n + n^2 - n + n(7n-2) = 8n^2 - 2n \in O(n^2).$$

□

Lemma 3.26. *Algorithm 24 can be implemented using space for $O(n)$ elements of F*

Proof. Algorithm 24 needs space to store the master polynomial M of degree n, which takes $n+1$ words. Also, in the second for loop, the algorithm requires space to store polynomial q of degree $n-1$. It takes n words to store q. The array for storing q can be reused for the other iterations. Then q can be evaluated at u_i and multiplied by $q(u_i)$ using $O(1)$ elements of F. Thus, Algorithm 24 uses space for $n + 1 + n + O(1) = 2n + 1 + O(1) \in O(n)$ elements of F. □

3.3.2 Fast Transposed Vandermonde Solver

Although Zippel's algorithm is faster than Gaussian elimination, its cost is quadratic in time. This implies that it takes a large amount of time when n is very large. Kaltofen and Yagati presented a faster version of the transposed Vandermonde solver [8]. Kaltofen and Yagati's algorithm costs $O(M(n)\log n)$ arithmetic operations in F, where $M(n)$ is the number of arithmetic operations in F for polynomial multiplication where the sum of the degrees of two polynomials is less than $2n$. However, $O(n\log n)$ space is needed as their algorithm needs to construct the product tree for polynomial multipoint evaluation.

We know that

$$p_i(x) = q_i(u_i)^{-1} \cdot q_i(x) = s_{i,1} + s_{i,2}x + \cdots + s_{i,n}x^{n-1}.$$

from Zippel's algorithm. It follows that

$$\mathbf{a} = U^{-1}\mathbf{b} = \begin{bmatrix} \sum_{j=1}^{n} s_{1,j}b_j \\ \sum_{j=1}^{n} s_{2,j}b_j \\ \vdots \\ \sum_{j=1}^{n} s_{n,j}b_j \end{bmatrix}.$$

We define

$$D = b_n x + b_{n-1}x^2 + \cdots + b_1 x^n \in F[x]. \tag{3.5}$$

Consider

$$p_i \cdot D = (s_{i,1} + s_{i,2}x + \cdots + s_{i,n}x^{n-1})(b_n x + b_{n-1}x^2 + \cdots + b_1 x^n).$$

Observe that the coefficient of x^n in $p_i \cdot D$ is $s_{i,1} \cdot b_1 + s_{i,2} \cdot b_2 + \cdots + s_{i,n} \cdot b_n$ which equals a_{i-1} for $1 \le i \le n$. To compute this, we can rewrite $p_i(x)$ as

$$p_i(x) = q_i(u_i)^{-1} \cdot q_i(x) = q_i(u_i)^{-1}\left(\frac{M(x)}{x-u_i}\right) = \left(\frac{1}{\prod_{j\neq i}(u_i-u_j)}\right)\left(\frac{M(x)}{x-u_i}\right) = \frac{M(x)}{r_i \cdot (x-u_i)}$$

where $r_i = \prod_{j\neq i}(u_i - u_j)$. This implies that

$$M(x) = r_i \cdot (x-u_i) \cdot p_i(x).$$

Suppose we compute

$$H(x) = M \cdot D = h_0 x + h_1 x^2 + \cdots + h_{2n-2}x^{2n-1} + h_{2n-1}x^{2n}.$$

It follows that

$$\frac{H(x)}{x-u_i} = \frac{M \cdot D}{x-u_i} = \frac{r_i \cdot (x-u_i) \cdot p_i(x) \cdot D}{x-u_i} = r_i \cdot p_i(x) \cdot D.$$

Thus, the coefficient of x^n in $\frac{H(x)}{x-u_i}$ is $r_i \cdot a_{i-1}$.

In general, when we compute $\frac{H(x)}{x-z}$, the coefficient of x^i in the quotient is of the form

$$v_i(z) = h_i + h_{i+1}z + \cdots + h_{2n-1}z^{2n-1-i}$$

It follows that the coefficient of x^n in this quotient is

$$v_n(z) = h_n + h_{n+1}z + \cdots + h_{2n-1}z^{n-1}. \tag{3.6}$$

This implies that for $1 \le i \le n$,

$$v_n(u_i) = r_i \cdot a_{i-1}.$$

Now, we need to compute r_i. Recall that $q_i = \frac{M(x)}{x-u_i}$. The derivative of $M(x)$ can be expressed as

$$M'(x) = (x-u_i)q_i'(x) + (x-u_i)'q_i(x) = (x-u_i)q_i'(x) + q_i(x).$$

Then

$$M'(u_i) = (u_i-u_i)q_i'(u_i) + q_i(u_i) = q_i(u_i) = \prod_{j \ne i}(u_i - u_j) = r_i.$$

Hence, r_i can be obtained by evaluating $M'(x)$ at $x = u_i$ for all i. It follows that

$$a_{i-1} = \frac{v_n(u_i)}{r_i} = \frac{v_n(u_i)}{M'(u_i)}$$

for $1 \le i \le n$. Algorithm 25 computes a_{i-1} this way.

We present an example for Algorithm 25.

Example 3.27. Suppose $F = \mathbb{Z}_{17}$. Let U be a transposed Vandermonde matrix made up of $1, 2, 3$, and 4, and $\mathbf{b} = [5, 6, 7, 8]$.

$$\underset{U}{\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \\ 1 & 4 & 9 & 16 \\ 1 & 8 & 10 & 13 \end{bmatrix}} \underset{\mathbf{a}}{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}} = \underset{\mathbf{b}}{\begin{bmatrix} 5 \\ 6 \\ 7 \\ 8 \end{bmatrix}}$$

We can compute the product tree. From this tree, we have

$$M(x) = T_{2,0} = 7 + x + x^2 + 7x^3 + x^4.$$

Then we set D as

$$\begin{aligned} D(x) &= b_4 x + b_3 x^2 + b_2 x^3 + b_1 x^4 \quad \text{(See (3.5))} \\ &= 8x + 7x^2 + 6x^3 + 5x^4. \end{aligned}$$

H is computed by

$$H(x) = M \cdot D = 5x + 6x^2 + 6x^3 + 2x^4 + 3x^6 + 7x^7 + 5x^8.$$

We can obtain the coefficient of x^4 of $\frac{H}{x-z}$ by reading off the coefficients from H as stated in (3.6). It follows that

$$Q(z) = 3z + 7z^2 + 5z^3.$$

By dividing down the product tree, we have

$$v_1 = Q(1) = 15,\ v_2 = Q(2) = 6,\ v_3 = Q(3) = 3,\ v_4 = Q(4) = 2.$$

Now, we compute $M' = 1 + 2x + 4x^2 + 4x^3$. Using dividing down the product tree again, we obtain

$$r_1 = M'(1) = 11,\ r_2 = M'(2) = 2,\ r_3 = M'(3) = 15,\ r_4 = M'(4) = 6.$$

Once we divide v_i by r_i for $1 \le i \le n$, we have

$$\begin{aligned} a_0 &= v_1/r_1 = 15/11 \mod 17 = 6 \\ a_1 &= v_2/r_2 = 6/2 \mod 17 = 3 \\ a_2 &= v_3/r_3 = 3/15 \mod 17 = 7 \\ a_3 &= v_4/r_4 = 2/6 \mod 17 = 6. \end{aligned}$$

Thus, we have the vector $\mathbf{a} = [6, 3, 7, 6]$.

This motivates Algorithm 25 with our BUPT and DDPT algorithms.

Theorem 3.28. *Let $M(n)$ be the number of arithmetic operations in F for multiplying two polynomials where the sum of the degrees of these polynomials is less than $2n$. Algorithm 25 does at most $\frac{53}{6} M(n) \log_2 n + O(n \log n)$ arithmetic operations in F.*

Proof. Let $V(n)$ be the total number of arithmetic operations in F that Algorithm 25 does to solve the $n \times n$ transposed Vandermonde system. Algorithm 25 calls Algorithm 21 BUPT

Algorithm 25 Optimized fast transposed Vandermonde solver(FastTVS)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $u_1, u_2, \ldots, u_n \in F$, which compose the transposed Vandermonde matrix $U$ and $\mathbf{b} = [b_1, b_2, \ldots, b_n] \in F^n$
Output: $\mathbf{a} = [a_0, a_1, \ldots, a_{n-1}] \in F^n$ satisfying $U\mathbf{a} = \mathbf{b}$
 1: $T \leftarrow$ BUPT($n$, $u_1, u_2, \ldots, u_n$)   // $T$ is the modified product tree ............... $B(n)$
 2: $M \leftarrow T_{k,0}$ from $T$   // $M = \prod_{i=1}^{n}(x - u_i)$
 3: $D \leftarrow b_n + b_{n-1}x + \cdots + b_1x^{n-1}$
 4: $H \leftarrow (M \cdot D) \cdot x$ using fast multiplication   // $H = \sum_{i=0}^{2n-1} h_i x^{i+1}$ .............. $M(n)$
 5: $Q \leftarrow \sum_{i=0}^{n-1} h_{n+i}x^i$   // $\sum_{i=0}^{n-1} h_{n+i}z^i$ is the coefficient of $x^n$ in $H/(x - z)$
 6: $q_1, q_2, \ldots, q_n \leftarrow$ DDPT($n$,$Q$,$T$)   // $q_i = Q(u_i)$ ................................ $C(n)$
 7: Differentiate $M$ ............................................................ $O(n)$
 8: $r_1, r_2, \ldots, r_n \leftarrow$ DDPT($n$,$M'$, $T$)   // $r_i = M'(u_i)$ ............................ $C(n)$
 9: for $i$ from 1 to $n$ do ...................................................... $O(n)$
10:     $t \leftarrow r_i^{-1}$
11:     $a_{i-1} \leftarrow t \cdot q_i$
12: end for
13: return $[a_0, a_1, \ldots, a_{n-1}]$
```

of size n once, which performs $B(n)$ arithmetic operations in F. Also, Algorithm 25 calls Algorithm 22 DDPT of size n twice which does $2C(n)$ arithmetic operations in F. To obtain H in line 4, one polynomial multiplication of M of degree n and D of degree $n-1$ is done, which costs $M(n)$. In line 7, computing M' does n multiplications in F. From line 9 to 12, it takes n multiplications and n inverses to compute r_i^{-1} and $q_i \cdot r_i^{-1}$ for $1 \leq i \leq n$. Thus, the total number of arithmetic operations in F for our FastTVS is

$$V(n) \leq B(n) + 2C(n) + M(n) + 3n.$$

Since we use our BUPT algorithm and DDPT algorithm, $B(n) = \frac{1}{2}M(n)\log_2 n + O(n \log n)$ and $C(n) = D(n)\log_2 n$ by Theorem 3.14 and Theorem 3.17. It follows that

$$V(n) = \frac{1}{2}M(n)\log_2 n + O(n \log n) + 2D(n)\log_2 n + M(n) + 3n.$$

With our fast division, $D(n) = \frac{11}{3}M(n) + O(n)$ according to Theorem 3.12. This implies that

$$\begin{aligned} V(n) &= \frac{1}{2}M(n)\log_2 n + O(n \log n) + 2\left(\frac{11}{3}M(n) + O(n)\right)\log_2 n + M(n) + 3n \\ &= \frac{1}{2}M(n)\log_2 n + O(n \log n) + \frac{22}{3}M(n)\log_2 n + O(n \log n) + M(n) + 3n \\ &\leq \frac{53}{6}M(n)\log_2 n + O(n \log n). \end{aligned}$$

□

If we use fast multiplication, $M(n) = 9n \log_2 n + O(n)$. Thus,

$$V(n) \leq \frac{53}{6} (9n \log_2 n + O(n)) \log_2 n + O(n \log n) = \frac{159}{2} n \log_2^2 n + O(n \log n) \in O(n \log^2 n).$$

The following is the table of algorithms presented in Chapter 3.

Operation	Algorithm	Complexity	Ref.
Newton inversion to order n	Algorithm 12 Newton inversion	$3M(n) + O(n)$	[14]
	Algorithm 13 Newton inversion with the middle product	$2M(n) + O(n)$	[6]
	Algorithm 14 NIwithMP	$\frac{5}{3}M(n) + O(n)$	
Polynomial division of degree $2n-1$ by n	Algorithm 11 Classical division	$O(n^2)$	[14]
	Algorithm 15 Fast division	$\frac{11}{3}M(n) + O(n)$	
	Algorithm 16 FastDiv	$\frac{11}{3}M(n) + O(n)$	
Building a product tree with n points	Algorithm 17 Building up a product tree	$M(n) \log_2 n + O(n)$	[2]
	Algorithm 20 Recursive building up a product tree	$\frac{1}{2}M(n) \log_2 n + O(n \log n)$	
	Algorithm 21 BUPT	$\frac{1}{2}M(n) \log_2 n + O(n \log n)$	
Dividing down the product tree with n points	Algorithm 18 Dividing down the product tree	$\frac{11}{3}M(n) \log_2 n + O(n \log n)$	[2]
	Algorithm 22 DDPT	$\frac{11}{3}M(n) \log_2 n + O(n \log n)$	
Polynomial multipoint evaluation with n points	Horner's method	$2n^2 - 2n$	[14]
	Algorithm 19 Fast multipoint evaluation	$\frac{14}{3}M(n) \log_2 n + O(n \log n)$	[2]
	Algorithm 23 FastEval	$\frac{25}{6}M(n) \log_2 n + O(n \log n)$	
$n \times n$ Transposed Vandermonde solver	Algorithm 24 Zippel's transposed Vandermonde solver	$8n^2 - 2n$	[15]
	Algorithm 25 FastTVS	$\frac{53}{6}M(n) \log_2 n + O(n \log n)$	

Table 3.3: The list of algorithms in Chapter 3 and complexity in the number of arithmetic operations in F

Chapter 4

Benchmarks

In Chapter 3, we gave the number of arithmetic operations for every fast algorithm. We want to demonstrate that our optimized fast algorithms are much faster than the classical algorithms in practice. The implementation of the classical algorithms is from Monagan's libraries `gcd4.c` and `VSolve3.c`. We used the prime $p = 3{\cdot}2^{30}{+}1$ for testing polynomials over a field $\mathbb{Z}_p$. For transposed Vandermonde solvers, we also used the prime $p = 116 \cdot 2^{55} + 1$, which is 62-bit. We recorded the timings of both algorithms using `clock()` function of `time.h` library in C. Then we converted the values from `clock()` to milliseconds by dividing by 1000. We implemented all the algorithms discussed in Chapter 3 on the computer with 32 GB of RAM and one AMD FX8350 8-core CPU at 4.2GHz.

4.1 Fast Division

Chapter 3 shows that our optimized fast division (FastDiv) does $O(n \log n)$ arithmetic operations in F. We use `poldiv64s` in `gcd4.c` library for the classical division, which does $O(n^2)$ arithmetic operations in F. During our optimized fast division, we also record the time for our NIwithMP. Table 4.1 shows the time for Monagan's `poldiv64s` and our FastDiv, where n is the degree of divisor polynomial and $2n-1$ is the degree of dividend polynomial. All the dividend and divisor polynomials' coefficients are chosen randomly from $[0, p)$.

The first column labelled "n" represents the degree of the divisor. The second column labelled "Classical division" contains the time it took to execute the classical division algorithm. The third column labelled "ratio" indicates the ratio computed as

$$\text{ratio} = \frac{\text{run time of division algorithm with } n}{\text{run time of division algorithm with } \frac{n}{2}}$$

from the classical division. The fourth column contains the time spent in our Newton inversion with the middle product (NIwithMP) when our FastDiv is executed. Like the second and third columns, the fifth and sixth columns are the time for FastDiv and the ratio for our FastDiv. The following column labelled "speed up" indicates the value

$$\text{speed up} = \frac{\text{Classical division time}}{\text{FastDiv time}}.$$

The last column labelled "Maple" presents the execution time for Maple's division routine `Rem`, which computes remainder for $\mathbb{Z}_p[x]$ [10].

n	Classical division	ratio	NIwithMP	FastDiv	ratio	speed up	Maple
2^6	0.0085	-	-	0.0085	-	1.00	0.1977
2^7	0.0273	3.21	-	0.0270	3.17	1.01	0.3330
2^8	0.0930	3.41	-	0.0920	3.41	1.01	0.6933
2^9	0.3356	3.60	-	0.3355	3.64	1.00	1.6250
2^{10}	1.3240	3.94	0.5310	1.1998	3.57	1.10	3.9531
2^{11}	5.2450	3.96	1.081	2.6130	2.17	2.00	9.4062
2^{12}	20.454	3.90	2.364	5.8370	2.23	3.50	22.062
2^{13}	81.311	3.98	5.015	12.337	2.11	6.59	50.250
2^{14}	321.84	3.96	10.654	26.598	2.16	12.10	115.75
2^{15}	1,293.5	4.02	23.149	56.460	2.12	22.91	275.50
2^{16}	5,237.0	4.05	49.806	122.07	2.16	42.90	646.00

Table 4.1: CPU timings in ms for polynomial divisions over $\mathbb{Z}_p$ of degree $2n-1$ divided by n where $p = 3 \cdot 2^{30} + 1$

Our FastDiv outperforms the classical division algorithm for $n \geq 2^{10} = 1024$. When the degree of the divisor is $2^{16} = 65,536$, our FastDiv executes approximately 43 times faster than the classical algorithm. For $n \leq 512$, our FastDiv algorithm executes the classical division algorithm instead. Consequently, FastDiv increases by a factor of around 3.5 as the degrees of the dividend and divisor are doubled for $n \leq 512$. Moreover, the time for the classical division algorithms increases by a factor approaching 4 as we double the degrees of dividend and divisor. This indicates that the classical division algorithm is quadratic.

4.2 Fast Multipoint Evaluation

We observed that our fast multipoint evaluation (FastEval) costs $O(n \log^2 n)$ in Chapter 2. In `gcd4.c`, `poleval64s` is the classical evaluation algorithm based on Horner's method. To compare the run time, we executed our FastEval and `poleval64s`. We also computed the time for our optimized building up a product tree (BUPT) algorithm and dividing down the product tree (DDPT) algorithm during execution. Table 4.2 presents the time for the classical polynomial evaluation and our optimized version of fast evaluation algorithms at n distinct evaluation points. We set the degree of polynomials being evaluated as $n-1$.

All coefficients of the polynomials and distinct evaluation points are randomly chosen from $[0, p)$.

n	Classical evaluation	ratio	BUPT	DDPT	FastEval	ratio	speed up
2^6	0.0390	-	-	-	0.0380	-	1.02
2^7	0.1460	3.74	0.074	0.112	0.2050	5.39	0.71
2^8	0.5709	3.91	0.104	0.240	0.3850	1.88	1.48
2^9	2.255	3.95	0.320	0.659	0.9900	2.57	2.27
2^{10}	9.011	3.99	0.841	1.987	2.840	2.87	3.17
2^{11}	35.994	3.99	1.934	6.359	8.231	2.90	4.37
2^{12}	142.65	3.96	4.536	17.700	22.357	2.71	6.38
2^{13}	572.67	4.01	10.701	46.824	57.665	2.58	9.93
2^{14}	2,287.8	4.00	25.040	117.22	142.48	2.47	16.05
2^{15}	9,132.0	3.99	57.022	286.26	343.67	2.41	26.57
2^{16}	36,676.9	4.02	129.92	677.44	808.12	2.35	45.38

Table 4.2: CPU timings in ms for polynomial multipoint evaluations over $\mathbb{Z}_p$ of degree $n-1$ at n distinct points where $p = 3 \cdot 2^{30} + 1$

In Table 4.2, the first column is the number of evaluation points n. The second column labelled "Classical evaluation" contains the time for the classical evaluation algorithm. The following column labelled "ratio" contains the value defined as

$$\text{ratio} = \frac{\text{execution time for evaluation algorithm with } n}{\text{execution time for evaluation algorithm with } \frac{n}{2}}.$$

We compute these values from the time for the classical evaluation algorithm. The fourth column labelled "BUPT" represents the time for our BUPT algorithm. Similarly, the fifth column labelled "DDPT" shows the time for our DDPT algorithm. The following column labelled "FastEval" contains the total time to execute our FastEval. Like the third column, the sixth column contains the ratio values defined above for our FastEval algorithm. Lastly, the last column labelled "speed up" contains the values computed by

$$\text{speed up} = \frac{\text{Classical evaluation time}}{\text{FastEval time}}.$$

From Table 4.2, we note that our fast multipoint evaluation is slower than the classical evaluation when $n = 128$. As we set $n = 64$ as a cut-off, our FastEval runs the classical evaluation for $n \leq 64$. However, when $n = 128$, we need to compute the products of linear polynomials, $\prod_{i=0}^{63}(x - u_i)$ and $\prod_{i=64}^{127}(x - u_i)$, where u_is are evaluation points. Since computing these two products is expensive, our FastEval is slower than the classical evaluation for $n = 128$. For $n \geq 256$, our FastEval outperforms the classical evaluation algorithm. In particular, our fast evaluation runs 45 times faster than the classical evaluation when

$n = 2^{16} = 65,536$. As we expected, the classical algorithm grows a factor of 4 as n increases by a factor of 2.

4.3 Fast Transposed Vandermonde Solver

In `VSolve3.c`, Zippel's transposed Vandermonde solver algorithm `VandermondeSolve64s` does $O(n^2)$ arithmetic operations in F for an $n \times n$ transposed Vandermonde system. As we discussed in Chapter 3, our FastTVS does $O(n \log^2 n)$ arithmetic operations in F.

We use two primes: a 32-bit prime $p = 3 \cdot 2^{30} + 1$ and a 62-bit prime $p = 116 \cdot 2^{55} + 1$. Table 4.3 presents the time for transposed Vandermonde solvers over $\mathbb{Z}_p$ with $p = 3 \cdot 2^{30} + 1$. Similarly, Table 4.4 presents the time for transposed Vandermonde solvers over $\mathbb{Z}_p$ with $p = 116 \cdot 2^{55} + 1$. Table 4.3 and Table 4.4 show the time for Zippel's transposed Vandermonde solver (Zippel TVS) and our optimized fast transposed transposed Vandermonde solver (FastTVS) with an $n \times n$ transposed Vandermonde matrix. We also track the time for the BUPT and the two DDPTs during our FastTVS execution.

For each benchmark, we create a polynomial f of degree $n - 1$ where all the coefficients are randomly chosen from $[0, p)$. Using α chosen randomly from $[0, p)$, we evaluate f at α^i for $0 \leq i \leq n - 1$. Then we use $[\alpha^0, \alpha^1, \ldots, \alpha^{n-1}] \in F^n$, which compose $n \times n$ transposed Vandermonde matrix, and $[f(\alpha^0), f(\alpha^1), \ldots, f(\alpha^{n-1})] \in F^n$ as inputs.

In both Table 4.3 and Table 4.4, the first column n is the number of interpolation points. The second column labelled "Zippel TVS" shows the time for Zippel's transposed Vandermonde solver. We define ratio as

$$\text{ratio} = \frac{\text{execution time for TVS algorithm with } n}{\text{execution time for TVS algorithm with } \frac{n}{2}}.$$

The third column labelled "ratio" contains the ratio computed from Zippel TVS. Then the following columns represent the time for subroutines while our FastTVS runs. The column labelled "BUPT" contains the time for our BUPT algorithm. Moreover, the columns labelled "DDPT1" and "DDPT2" show the time for our DDPT algorithm executed twice in our FastTVS. Like the second and third columns, the seventh column labelled "FastTVS" indicates the total time for our FastTVS algorithm and the eighth column shows the ratio as defined above computed from our FastTVS. The following column labelled "speed up" contains the value obtained by

$$\text{speed up} = \frac{\text{Zippel TVS time}}{\text{FastTVS time}}.$$

The last column labelled "Maple" presents the execution time for Monagan's Maple implementation of Algorithm 25 FastTVS, which uses Maple's fast multiplication and division routines for $\mathbb{Z}_p[x]$.

n	Zippel TVS	ratio	BUPT	DDPT1	DDPT2	FastTVS	ratio	speed up	Maple
2^6	0.1270	-	0.020	0.042	0.041	0.132	-	0.96	2.8
2^7	0.4830	3.80	0.047	0.109	0.096	0.309	2.34	1.56	7.4
2^8	1.9249	3.99	0.128	0.243	0.236	0.875	2.83	2.19	15.8
2^9	7.5300	3.91	0.303	0.654	0.653	2.055	2.35	3.66	32.6
2^{10}	30.091	4.00	0.768	1.965	1.979	5.664	2.76	5.31	81.6
2^{11}	119.46	3.97	1.929	6.313	6.397	16.677	2.94	7.16	173.6
2^{12}	476.21	3.99	4.556	17.676	17.719	44.234	2.65	10.76	349.0
2^{13}	1,903.1	4.00	10.692	46.762	46.306	112.81	2.55	16.86	973.0
2^{14}	7,617.6	4.00	24.734	116.68	116.82	277.05	2.46	27.49	2,167
2^{15}	30,629	4.02	58.236	284.83	285.75	670.03	2.42	45.71	4,860
2^{16}	122,162	3.99	130.42	678.51	681.43	1,575.4	2.35	77.54	11,124

Table 4.3: CPU timings in ms for solving $n \times n$ transposed Vandermonde system over $\mathbb{Z}_p$ with $p = 3 \cdot 2^{30} + 1$

n	Zippel TVS	ratio	BUPT	DDPT1	DDPT2	FastTVS	ratio	speed up	Maple
2^6	0.1389	-	0.046	0.049	0.040	0.1990	-	0.69	3.4
2^7	0.4879	3.51	0.052	0.103	0.093	0.3220	1.61	1.51	8.6
2^8	1.9039	3.90	0.120	0.244	0.242	0.7900	2.45	2.41	20.8
2^9	7.4640	3.92	0.325	0.664	0.662	1.9690	2.49	3.79	63.0
2^{10}	30.826	4.12	0.748	1.952	1.952	5.4069	2.74	5.70	113.2
2^{11}	116.84	3.79	1.960	6.234	6.348	15.577	2.88	7.50	270.0
2^{12}	469.64	4.01	4.546	17.596	17.678	42.371	2.71	11.08	608.0
2^{13}	1,868.0	3.97	10.784	45.389	45.714	107.41	2.53	17.39	1,321
2^{14}	7,455.7	3.99	24.692	114.60	114.22	265.43	2.47	28.08	3,025
2^{15}	29,986	4.02	56.525	279.13	277.70	649.44	2.44	46.17	7,190
2^{16}	120,292	4.01	128.54	663.53	661.97	1,500.7	2.31	80.15	16,403

Table 4.4: CPU timings in ms for solving $n \times n$ transposed Vandermonde system over $\mathbb{Z}_p$ with $p = 116 \cdot 2^{55} + 1$

Table 4.3 and Table 4.4 show that our fast transposed Vandermonde solver beats Zippel's transposed Vandermonde solver for $n \geq 2^7 = 128$. When $n = 2^{16} = 65,536$, our FastTVS is 77 times faster than Zippel's transposed Vandermonde solver with $p = 3 \cdot 2^{30} + 1$. Also, our FastTVS is 80 times as fast as Zippel's method with $p = 116 \cdot 2^{55} + 1$. Both tables indicate that the time for Zippel's method increases by a factor of 4 as we double the number of interpolation points since it costs $O(n^2)$.

Chapter 5

Implementation Notes

5.1 Polynomial Representation and Underlying Library

We have coded the fast algorithms in C to speed up our implementation. The C code for the fast algorithms is provided in Appendix A. Our C code supports primes $p < 2^{63}$. We used the prime $p = 3 \cdot 2^{30} + 1$ for testing. We use a dense array of coefficients for polynomial representation, as shown in Figure 5.1.

0	1	2	3	$\cdots$	$d-1$	d
f_0	f_1	f_2	f_3	$\cdots$	f_{d-1}	f_d

Figure 5.1: Polynomial $f(x) = \sum_{i=0}^{d} f_i x^i$ in C where $f_i \in \mathbb{Z}_p$

For arithmetic in $\mathbb{Z}_p$, we use Monagan's classical arithmetic library for $\mathbb{Z}_p[x]$ `gcd4.c`. In `gcd4.c`, we have the following subroutines for arithmetic in $\mathbb{Z}_p$. All routines assume $a, b \in \mathbb{Z}_p$.

```
#define LONG long long int
```

defines `LONG` as a signed 64-bit integer.

```
LONG add64s(LONG a, LONG b, LONG p);
```

returns $a + b \mod p$.

```
LONG sub64s(LONG a, LONG b, LONG p);
```

returns $a - b \mod p$.

```
LONG neg64s(LONG a, LONG p);
```

returns $-a \mod p$.

```
LONG modinv64s(LONG a, LONG p);
```

returns $a^{-1} \mod p$.

```
LONG mul64s(LONG a, LONG b, LONG p);
```

returns $a \cdot b \mod p$.

`LONG powmod64s(LONG a, LONG n, LONG p);`
returns $a^n \mod p$ for $n \geq 0$.

We note that `mul64s(a,b,p)` computes $a \cdot b$ to get a 128-bit signed integer $c = a \cdot b$ first. Then this subroutine divides c by the 64-bit signed integer p. However, the hardware division instruction is very slow. The division instruction is typically 10 to 40 times longer than the multiplication $a \cdot b$ [11]. To speed up multiplication in $\mathbb{Z}_p$, we use Pearce's library `int128g.c`. In this library, we use the following subroutines for multiplication in $\mathbb{Z}_p$.

`#define UINT64 unsigned long long`
defines `UINT64` as an unsigned 64-bit integer.
`typedef struct {UINT64 s; UINT64 v; UINT64 d0; UINT64 d1;} recint;`
defines `recint` as a composite datatype to store the pre-computed inverse of p.
`recint recip1(UINT64 p);`
returns a new `recint` variable including the inverse of p.
`UINT64 mulrec64(UINT64 a, UINT64 b, recint v);`
returns $a \cdot b \bmod p$ where p^{-1} is stored in v.

Let $c = a \cdot b$. Based on Möller and Granlund's idea [11], `mulrec64(a,b,v)` gets the pre-computed integer p^{-1} stored in v and multiplies p^{-1} by c instead of dividing c by p. Let $q = c \cdot p^{-1}$. Then this subroutine computes $c - q \cdot p$ to obtain $a \cdot b \mod p$. While `mul64s` does one multiplication and one division, `mulrec64` does three multiplications, some other bit operations, and a subtraction.

We want to demonstrate that `mulrec64` is faster than `mul64s` in practice. We chose two primes $p = 3 \cdot 2^{30} + 1$ and $p = 2^{62} - 67$. We created an array of one million elements which are chosen from $[0, p)$ at random. Then we compute point-wise multiplication of this array by itself twice, which does two million modular multiplications in total.

Prime p	Computer	`mul64s`	`mulrec64s`
$3 \cdot 2^{30} + 1$	`luke`	21.431	9.737
	`steph`	19.442	5.067
	`maple`	5.743	4.326
$2^{62} - 67$	`luke`	38.096	8.952
	`steph`	53.870	4.658
	`maple`	5.741	4.194

Table 5.1: CPU timings in ms for two million multiplications in $\mathbb{Z}_p$ on the three different computers: `luke`, `steph`, and `maple`

We recorded the time for `mul64s` and `mulrec64` on three different computers. The first computer `luke` has 32 GB of RAM and one AMD FX8350 8-core CPU at 4.2GHz. Another computer `steph` has 16 GB of RAM and Intel Core i5 8500 with 6 cores CPU at 3.4/4.1 GHz. The last computer `maple` has 256 GB of RAM and 2 Intel Gold 6342 with 24 core

CPUs at 2.8/3.5 GHz. Table 5.1 shows that `mulrec64` is faster than `mul64s` in practice. Thus, we use `mulrec64` instead of `mul64s`.

Monagan's `gcd4.c` library also has the classical algorithms for operations on polynomials over $\mathbb{Z}_p$.

```
int poladd64s(LONG *A, LONG *B, LONG *C, int da, int db, LONG p);
```
computes $C = A + B \mod p$ where $da = \deg(A)$ and $db = \deg(B)$ and returns $\deg(C)$.

```
int polsub64s(LONG *A, LONG *B, LONG *C, int da, int db, LONG p);
```
computes $C = A - B \mod p = C$ where $da = \deg(A)$ and $db = \deg(B)$ and returns $\deg(C)$.

```
int poldiff64s(LONG *f, int d, LONG *g, LONG p);
```
computes $f' \mod p$, stores $f' \mod p$ in g where $d = \deg(f)$, and returns $\deg(g)$.

```
int polmul64s(LONG *A, LONG *B, LONG *C, int da, int db, LONG p);
```
computes $C = A \cdot B \mod p$ where $da = \deg(A)$ and $db = \deg(B)$ and returns $\deg(C)$.

```
int poldiv64s(LONG *A, LONG *B, int da, int db, LONG p);
```
computes the quotient Q and the remainder R satisfying $A = BQ + R \mod p$ where $da = \deg(A)$ and $db = \deg(B)$ and returns $\deg(R)$. Q and R are overwritten in A as $A = [\ R \mid Q\ ]$ in this division.

```
LONG poleval64s(LONG *f, int d, LONG x, LONG p);
```
returns $f(x) \mod p$ where $d = \deg(f)$.

```
void polcopy64s(LONG *A, int d, LONG *B);
```
copies A where $d = \deg(A)$ and pastes this into B.

Moreover, we use Zippel's transposed Vandermonde solver algorithm from the library `VSolve3.c`.

```
void VandermondeSolve64s(LONG *m, LONG *y, int n, LONG *a, LONG *M, int shift,
LONG p);
```
solves the transposed Vandermonde system of equations

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ m_0 & m_1 & m_2 & \cdots & m_{n-1} \\ m_0^2 & m_1^2 & m_2^2 & \cdots & m_{n-1}^2 \\ \vdots & \vdots & \vdots & & \vdots \\ m_0^{n-1} & m_1^{n-1} & m_2^{n-1} & \cdots & m_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

where M is an array of size $n + 1$ to store $\prod_{i=0}^{n-1}(x - m_i)$. The algorithm does not allocate any arrays, so it uses $O(n)$ space.

5.2 Fast Multiplication

Let $f, g \in \mathbb{Z}_p[x]$ be polynomials to be multiplied. For our Algorithm 8 FastMul, we use the following subroutines.

The following subroutines are from Monagan's `fftutil8.c` library.

`void MakeW64(LONG n, LONG w, LONG *W, LONG p, recint P);`

computes the powers of w and stores them in W where w is a primitive n-th root of unity in $\mathbb{Z}_p$. Then W will contain $[1, \omega, \omega^2, \dots, \omega^{\frac{n}{2}-1}, 1, \omega^2, \omega^4 \dots, \omega^{\frac{n}{2}-2}, 1, \omega^4, \omega^8, \dots, \omega^{\frac{n}{2}-4}, \dots, 1, 0]$.

`void MakeWinv64(LONG n, LONG *W, LONG p);`

obtains the powers of w^{-1} from the existing W filled with the powers of w where w is a primitive n-th root of unity in $\mathbb{Z}_p$. W will become $[1, \omega^{-1}, \omega^{-2}, \dots, \omega^{-\frac{n}{2}+1}, 1, \omega^{-2}, \omega^{-4}, \dots, \omega^{-\frac{n}{2}+2}, 1, \omega^{-4}, \omega^{-8}, \dots, \omega^{-\frac{n}{2}+4}, \dots, 1, 0]$.

Then we implement the following subroutines.

`void FFT1(LONG *A, int n, LONG *W, LONG p, recint P);`

computes $F_w(A)$ with a primitive n-th root of unity in $\mathbb{Z}_p$. This is our C implementation of Algorithm 5 FFT1.

`void FFT2(LONG *A, int n, LONG *W, LONG p, recint P);`

computes $F_w(A)$ with a primitive n-th root of unity in $\mathbb{Z}_p$. This is our C implementation of Algorithm 6 FFT2.

`int polFFTmul(LONG *A, LONG *B, LONG *C, int dA, int dB, LONG p, LONG alpha);`

computes $C = A \cdot B \mod p$ where $dA = \deg(A)$ and $\deg(B) = dB$ and returns $\deg(C)$. α is the primitive element of $\mathbb{Z}_p$, which is pre-computed in Maple. This is our C implementation of Algorithm 7 Fast multiplication.

`int polFASTmul(LONG *A, LONG *B, LONG *C, int dA, int dB, LONG p, LONG alpha);`

computes $C = A \cdot B \mod p$ where $dA = \deg(A)$ and $\deg(B) = dB$ and returns $\deg(C)$. α is the primitive element of $\mathbb{Z}_p$, which is pre-computed in Maple. This is our C implementation of Algorithm 8 FastMul.

Assume we compute $f \cdot g$ using `polFASTmul`. Let n be the smallest power of 2 greater than $\deg(f) + \deg(g)$. First, we need to compute a primitive n-th root of unity.

Lemma 5.1. *Assume p is a prime and F is a field such that $F = \mathbb{Z}_p$. Suppose $n = 2^k$ for some $k \in \mathbb{N}$. Then a primitive n-th root exists if and only if n divides $p - 1$.*

Proof. Let α be a primitive element in $\mathbb{Z}_p$. Since n divides $p - 1$, we can say that

$$n \cdot q = p - 1 \quad \text{for some } q$$

This implies that

$$\begin{aligned} \alpha^{p-1} \mod p = 1 &\iff \alpha^{q \cdot n} \mod p = 1 \\ &\iff (\alpha^q)^n \mod p = 1 \end{aligned}$$

Hence, α^q is a primitive n-th root of unity in F. □

Using Lemma 5.1, we are able to compute a primitive n-th root of unity in F. In `polFASTmul`, we get a primitive element of F as an input, called α. In our implementation, we pre-compute the primitive element α from Maple using `NumberTheory[PrimitiveRoot]` [10]. Then we compute $(p-1)/n$ and call `powmod64s` to calculate $\alpha^{(p-1)/n} \mod p$. We note that we should choose a prime in the form of $p = c \cdot 2^k + 1$ for some c and large k to use Lemma 5.1.

Next, we need to discuss memory allocation for fast multiplication. `polFASTmul` is our C implementation of Algorithm 8. In `polFASTmul` routine, if the product of the degrees of two polynomials is less than 2^{16}, we use the classical multiplication `polmul64s`. Otherwise, we use the FFT by calling `polFFTmul`.

In Algorithm 7, we need an array A of length n containing the coefficients of the f and padded with zeros. Likewise, an array B of length n is needed, which contains the coefficients of g and is padded with zeros. Also, we need another array W for the powers of a primitive n-th root of unity ω using `MakeW64`. We already discussed that our W is of length n in Chapter 2. After computing $F_\omega(A)$ and $F_\omega(B)$, we overwrite $F_\omega(A) \times F_\omega(B)$ in A where $\times$ is a point-wise multiplication. Then, to compute the inverse FFT, we can overwrite the powers of ω^{-1} in W using `MakeWinv64`. Thus, we create a temporary array T of size $3n$ words for A, B, and W using `malloc`.

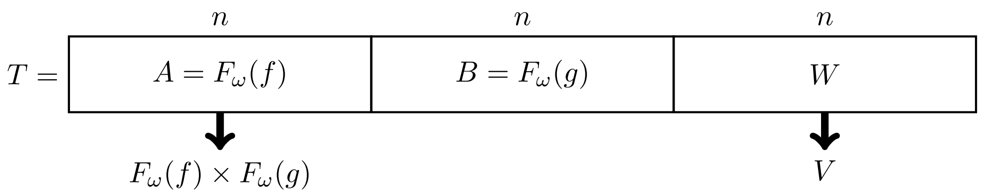

Figure 5.2: The temporary array T of size $3n$ used in `polFFTmul`

5.3 Fast Division

Let $f, g \in \mathbb{Z}_p[x]$ be polynomials where f is a dividend polynomial and g is a divisor polynomial. Assume that $d = \deg(f) - \deg(g) + 1$, which is the degree of the quotient with adding 1. Also, let n be the smallest power of 2 greater than $d - 1$.

We implement the following routines to implement Algorithm 16 FastDiv.

```
void polNIwithMP(LONG *A, LONG *B, LONG *T, int dA, int n, LONG p, LONG alpha);
```

computes A^{-1} to order n approximation using Newton inversion with the middle product and stores this result in B where $dA = \deg(A)$ and T is an extra array for computation. Also, α is the primitive element of $\mathbb{Z}_p$, which is pre-computed in Maple. This is our C implementation of Algorithm 14 NIwithMP.

```
int polFFTdiv (LONG *A, LONG *B, int dA, int dB, LONG p, LONG alpha);
```

computes the quotient Q and the remainder R satisfying $A = B \cdot Q + R$ where $dA = \deg(A)$ and $dB = \deg(B)$. This division stores R and Q in A and returns $\deg(R)$. Also, α is the primitive element of $\mathbb{Z}_p$, which is pre-computed in Maple. This is our C implementation of Algorithm 15 Fast division.

```
int polFASTdiv (LONG *A, LONG *B, int dA, int dB, LONG p, LONG alpha);
```

computes the quotient Q and the remainder R satisfying $A = B \cdot Q + R$ where $dA = \deg(A)$ and $dB = \deg(B)$. This division stores R and Q in A and returns $\deg(R)$. Also, α is the primitive element of $\mathbb{Z}_p$, which is pre-computed in Maple. This is our C implementation of Algorithm 16 FastDiv.

Our C implementation of Algorithm 16 `polFASTdiv` calls the classical division `poldiv64s` for $\deg(g) \cdot (\deg(f) - \deg(g)) \leq 2^{18} = 262,144$. Otherwise, `polFFTdiv` is executed. During `polFFTdiv` execution, `polNIwithMP` is called to compute the inverse of $g^{(rec)}$ to order d approximation where $d - 1$ is the degree of the quotient. While using Newton inversion, we need an array Y for $F_\omega((g^{(rec)})^{-1} \mod x^{\frac{d}{2}})$. Also, we need an array G for $F_\omega(g \mod x^d)$. Both of these arrays are of length n.

Also, for the FFT algorithm, we need the arrays W and V containing the powers of ω and ω^{-1} where $\omega \in \mathbb{Z}_p$ is a primitive n-th root of unity. Since Algorithm 14 uses the FFT and the inverse FFT of size n at least twice, we do not overwrite V in W for reuse, unlike Algorithm 7. The length of W is n, and so is V. It follows that space for $4n$ words is required to store these values for `polNIwithMP` as illustrated in Figure 5.3.

	n	n	n	n
$T =$	$F_\omega((g^{(rec)})^{-1} \mod x^{\frac{d}{2}})$	$F_\omega(g \mod x^d)$	W	V

Figure 5.3: The array T of size $4n$ used in `polNIwithMP`

We store $f^{(rec)}$ in the input array of f by reversing the order of the coefficients. As a result, we do not allocate space for $f^{(rec)}$. In `polFFTdiv`, we need to store $g^{(rec)}$, which requires an array of size d. Also, we should store the output of Newton inversion, $(g^{(rec)})^{-1} \mod x^d$ in an array of size d. After computing Newton inversion, we store $(f^{(rec)} \mod x^d) \cdot ((g^{(rec)})^{-1} \mod x^d)$ in T which is of size at most $2d - 1$ words. Thus, we need a temporary array of size at most $2d + 4n$ words to get the quotient.

After computing the quotient, we reuse the existing temporary array to store the result of $g \cdot q$, where q is the quotient polynomial. Since the degree of $g \cdot q$ is $\deg(f)$, the size of a temporary array to store $g \cdot q$ must be $\deg(f) + 1$. Also, we need space for $\deg(f) + 1$ words to store f to get the remainder since our `polFFTdiv` stores q in the input array containing the dividend polynomial.

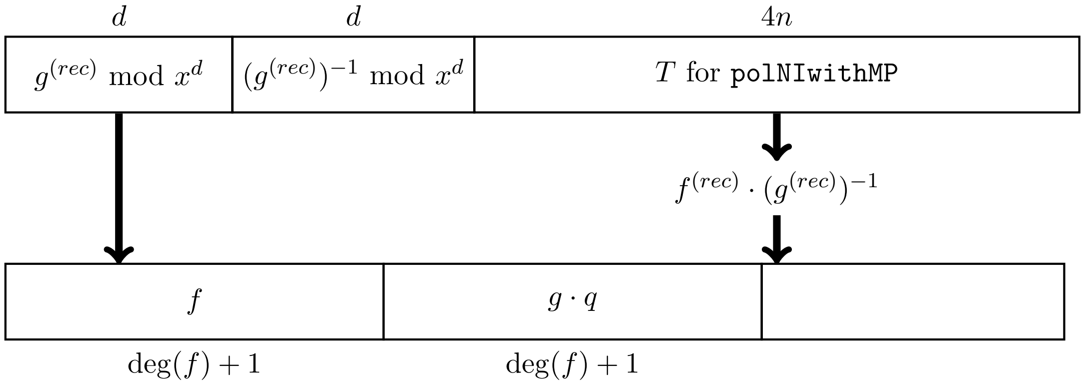

Figure 5.4: The temporary array of size $\max(2d+4n, 2\deg(f)+2)$ used in `polFFTdiv`

Consequently, we allocate the temporary array of size $\max(2d+4n, 2\deg(f)+2)$ words using `malloc`.

5.4 Fast Multipoint Evaluation

Assume we have $n = 2^k$ evaluation points $u_0, u_1, \ldots, u_{n-1} \in \mathbb{Z}_p$ for some $k \in \mathbb{N}$. Let $f \in F[x]$ be a polynomial of degree $n-1$ to be evaluated. The following subroutines are used for our C implementation of FastEval.

`void BUPT(LONG *A, LONG *T, int n, int k, int M, LONG p, LONG alpha, recint P);`
builds a product tree T using $u_0, u_1, \ldots, u_{n-1} \in \mathbb{Z}_p$ stored in A where $n = 2^k$ for some $k \in \mathbb{N}$. M is the distance between every parent node and its first child node in the array T, and α is the primitive element of $\mathbb{Z}_p$ pre-computed in Maple. Also, P is used for `mulrec64`. This is our C implementation of Algorithm 21 BUPT.

`void DDPT(LONG *A, LONG *B, LONG *T, int n, int dA, int k, int M, LONG p, LONG alpha);`
divides A down the product tree T built up with $u_0, u_1, \ldots, u_{n-1} \in \mathbb{Z}_p$ where A is the polynomial to be evaluated and $dA = \deg(A)$. This stores $A(u_0), A(u_1), \ldots, A(u_{n-1}) \in \mathbb{Z}_p$ in B where $n = 2^k$ for some $k \in \mathbb{N}$. M is the distance between a parent node and its first child node in the array T. Also, α is a primitive element of $\mathbb{Z}_p$ pre-computed in Maple. This is our C implementation of Algorithm 22 DDPT.

`void polFASTeval(LONG *A, LONG *B, LONG *U, int n, int dA, int M, LONG p, LONG alpha);`
computes $A(u_0), A(u_1), \ldots, A(u_{n-1}) \in \mathbb{Z}_p$ where A is the polynomial to be evaluated and $dA = \deg(A)$. This routine stores $B_i = A(u_i)$ for $0 \leq i < n$ in B where $u_0, u_1, \ldots u_{n-1} \in \mathbb{Z}_p$ are stored in U. Also, M is the distance between every parent node and its first child in the product tree array, and α is a primitive element of $\mathbb{Z}_p$ computed from Maple. This is our C implementation of Algorithm 23 FastEval.

In our C implementation of Algorithm 23 `polFASTeval`, for $n \leq 64$, the classical evaluation `poleval64s` is called n times. When $n > 64$, `polFASTeval` calls `BUPT` first.

We assume $n = 2^k > 64$ for some $k \in \mathbb{N}$. After two recursive calls in `BUPT`, we need a temporary array C of length $\frac{n}{2}$ to store $(T_{k-1,0} - x^{2^{k-1}}) + (T_{k-1,1} - x^{2^{k-1}})$ before shifting its coefficients.

$$C = \overset{\frac{n}{2}}{\boxed{\quad (T_{k-1,0} - x^{2^{k-1}}) + (T_{k-1,1} - x^{2^{k-1}}) \quad}}$$

Figure 5.5: The temporary array C of size $\frac{n}{2}$ used in `BUPT`

Now, we discuss how `BUPT` returns the modified product tree. As mentioned before, we do not store the leading term of every polynomial in the product tree since it is monic. Since our BUPT is a recursive algorithm, we consider the base case when $n = 64$. Our C implementation of Algorithm 21 `BUPT` computes $T_{6,0} - x^{64}$ and stores this result in the array T.

$$T_{6,0} - x^{64} = \prod_{i=0}^{63}(x - u_i) - x^{64} = a_0 + a_1 x + \cdots + a_{63}x^{63}$$

Depending on the distance M, `BUPT` stores $u_0, u_i, \ldots, u_{n-1}$ in $T[M], T[M+1], \ldots, T[M+n-1]$. Thus, an array of size 128 is required to store $T_{6,0} - x^{64}$ and $u_0, u_1, \ldots u_{n-1}$ when $n = 64$. This is illustrated in Figure 5.6.

	0	1	2	$\cdots$	63	$\cdots$	M	$M+1$	$\cdots$	$M+63$
$T =$	a_0	a_1	a_2	$\cdots$	a_{63}	$\cdots$	u_0	u_1	$\cdots$	u_{63}

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{M}$$

Figure 5.6: The way `BUPT` stores $T_{6,0} - x^{64}$ in the array T when $n = 64$

Now, consider when $n > 64$. Let $n = 2^k$ for some $k \in \mathbb{N}$. Assume our `BUPT` stores every node of polynomials up to $T_{i-1,j} - x^{2^{i-1}}$ in T for $i \geq 7$ and $0 \leq j \leq 2^{k-i+1} - 1$. Then, for $i \geq 7$ and $0 \leq j \leq 2^{k-i} - 1$,

$$T_{i,j} - x^{2^i} = T_{i-1,2j} \cdot T_{i-1,2j+1} - x^{2^i}$$

Let $A = T_{i-1,2j} - x^{2^{i-1}}$ and $B = T_{i-1,2j+1} - x^{2^{i-1}}$. Then

$$T_{i,j} - x^{2^i} = (A + x^{2^{i-1}})(B + x^{2^{i-1}}) - x^{2^i} = A \cdot B + (A + B) \cdot x^{2^{i-1}}$$

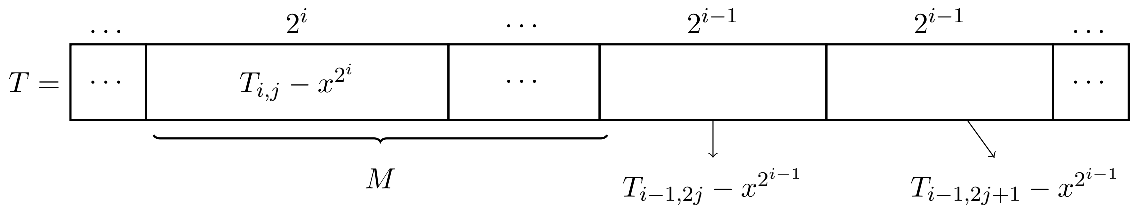

Figure 5.7: The way BUPT stores $T_{i,j} - x^{2^i}$in the array T when $n > 64$

Thus, we can compute $T_{i,j} - x^{2^i}$ without recovering the leading term of $T_{i-1,2j}$ and $T_{i-1,2j+1}$. After computing $T_{i,j} - x^{2^i}$, this polynomial is stored in T, as described in Figure 5.7.

Likewise, our BUPT repeats this procedure until the root $T_{k,0} - x^{2^k}$ is computed. In the end, BUPT stores the modified product tree in T, as shown in Figure 5.8.

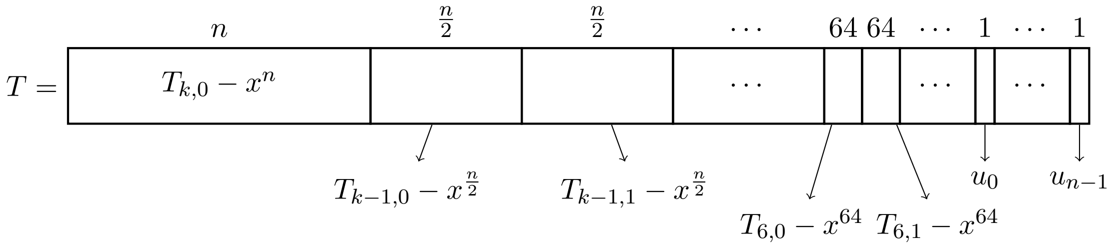

Figure 5.8: The array T of the modified product tree returned from BUPT

Let $S(n)$ be the space required to represent the modified product tree in a one-dimensional array T with n evaluation points. For the base case, $S(64) = 128$ when $n = 64$.

For $n > 64$, we need to store two trees rooted at $T_{i-1,2j} - x^{2^{i-1}}$ and $T_{i-1,2j+1} - x^{2^{i-1}}$. Also, we need the space to store $T_{i,j} - x^{2^i}$ for $0 \leq i \leq 2^{k-i} - 1$, each of which is of size 2^i. Since there are 2^{k-i} many such polynomials, we need space for $2^i \cdot 2^{k-i} = 2^k = n$ words. This implies that $S(n) = 2S(\frac{n}{2}) + n$. Thus, we have

$$\begin{cases} S(64) = 128 \\ S(n) = 2S(\frac{n}{2}) + n \ \text{ for } n > 64 \end{cases}$$

It follows that

$$\begin{aligned} S(n) &= n + n + \cdots + n + 128 \cdot 2^{k-6} \\ &= (k-6)n + 2^{k+1} \\ &= (k-6)n + 2n \\ &= (k-4)n \end{aligned}$$

Therefore, with n evaluation points, we need to allocate a temporary array T of size $(k-4)n$ for the modified product tree using `malloc` in `polFASTeval`.

Additionally, let $f \in F[x]$ be the polynomial to be evaluated. `DDPT` divides f by $T_{k-1,0}$ and $T_{k-1,1}$. Each child polynomial of the root is of degree $\frac{n}{2}$. Since every child polynomial is stored without its leading term, we need to recover the leading term. This implies we need an array of size $\frac{n}{2}+1$. Also, after applying `polFASTdiv`, the dividend polynomial is overwritten with the remainder and the quotient. We should store the dividend in another array for the second division. Thus, `DDPT` requires a temporary array C of size $(\frac{n}{2}+1)+\deg(f)+1 = \frac{n}{2}+\deg(f)+2$.

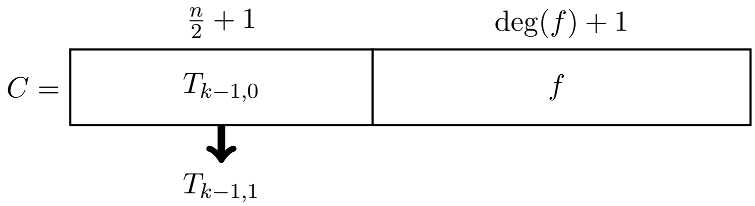

Figure 5.9: The temporary array C of size $\frac{n}{2}+\deg(f)+2$ used in `DDPT`

5.5 Fast Transposed Vandermonde Solver

Let $n = 2^k$ for some $k \in \mathbb{N}$. Suppose the size of a transposed Vandermonde matrix is $n \times n$. We assume the transposed Vandermonde system we want to solve is $U\mathbf{a} = \mathbf{b}$

`void polFASTTVS(LONG *V, LONG *U, LONG *A, int n, int k, LONG p, LONG alpha);`
solves the transposed Vandermonde system of equations

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ u_1 & u_2 & u_3 & \cdots & u_n \\ u_1^2 & u_2^2 & u_3^2 & \dots & u_n^2 \\ \vdots & \vdots & \vdots & & \vdots \\ u_1^{n-1} & u_2^{n-1} & u_3^{n-1} & \cdots & u_n^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_n \end{bmatrix}$$

where $U = [u_1, u_2, \dots, u_n]$ and $V = [v_1, v_2, \dots, v_n]$ and stores the result in A. $n = 2^k$ for some $k \in \mathbb{N}$. Also, α is a pre-computed primitive element of $\mathbb{Z}_p$ from Maple, where p is a prime. This is our C implementation of Algorithm 25 FastTVS.

Like `polFASTeval`, `polFASTTVS` needs an array of size $(k-4)n$ for the modified product tree. Also, this algorithm needs to store the root polynomial M of the modified product tree, which takes $n+1$ words. To store $D = b_n + b_{n-1}x + \cdots + b_1 x^{n-1}$, it needs space for n words. Then $H = (M \cdot D) \cdot x$ should be stored in an array of size $2n+1$. Since Algorithm 25 does not use M,D, and H again, we can reuse the arrays for these polynomials. `polFASTTVS` stores Q in the array for D and $Q(u_1), Q(u_2), \ldots, Q(u_n)$ in the array for H. Moreover, M' is stored in the array for M and $M'(u_1), M'(u_2), \ldots, M'(u_n)$ are stored in the rest of array for H. Therefore, we need to allocate a temporary array C of size $(k-4)n+4n+2 = kn+2$ for `polFASTTVS`.

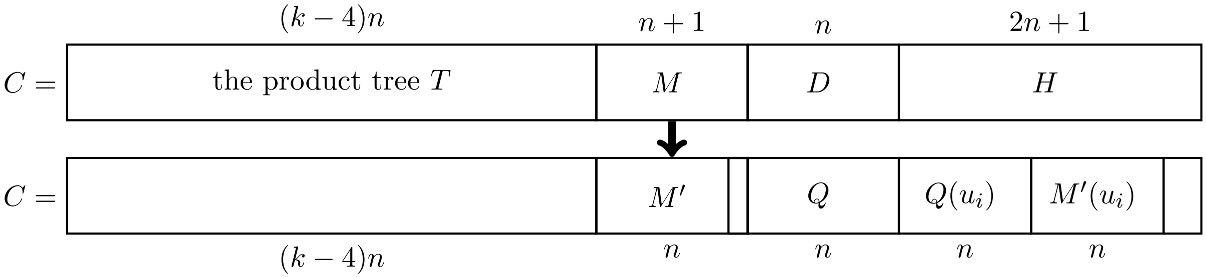

Figure 5.10: The temporary array C of size $kn+2$ used in `polFASTTVS`

Bibliography

[1] Michael Ben-Or and Prasoon Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of the twentieth annual ACM symposium on theory of computing*, pages 301–309. ACM, 1988.

[2] Allan Borodin and Ian Munro. Evaluating polynomials at many points. *Information Processing Letters*, 1(2):66–68, 1971.

[3] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, third edition.* Computer science. MIT Press, 2009.

[5] Roderick Gow. Cauchy's matrix, the Vandermonde matrix and polynomial interpolation. *Bulletin of the Irish Mathematical Society*, 28:45–52, 1992.

[6] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann. The middle product algorithm I. Speeding up the division and square root of power series. *Applicable algebra in engineering, communication and computing*, 14(6):415–438, 2004.

[7] Jiaxiong Hu and Michael Monagan. A fast parallel sparse polynomial gcd algorithm. *Journal of Symbolic Computation*, 105:28–63, 2021.

[8] Erich Kaltofen and Lakshman Yagati. Improved sparse multivariate polynomial interpolation algorithms. In *ISSAC*, Lecture Notes in Computer Science, pages 467–474. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989.

[9] Marshall Law and Michael Monagan. A parallel implementation for polynomial multiplication modulo a prime. In *Proceedings of the 2015 International Workshop on parallel symbolic computation*, pages 78–86. ACM, 2015.

[10] Michael Monagan and Keith Geddes et al. *Maple 8 Introductory Programming Guide.* Number v. 2 in Maple 8 / Waterloo Maple Inc. Waterloo Maple, 2002.

[11] Niels Möller and Torbjörn Granlund. Improved division by invariant integers. *IEEE transactions on computers*, 60(2):165–175, 2011.

[12] John G Proakis. *Digital signal processing : principles, algorithms, and applications / John G. Proakis, Dimitris G. Manolakis.* Prentice Hall, 3rd ed. edition, 1996.

[13] Douglas Robert Stinson and Maura Paterson. *Cryptography : theory and practice.* Textbooks in Mathematics. Chapman and Hall/CRC, Boca Raton, 4th ed. edition, 2018.

[14] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra.* Cambridge University Press, United States, 2013.

[15] Richard Zippel. Interpolating polynomials from their values. *Journal of symbolic computation*, 9(3):375–403, 1990.

Appendix A

Code

```
void polNIwithMP(LONG *A, LONG *B, LONG *T, int dA, int n, LONG p, LONG alpha){
     LONG *B1,*W,*T1,*Mid, *Winv, invN;
     int m,i,N,N2;
     LONG omega;
     recint P;
     //P is initialized to use mulrec64
     P = recip1(p);
     //Base case
     if (n == 1){
          B[0] = modinv64s(A[0],p);
          return;
     }
     //m = ceiling of n/2
     m = n/2;
     if (n & 1) m = m+1;
     //N is the smallest power of 2 greater than or equal to n
     N = 1;
     while (N < n) N = N«1;
     //Recursive newton inversion call
     polNIwithMP(A,B,T,dA,m,p,alpha);
     //Copy the polynomial T = A mod x^n
     for (i = 0;i < n;i++) T[i] = A[i];
     if (n < dA) for(i = n;i < N;i++) T[i] = 0;
     if (dA < N) for (i = dA+1;i < N;i++) T[i] =0;
     //omega is the N-th root of unity
     W = T+2*N;
     omega = powmod64s(alpha,(p-1)/N,p);
     MakeW64(N,omega,W,p,P);
     //FFT multiplication to obtain middle product terms
     T1 = T+N;
     polcopy64s(B,m-1,T1);
     for (i = m;i < N;i++) T1[i]= 0;
     FFT2(T,N,W,p,P);
     FFT2(T1,N,W,p,P);
     for(i = 0;i < N;i++) T[i] = mulrec64(T[i],T1[i],P);
     Winv = W+N;
     for (i = 0;i < N;i++) Winv[i]=W[i];
     MakeWinv64(N,Winv,p);
     FFT1(T,N,Winv,p,P);
     invN = modinv64s(N,p);
     for (i = 0;i < N;i++) T[i] = mulrec64(invN,T[i],P);
     //Extract the middle product terms from above.
     for (i = 0;i < n-m;i++) T[i] = neg64s(T[i+m],p);
     for (i = n-m;i < N;i++) T[i] = 0;
     //Another FFT multiplication
     FFT2(T,N,W,p,P);
     for (i = 0;i < N;i++) T[i] = mulrec64(T[i],T1[i],P);
     FFT1(T,N,Winv,p,P);
     for (i = 0;i < N;i++) T[i] = mulrec64(invN,T[i],P);
     //Add the middle product terms to the output of recursive call
     for (i = 0;i < n-m;i++) B[i+m] = add64s(T[i],B[i+m],p);
      return;
}


int polFFTdiv(LONG *A, LONG *B, int dA, int dB, LONG p, LONG alpha){
     LONG *Ar, *Br, *Brinv, *T, *Q, *R, *Qu, *M;
     int i, s, dQ, dR, N, dM, m, len, min, dm, da;
     LONG temp;
     //B = 0
     if (dB < 0) {
          printf("division by zero \n");
          exit(1);
     }
     //The degree of A is less than the degree of B
     if (dA < dB) return dA;
     //dQ is the degree of quotient
     dQ = dA-dB;
     s = dQ+1;
     //Br is the array to save all temporary value in this algorithm
     N = 1;
     while (N < s) N = N « 1;
     if (4*N +2*s < 2*dA+2) len = 2*dA+2;
     else len = 4*N + 2*s;
     Br = array(len);
     //Br is the reciprocal of B mod x^s
     if (dB > dQ) dm = dQ;
     else dm = dB;
     for (i = 0;i <= dm;i++) Br[i] = B[dB-i];
```

```
    if (dB < dQ) for (i = dB+1;i <= dQ; i++) Br[i] = 0;
    //Compute Br^-1 mod x^s
    Brinv = Br+s;
    T = Brinv+s;
    for (i = 0;i < s;i++) Brinv[i] = 0;
    polNIwithMP(Br,Brinv,T,dB,s,p,alpha);
    //Compute the reciprocal of A
    da = dA/2;
    for (i = 0;i <= da;i++) {
        temp = A[i];
        A[i] = A[dA-i];
        A[dA-i] = temp;
    };
    //Compute the reciprocal of A x Brinv mod x^s
    for (i = 0;i <= 2*dQ; i++ ) T[i] = 0;
    polFASTmul(A,Brinv,T,dQ,dQ,p,alpha);
    //Copy A in the temporary array and store Q mod x^s in A
    for (i = 0;i <= dA;i++ ) Br[i] = A[dA-i];
    Q = A + dB;
    for (i = 0;i <= dQ;i++) Q[i] = T[dQ-i];
    //Store B x Q in M
    M = Br+dA+1;
    polFASTmul(B,Q,M,dB,dQ,p,alpha);
    dM = dQ+dB;
    //Compute A - (B x Q) to get remainder
    dR = polsub64s(Br,M,Br,dA,dM,p);
    //Store the remainder in A
    R = A;
    for (i = 0;i <= dR;i++) R[i] = Br[i];
    free(Ar);
    return dR;
}


int polFASTdiv (LONG *A, LONG *B, int dA, int dB, LONG p, LONG alpha){
    int dR, i;
    if (dB*(dA-dB) <= 262144) {
        dR = poldiv64s(A,B,dA,dB,p);
        return dR;
    }
    dR = polFFTdiv(A,B,dA,dB,p,alpha);
    return dR;
}
```

```
void BUPT(LONG *A, LONG *T, int n, int k, int M, LONG p, LONG alpha, recint P){
     int kn, h1, n1, i, l, t, N, w, q, j;
     LONG *T1, *T2, *C, *A1, *A2, *W, *C1, *C2;
      LONG u;
     //Base case
     if (n <= 64){
          T[0] = neg64s(A[0],p);
          for (i = 1;i < n;i++){
               u = neg64s(A[i],p);
               for (j = i-1;j >= 0;j-) T[j+1] = T[j];
               T[0] = 0;
               for (j = 0;j < i;j++) T[j] = add64s(T[j],mulrec64(T[j+1],u,P),p);
               T[i] = add64s(T[i],u,p);
          }
          for (i = 0;i < n;i++) T[i+M] = A[i];
          return;
     }
     //Set N= 2n and n1 = n/2
     N = 2*n;
     n1 = n»1;
     //Set pointers for recursive call
     T1 = T+M;
     T2 = T1+n1;
     A1 = A;
     A2 = A+n1;
     //Recursive calls for the first half and the second half
     BUPT(A1,T1,n1,k-1,M,p,alpha,P);
     BUPT(A2,T2,n1,k-1,M,p,alpha,P);
     //C is an temporary array for saving the product of two polynomials
     C = array(n1);
     //if n=2, multiplication and addition happen on integers.
     //otherwise, use polmul64s and poladd64s for product and sum of two polynomials
     if(n == 2){
          T[0] = mulrec64(T1[0],T2[0],P);
          C[0] = add64s(T1[0],T2[0],p);
     }else{
          polFASTmul(T1,T2,T,n1-1,n1-1,p,alpha);
          poladd64s(T1,T2,C,n1-1,n1-1,p);
     }
     //Add shifted values of C to T
     for (l = 0;l < n1;l++) T[l+n1] = add64s(T[l+n1],C[l],p);
     free(C);
     return;
}
```

```
void DDPT(LONG *A, LONG *B, LONG *T, int n, int dA, int k, int M, LONG p, LONG
alpha){
     int i, d, m, j, l, doublen, drL, drR, x, kn, km;
     LONG *C, *T1, *T2, *R, *cA, *TL, *TR, *pL, *pR, *AL, *AR, *BL, *BR;
     //Base case
     if (n <= 64){
          for (i = 0;i < n;i++) B[i] = poleval64s(A,dA,T[i+M],p);
          return;
     }
     //Set m,kn, and km
     m = n»1;
     kn = k*n;
     km = kn»1;
     //Set the pointers for polynomial division
     TL = T+M;
     TR = TL+m;
     doublen = n*2;
     C = array(m+dA+2);
     cA = C+m+1;
     for (i = 0;i <= dA;i++) cA[i] = A[i];
     //Obtain the remainder of A divided by the root of the left subtree
     for (i = 0;i < m;i++) C[i] = TL[i];
     C[m] = 1;
     drL = polFASTdiv(A,C,dA,m,p,alpha);
     //Obtain the remainder of A divided by the root of the right subtree
     for (i = 0;i < m;i++) C[i] = TR[i];
     C[m] = 1;
     drR = polFASTdiv(cA,C,dA,m,p,alpha);
     for (i = 0;i <= drR;i++) A[i+m] = cA[i];
     free(C);
     //Recursive calls;
     AL = A;
     AR = A+m;
     BL = B;
     BR = B+m ;
     DDPT(AL,BL,TL,m,drR,k-1,M,p,alpha);
     DDPT(AR,BR,TR,m,drL,k-1,M,p,alpha);
     return;
}

void polFASTeval(LONG *A, LONG *B, LONG *U, int n, int dA, int M, LONG p, LONG
alpha){
     LONG *T;
     int k, m, i, len;
     float t1, t2;
     recint P;
     //For n <= 64, use poleval64s
     if (n <= 64){
          for (i = 0;i < n;i++) B[i] = poleval64s(A,dA,U[i],p);
          return;
     }
     //Compute k = log_2(n)
     k = 0;
     m = n;
     while (m != 1){
          k = k+1;
          m = m»1;
     }
     //Construct a temporary array T of length len
     len = (k-4)*n;
     T = array(len);
     P = recip1(p);
     //Call BUPT to construct a product tree
     BUPT(U,T,n,k,M,p,alpha,P);
     //Call DDPT to get evaluations
     DDPT(A,B,T,n,dA,k,M,p,alpha);
     free(T);
     return;
}


void polFASTTVS(LONG *V, LONG *U, LONG *A, int n, int k, LONG p, LONG alpha){
     int kn, quadn, i, x, d, N, tlen, tlen2;
     LONG *T, *R, *D, *H, *Q, *r, *s, *E1, *E2, *Rd;
     LONG M, t;
     recint P;
     P = recip1(p);
     //Create a product tree using U, of which
     //entries make up the transposed Vandermonde matrix and store in T
     N = n«1;
     if (n <= 64) tlen = 2*n;
     else tlen = (k-4)*n;
     T = array(tlen + 2*N + 2);
     BUPT(U,T,n,k,n,p,alpha,P);
     //Set R is the root of the product tree
     R = T + tlen;
     for (i = 0;i < n;i++) R[i] = T[i];
     R[n] = 1;
     //D is the polynomial with coefficients from V
     D = R+n+1;
     for (i = 0;i < n;i++) D[n-1-i] = V[i];
     //Compute H = (M times D) times x
     H = D+n;
     polFASTmul(D,R,H,n-1,n,p,alpha);
     for (i = 0;i < N+1;i++) H[N+1-i] = H[N-i];
     H[0] = 0;
     //Read off the coefficients of H to get Q
     Q = D;
     for (i = 0;i < n;i++) Q[i] = H[n+i+1];
     //Call DDTP to evaluate Q at U[i]
     E1 = H;
     DDPT(Q,E1,T,n,n-1,k,n,p,alpha);
     //Call DDPT to evaluate R’
     E2 = E1 + n;
     Rd = R;
     d = poldiff64s(R,n,Rd,p);
     DDPT(Rd,E2,T,n,n-1,k,n,p,alpha);
     //Obtain the coefficient vector
     for (i = 0;i < n;i++) A[i] = mulrec64(E1[i],modinv64s(E2[i],p),P);
     free(T);
     return;
}
```