A Maple implementation of FFT-based algorithms for polynomial multipoint evaluation, interpolation, and solving transposed Vandermonde systems

by

Kimberly Connolly

B.Sc., Alcorn State University, 2017

MSc. Project Submitted in Partial Fulfillment of the
Requirements for the Degree of
Masters of Science

in the
Department of Mathematics
Faculty of Science

SIMON FRASER UNIVERSITY
Summer 2020

Approval

Name:	**Kimberly Connolly**
Degree:	**Masters of Science (Mathematics)**
Title:	**A Maple implementation of FFT-based algorithms for polynomial multipoint evaluation, interpolation, and solving transposed Vandermonde systems**
Examining Committee:	**Michael Monagan** Supervisor Professor **Tom Archibald** Co-Supervisor Dept. Chair & Professor
Date Defended:	**August 14, 2020**

Abstract

Classically, algorithms for polynomial multipoint evaluation, interpolation and solving transposed Vandermonde systems over a field have a quadratic running time. More efficient algorithms have been discovered, which are based on the Fast Fourier Transform, but these algorithms are described separately in different literature. To aid in understanding and help with clarity, we will consolidate and examine these fast algorithms all in one place. We will present the algorithms, analyze their complexity, and implement them in Maple. All three algorithms require $O(n \log^2 n)$ arithmetic operations in the field and this improvement on their classical running times allows multipoint evaluation, interpolation and Vandermonde systems to be used in other subquadratic algorithms.

Keywords: Computer Algebra, Subproduct Tree, Vandermonde Systems, Interpolation, Multipoint Evaluation, Fast Fourier Transform, FFT-based Algorithms, Bluestein's Algorithm

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Let F be a field and suppose that we have a polynomial $f(x) = a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1} \in F[x]$. Polynomial evaluation is the process of simplifying a polynomial such as $f(x)$ down to a single numerical value by substituting a given point for the variable. Multi-point evaluation evaluates the same polynomial at a number of arbitrary points. Polynomial interpolation is the reconstruction of the polynomial $f(x)$ from its values $v_0, ..., v_{n-1}$ at n distinct points $u_0, ..., u_{n-1}$, where $f(u_i) = v_i$ for $0 \leq i \leq n-1$. The requirement $f(u_i) = v_i$ yields the following linear system of equations

$$\begin{aligned}
f(u_0) &= a_0 + a_1u_0 + a_2u_0^2 + ... + a_{n-1}u_0^{n-1} = v_0 \\
f(u_1) &= a_0 + a_1u_1 + a_2u_1^2 + ... + a_{n-1}u_1^{n-1} = v_1 \\
f(u_2) &= a_0 + a_1u_2 + a_2u_2^2 + ... + a_{n-1}u_2^{n-1} = v_2 \\
&\vdots \\
f(u_{n-1}) &= a_0 + a_1u_{n-1} + a_2u_{n-1}^2 + ... + a_{n-1}u_{n-1}^{n-1} = v_{n-1}.
\end{aligned}$$

In matrix form, this looks like

$$\underbrace{\begin{bmatrix} 1 & u_0 & u_0^2 & \cdots & u_0^{n-1} \\ 1 & u_1 & u_1^2 & \cdots & u_1^{n-1} \\ 1 & u_2 & u_2^2 & \cdots & u_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & u_{n-1} & u_{n-1}^2 & \cdots & u_{n-1}^{n-1} \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-1} \end{bmatrix}}_{v}.$$

The matrix V is called the Vandermonde matrix of order n, and $Va = v$ is known as a Vandermonde linear system of equations. Thus, solving $Va = v$ for the unknown coefficient vector a is equivalent to interpolating the polynomial $f(x)$ from its values at n points.

Now, in contrast, a transposed Vandermonde system of equations, $V^T b = v$, has the form

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ u_0 & u_1 & u_2 & \cdots & u_{n-1} \\ u_0^2 & u_1^2 & u_2^2 & \cdots & u_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_0^{n-1} & u_1^{n-1} & u_2^{n-1} & \cdots & u_{n-1}^{n-1} \end{bmatrix}}_{V^T} \underbrace{\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}}_{b} = \underbrace{\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-1} \end{bmatrix}}_{v}.$$

The need to solve transposed Vandermonde systems arise in sparse interpolation algorithms, such as in the Ben-Or and Tiwari algorithm [1].

Multipoint evaluation, interpolation and Vandermonde systems are helpful in many algorithms and so their efficiency is important. Subquadratic algorithms can be used as subroutines in other subquadratic algorithms, without increasing the running time. The significance of having all subroutines within an algorithm be subquadratic cannot be overstated. For a quadratic time algorithm, with input of size n, if we double n then the time it takes for a computer to calculate the answer increases by a factor of four. FFT-based subquadratic algorithms, on the other hand, increase by a factor closer to two. This means that, for large n, the same computation will take significantly more time for quadratic algorithms. Thus, studying FFT-based subquadratic algorithms for multipoint evaluation, interpolation, and Vandermonde systems is clearly a worthwhile endeavor to undertake.

Classically, multipoint evaluation, interpolation and Vandermonde systems all have quadratic time algorithms. Horner's method may be used n times to evaluate a polynomial at n points and this costs $O(n^2)$ arithmetic operations in F. Interpolation has traditionally been performed using Lagrange interpolation or Newton interpolation which both do $O(n^2)$ operations. The system of linear equations $V^T b = v$ can be solved naively in cubic time and quadratic space using Gaussian elimination. Zippel showed in [16] how to solve $V^T b = v$ in quadratic time and linear space.

This project will analyze three subquadratic algorithms, one for polynomial multipoint evaluation, another for polynomial interpolation and the last for solving transposed Vandermonde systems, denoted from here on as FastEval, FastInterp and FastVandermonde. These algorithms have been detailed separately in previous literature. FastEval and FastInterp are based on work by Lipson [13], Fiduccia [6], Horowitz [10], Moenck and Borodin [15], and Borodin and Moenck [3], but are described in detail in [7]. FastVandermonde is outlined in [11]. As these three algorithms use the same background tools, it would be beneficial to have all the information in one place, which will hopefully help with clarity and understanding. In this paper, we will show that each of the three algorithms require at most $O(n \log^2 n)$ arithmetic operations in F and that they can be implemented in Maple efficiently.

For FastEval, we require algorithms for fast multiplication and fast division in $F[x]$. For FastInterp and FastVandermonde, we need to use FastEval. Classical implementations for multiplication and division in $F[x]$ do $O(n^2)$ arithmetic operations in F, but we will discuss faster methods later on.

The subsequent chapters of this project are organized as follows. Chapter Two will explore in-depth the Fast Fourier Transform and fast polynomial multiplication as well as Bluestein's algorithm. The third chapter will examine fast polynomial division using Newton iteration. Chapter Four will give an overview of how FastEval works, present a small example, include timings of the algorithm and analyze its complexity. The fifth and sixth chapters are formatted in a similar manner as Chapter Four but instead examine the algorithms FastInterp and FastVandermonde, respectively. The final chapter consists of a brief conclusion.

Before we proceed, throughout this paper we use the following notation and make the following assumptions.

- When we write $\log n$, we are always referring to the binary logarithm, $\log_2 n$.
- In all pseudo code provided, $f \times g$ refers to a fast multiplication of the polynomials f and g, whereas $f \cdot g$ indicates integer multiplication.
- All algorithms presented in this project can work over any field that contains a primitive nth root of unity with $n = 2^k$. Our implementation was done in a prime field $\mathbb{F}_p$ with a Fourier prime p of the form $p = s2^k + 1$ with large k.

Chapter 2

The Fast Fourier Transform

Suppose we have a polynomial $f(x) = a_0 + a_1x + ... + a_{n-2}x^{n-2}a_{n-1}x^{n-1} \in F[x]$, and we wish to evaluate this polynomial at $x = \alpha \in F$. A naive way to do this would be to let $r = a_0$ and $t = \alpha$. Then for $1 \leq i \leq n-1$ do $r = a_i \cdot t + r$ and $t = t \cdot \alpha$. This technique requires $2(n-1)$ multiplications and $n-1$ additions in F.

A better way to evaluate $f(x)$ is to use Horner's method. We can rearrange $f(x)$ as follows

$$f(x) = a_0 + x(a_1 + x(a_2 + ... + x(a_{n-2} + x(a_{n-1}))...)).$$

This grouping is called the Horner form of f. For instance, if $f(x) = 1 + 2x + 3x^2$, then its Horner form is $f(x) = 1 + x(2 + x(3))$.

To begin, let $r = f_{n-1}$. Then, for $i = n-2$ down to 0, compute $r = r \cdot \alpha + f_i$. This method requires $n-1$ multiplications and $n-1$ additions, so we have saved half of the multiplications.

We've just seen that the cost of evaluating a polynomial of degree $n-1$ using Horner's method at one point is $O(n)$ operations in F. Thus, if we want to evaluate the polynomial at n points, $u_0, ..., u_{n-1}$, we will need $n \cdot O(n) = O(n^2)$ operations in F. This can be improved to $O(n \log n)$ operations if we evaluate f at n special points, specifically the n powers of a primitive nth root of unity.

An element w of a field F is an nth root of unity if $w^n = 1$, and w is a primitive nth root of unity if $w^n = 1$ and $w^i \neq 1$ for $1 \leq i \leq n-1$. For example, if $p = 97$, then $w = 22$ is a primitive 4th root of unity in $\mathbb{F}_p$ since $22^4 \bmod 97 = 1$ and $[w^1, w^2, w^3] = [22, 96, 75]$.

Given the polynomial f and w, which is a primitive nth root of unity with $n = 2^k$, then the Discrete Fourier Transform (DFT) evaluates f at the n powers of w and is defined by the vector

$$[A_k = f(w^k) = \sum_{i=0}^{n-1} a_i w^{ik} \colon 0 \leq k \leq n-1] \in F^n.$$

A direct calculation of the DFT costs $O(n^2)$ arithmetic operations in F, using Horner's method.

The Fast Fourier Transform (FFT) is an algorithm that can compute the DFT in $O(n \log n)$ arithmetic operations in F. Many FFT algorithms have been discovered, however, the most well-known one is the Cooley-Tukey [4] version. This algorithm was developed in 1965 and was the first major breakthrough in an implementation of the FFT. Many consider the FFT to be in the top ten of the most important algorithms discovered in the 20th century. Moving forward, when we mention the FFT, we are specifically referencing the Cooley-Tukey FFT algorithm, unless indicated otherwise.

The FFT efficiently implements the DFT using a divide and conquer approach. In each iteration, it splits the even and odd terms of a and recursively calculates FFTs of half the size. It achieves this in the following way.

Since $n = 2^k$, we are able to rewrite $f(x)$ as

$$\begin{aligned} f(x) &= (a_0 + a_2x^2 + ... + a_{n-2}x^{n-2}) + x(a_1 + a_3x^2 + ... + a_{n-1}x^{n-2}) \\ &= b(x^2) + xc(x^2). \end{aligned}$$

where $b(x) = a_0 + a_2x + ... + a_{n-2}x^{n/2-1}$ and $c(x) = a_1 + a_3x + ... + a_{n-1}x^{n/2-1}$. Notice that $b(x)$ and $c(x)$ are both half the size of $f(x)$.

From here, the properties of w, outlined in Lemma 2.1 below, are utilized by the FFT.

Lemma 2.1. *Let $w \in F$ be a primitive nth root of unity. If 2 divides n, then:*

I . $w^i = -w^{i+n/2}$

II . w^2 *is a primitive* $(n/2)$*th root of unity.*

III . $w^0 + w^1 + ... + w^{n-1} = 0$

Proofs for the properties in Lemma 2.1 can be found in [9].

Property I in Lemma 2.1 tells us that $b(x^2)$ and $c(x^2)$ evaluated at $x = w^i$ is equivalent to $b(x^2)$ and $c(x^2)$ evaluated at $x = -w^{i+n/2}$. This fact allows us to save approximately half of the work required to calculate A. To compute an FFT of size $n/2$, we need to use property II in the recursive call and, since $n = 2^k$, we can apply property I again to save another one-fourth of the work. Algorithm 1 presents pseudo code for the FFT.

We will give two complexity analyses for Algorithm 1. One that only counts the number of multiplications in F, as is typical for an FFT analysis. And another that counts all arithmetic operations in F, which we need later.

Let $T(n)$ be the number of multiplications in F that the FFT needs. First, if $n = 1$, then no multiplications are performed, thus $T(1) = 0$. Next, there are two recursive calls of size $n/2$ and we need one multiplication to compute the w^2 in them. Finally, we conduct two multiplications, $wi \cdot C_i$ and $wi \cdot w$, in the for loop that executes $n/2$ times. Hence, the for loop requires n multiplications in total. Therefore, the recurrence relation for Algorithm 1 is

Input : $n = 2^k, a = [a_0, a_1, ..., a_{n-1}] \in \mathbb{F}_p^n, p$ is prime, and $w \in \mathbb{F}_p$ of order n.

Output: $A = [f(1), f(w), f(w^2), ..., f(w^{n-1})] \in \mathbb{F}_p^n$, where $f(w^k) = \sum_{i=0}^{n-1} a_i w^{ik}$.

```
if n = 1 then
    return a
end
n2 ← n/2
b ← [a_0, a_2, ..., a_{n-2}]
c ← [a_1, a_3, ..., a_{n-1}]
B ← FFT(n2, b, w^2, p)
C ← FFT(n2, c, w^2, p)
wi ← 1;
for i from 0 to n2 − 1 do
    T ← wi · C_i
    A_i ← B_i + T
    A_{n2+i} ← B_i − T
    wi ← wi · w
end
return A
```

Algorithm 1: FFT

$$T(n) = 2T(n/2) + n + 1.$$

Solving the recurrence in Maple with the `rsolve` command gives $T(n) = n \log n + n - 1 \in O(n \log n)$. This recurrence relation can also be solved with the Master Theorem. It is the second case of Theorem 4.1 in [5]. And so, by the Master Theorem, $T(n) \in O(n \log n)$.

Algorithm 1 can be improved using an optimization found by Law and Monagan in [12]. Suppose we precompute the powers of w in an array, $W = [1, w, w^2, ..., w^{n/2-1}]$, of length $n/2$ and access them as needed using

$$w^{si} = W_{si} \quad \text{for } 0 \le i < \frac{n}{2^s} \quad \text{where } s \in \{1, 2, 4, ..., \frac{n}{2}\}.$$

This would reduce the number of multiplications in the FFT by half as we do not need to calculate w^2 in the recursive calls nor $wi \cdot w$ in the for loop. Thus, the new recurrence relation for $T(n)$ is

$$T(n) = 2T(n/2) + n/2.$$

which solves to $T(n) = \frac{1}{2}(n \log n) \in O(n \log n)$.

Now, let $\mathbf{F_n}$ be the number of arithmetic operations in F that the FFT needs using the Law and Monagan optimization. For each multiplication, there is also an addition and subtraction, and so the recurrence relation is

$$\mathbf{F_n} = 2\mathbf{F_{n/2}} + 3(n/2)$$

which solves to $\mathbf{F_n} = \frac{3}{2}(n \log n) \in O(n \log n)$. We now have the following theorem.

Theorem 2.2. *Let* $\mathbf{F_n}$ *be the number of arithmetic operations in* F *required to compute an FFT of size* n*. Then,* $\mathbf{F_n} = \frac{3}{2}(n \log n)$*.*

The Cooley-Tukey FFT algorithm can be used with any composite length but performs best with highly composite lengths, such as powers of two. If n is not a power of two, then we can pad a with zeroes to increase its length to a power of two, however, this will slow down the running time. This is why we assumed that $n = 2^k$; in an effort to make our implementation as efficient as possible.

Observe that the FFT requires w to exist, however, not every field has a primitive nth root of unity. It is recognized that, for a prime power q, a finite field $\mathbb{F}_q$ with q elements contains a primitive nth root of unity if and only if n divides $q-1$. [7]. This is why we assumed that we are working in F in the introduction. We now know that the finite field $\mathbb{F}_p$ has a primitive nth root of unity if and only if n divides $p-1$. And $\mathbb{F}_p$ is the field that we will use in all of our implementations. All algorithms given in this paper can work over any general field that contains a primitive nth root of unity with $n = 2^k$. When coding, we used $\mathbb{F}_p$ since that is the implementation that came most naturally. In $\mathbb{F}_p$, for the prime p, we will use a Fourier prime. A Fourier prime is a prime p where $p-1$ is divisible by some large power of two. This will allow us to apply the FFT and, later on, FFT multiplication on polynomials with large degrees.

2.1 The Inverse Fast Fourier Transform

The Inverse FFT (IFFT), as the name implies, is the opposite or reverse of the FFT. And so, since the FFT evaluates, this suggests that the IFFT interpolates. Indeed, the IFFT determines the coefficients of f by interpolating the n outputs of f evaluated at the powers of w. This can be achieved using Algorithm 1 with some variation.

Given $n = 2^k, w_{nv} = w^{-1} \in \mathbb{F}_p$, and the FFT of $f(x), A = [f(1), f(w), ..., f(w^{n-1})] \in \mathbb{F}_p^n$, the IFFT returns the coefficient vector of $f(x), a = [a_0, a_1, ..., a_{n-1}] \in \mathbb{F}_p^n$. It is defined by the vector

$$[a_k = \frac{1}{n} \sum_{i=0}^{n-1} A_i w_{nv}^{ik} \colon 0 \leq k \leq n-1] \in F^n.$$

Now, consider the following Vandermonde linear system of equations

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{n-1} \\ 1 & w^2 & w^4 & \cdots & w^{2n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{n-1} & w^{2n-2} & \cdots & w^{(n-1)^2} \end{bmatrix}}_{V_w} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} f(1) \\ f(w) \\ f(w^2) \\ \vdots \\ f(w^{n-1}) \end{bmatrix}}_{A} \Rightarrow V_w a = A.$$

Suppose we have A and we wish to find a. One way to do this would be to solve the linear system $V_w \cdot a = A$ for a. We could also use Gaussian elimination to calculate the inverse V_w^{-1} and then compute $a = V_w^{-1} \cdot B$. However, solving and inverting both require $O(n^3)$ operations in F when Gaussian elimination is used [9]. To improve on this, we need the following two lemmas.

Lemma 2.3. *Let $w \in F$ be a primitive nth root of unity. Then, $w^{-1} = w^{n-1}$ and w^{-1} is a primitive nth root of unity.*

Proof. We know that $w^n = 1$. This implies that $w^{n-1} \cdot w = 1$. Multiplying both sides by w^{-1}, we find that $w^{n-1} = w^{-1}$.

Now, towards a contradiction, suppose that $(w^{-1})^k = 1$ for some $1 \leq k \leq n-1$. Since $w^n = 1$, this means that $w^n \cdot (w^{-1})^k = 1$, which implies that $w^{n-k} = 1$ for some $1 \leq n-k \leq n-1$. But this implies that w is not a primitive nth root of unity, which is a contradiction. Therefore, w^{-1} is a primitive nth root of unity. □

Lemma 2.4. $V_w \cdot V_{w^{-1}} = nI \Rightarrow V_w^{-1} = \frac{1}{n} \cdot V_{w^{-1}}$.

A proof of the above lemma can be found in [8]. The proof makes use of property III of Lemma 2.1.

Thus, to execute the IFFT or, in other words, to interpolate $a = [a_0, a_1, ..., a_{n-1}]$ from $A = [f(1), f(w), ..., f(w^{n-1})]$, we do the following

$$a \in \mathbb{F}_p^n = V_w^{-1} \cdot A = \frac{1}{n}(V_{w^{-1}} \cdot A) = \frac{1}{n}\text{FFT}(n, A, w^{-1}, p) \in \mathbb{F}_p^n.$$

The running time of the IFFT is $\mathbf{F_n}$ arithmetic operations for the FFT plus n multiplications of a vector in $\mathbb{F}_p^n$ by n^{-1}, therefore, the IFFT also costs $O(n \log n)$ arithmetic operations in F. Now, the IFFT, together with the FFT, allows us to perform fast multiplication, which we will see in the next section.

2.2 Fast Polynomial Multiplication

Suppose we have two polynomials $f(x) = a_0 + a_1x + a_2x^2 + ... + a_dx^d$ and $g(x) = b_0 + b_1x + b_2x^2 + ... + b_dx^d$. Let their product be $h(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{2d}x^{2d}$. The classical multiplication algorithm multiplies each term in g by each term in f. Since both polynomials have at most $d+1$ terms, this means that, in the worst case, we would need $(d+1)^2$ multiplications in F to find $h(x)$. This can be reduced by using the FFT.

First, pick $n > 2d$ with $n = 2^k$. Then, compute w of order n in $\mathbb{F}_p$. Now, using the FFT, evaluate f and g at the n powers of w. Next, from those results, obtain the products $h(w^i) = f(w^i)g(w^i)$ through pairwise multiplication. Finally, acquire the coefficients of h by interpolating at the product values using the IFFT.

Suppose we have $a = [a_0, a_1, ..., a_d] \in \mathbb{F}_p^{d+1}, b = [b_0, b_1, ..., b_d] \in \mathbb{F}_p^{d+1}$, where a and b are the coefficient vectors of two polynomials $f(x)$ and $g(x)$. Then, a and b must be padded with zeros to length n. We define $A = [a_0, a_1, ..., a_d, 0, ..., 0] \in \mathbb{F}_p^n$ and $B = [b_0, b_1, ..., b_d, 0, ..., 0] \in \mathbb{F}_p^n$.

And so, if we are given A, B, and $w \in \mathbb{F}_p$, where w is a primitive nth root of unity, then, FFT multiplication returns $c = [c_0, c_1, ..., c_{2d}, 0, ..., 0] \in \mathbb{F}_p^n$, where c is the coefficient vector of the polynomial $h(x)$ and $h(x) = f(x) \times g(x) \in \mathbb{F}_p[x]$. Algorithm 2 outlines FFT multiplication.

Input : $n = 2^k, A = [a_0, a_1, ..., a_d, 0, ..., 0] \in \mathbb{F}_p^n, B = [b_0, b_1, ..., b_d, 0, ..., 0] \in \mathbb{F}_p^n$, and $w \in \mathbb{F}_p$ of order n, where p is a prime.
Output: $c = [c_0, c_1, ..., c_{2d}, 0, ..., 0] \in \mathbb{F}_p^n$.

1 $a \leftarrow \text{FFT}(n, A, w, p)$
2 $b \leftarrow \text{FFT}(n, B, w, p)$
3 $C \leftarrow [a_0 \cdot b_0, a_1 \cdot b_1, ..., a_{n-1} \cdot b_{n-1}]$
4 $c \leftarrow \text{FFT}(n, C, w^{-1}, p)$
5 $nv \leftarrow n^{-1}$
6 $c \leftarrow [nv \cdot c_0, nv \cdot c_1, ..., nv \cdot c_{n-1}]$
7 **return** c

Algorithm 2: FFTMult

Let $M(n)$ be the number of arithmetic operations in F required for multiplying two polynomials of degree less than or equal to n using FFTMult. Algorithm 2 calls the FFT three times as well as performs n multiplications each in lines 3 and 6. For the previously described Law and Monagan [12] optimization of the FFT, we must compute W and W^{-1}, which both cost $n/2$ multiplications. We also compute the inverse of w in line 4 and the inverse of n in line 5. The equation for $M(n)$ is shown in the following theorem.

Theorem 2.5. *Let $M(n)$ be the number of arithmetic operations in F required for a fast mutliplication of two polynomials of degree less than or equal to n. Then,*

$$M(n) = 3\mathbf{F_n} + 3n + 2 = \frac{9}{2}(n \log n) + O(n) \in O(n \log n).$$

We must now remark on a key idea. Let $f = \sum_{i=0}^{n-1} a_i x^i$ and $g = \sum_{j=0}^{n-1} b_j x^j$ in $F[x]$. In classical polynomial multiplication, since both f and g have degree $n-1$, this means that their product would have degree $2n-2$ and

$$f \times g = h = \sum_{k=0}^{2n-2} c_k x^k \text{ in } F[x],$$

where the coefficient c_k is given by: $c_k = \sum_{i+j=k} a_i b_j$. This is not the case for FFTMult.

We define the convolution, $f * g$, with respect to n of the polynomials f and g to be the polynomial h' such that

$$f * g = h' = \sum_{k=0}^{n-1} c'_k x^k \text{ in } F[x]$$

where the coefficient c'_k is given by

$$f * g = h' = \sum_{i+j=k \bmod n} a_i b_j = \sum_{i=0}^{n-1} a_i b_{k-j} \text{ for } 0 \leq j \leq n-1.$$

Then, $f * g = f \times g \bmod x^n - 1$ or $h' = h \bmod x^n - 1$.

The result of a polynomial $f \bmod x^n$ is just the remainder after dividing f by x^n. That is, $f = f^\star \bmod x^n$, where f and $f^\star$ have the same first n terms.

In other words, convolution and polynomial multiplication are equivalent mod $x^n - 1$. This is because we can regroup the sum in the polynomial h as $h = \sum_{k=0}^{n-1}(c_k + c_{k+n}x^n)x^k$ which simplifies to $h = \sum_{k=0}^{n-1} c'_k x^k \bmod x^n - 1$.

Now, we will consider convolution in regards to the FFT. We have that $FFT(f * g) = FFT(f) \cdot FFT(g)$, where we use pointwise multiplication to find the product of vectors $FFT(f)$ and $FFT(g)$. This is true since $f * g = f \times g \bmod x^n - 1$, and so there exists some $q \in F[x]$ where $f * g = f \times g + q(x^n - 1)$. Thus, since $w^n = 1$, we have

$$\begin{aligned} (f * g)(w^l) &= f(w^l)g(w^l) + q(w^l)(w^{ln} - 1) \\ &= f(w^l)g(w^l) + q(w^l) \cdot 0 \text{ for } 0 \leq l \leq n-1. \end{aligned}$$

Thus, if the degree of the product h of $f \times g$ is greater than n, then the coefficients of h begin to wrap back around. This means that c_n is added to c_0 and c_{n+1} is added to c_1 and so forth. And so, given coefficient vectors a and b of polynomials f and g, respectively, FFTMult returns their product mod $x^n - 1$ if an FFT of order n is used but $n \leq$ degree f + degree g.

Fast multiplication has been coded into Maple as the `Expand(...) mod` p command. It takes two polynomials and multiplies them using an FFT, then returns their product as a polynomial in $\mathbb{F}_p[x]$. If coding in Maple, it is sufficient to use this command for all subsequent algorithms in this paper, except for one, if the reader does not wish to explicitly code the FFT and FFT multiplication. However, the reader should be aware that `Expand(...) mod` p does not compute the product mod $x^n - 1$. This will be important in the next section.

For example, suppose $n = 2, p = 97$ and we have two polynomials $f(x) = 1 + 2x$ and $g(x) = 3+4x$. Then, $h(x) = f(x) \times g(x) = 8x^2+10x+3$ and $h(x) \bmod x^2-1 = 8+10x+3 = 10x+11$. For FFTMult, we would input two arrays $a = [1, 2]$ and $b = [3, 4]$ and the procedure would return $c = [11, 10] \Rightarrow h(x) = 10x + 11$ in $\mathbb{F}_{97}[x]$.

2.3 Bluestein's Algorithm

A few years after the Cooley-Tukey FFT was published, Bluestein presented in [2] a different version of the FFT. Bluestein's algorithm accomplished this by reorganizing the DFT as a convolution. Bluestein's algorithm performs well with prime sizes but is slower than the Cooley-Tukey FFT for composite sizes.

Let $a = [a_0, a_1, ..., a_{n-1}] \in F$ be the coefficient vector of the degree $n-1$ polynomial $f(x)$ and let w be a primitive nth root of unity. Now, recall the formula for the DFT

$$[A_k = f(w^k) = \sum_{i=0}^{n-1} a_i w^{ik} \colon 0 \leq k \leq n-1] \in F^n.$$

The key idea, credited to Bluestein, considers the identity

$$ik = \frac{1}{2}(k^2 + i^2 - (k-i)^2).$$

The $\frac{1}{2}$ in the above expression will give us a bit of trouble in the DFT. To circumvent this, throughout this section, our goal will be to compute

$$X_k = \sum_{i=0}^{n-1} a_i w^{2ik}, \quad 0 \leq k \leq n-1. \tag{2.1}$$

Now, substituting the identity into the w^{2ik} in (2.1) gives

$$w^{2ik} = w^{k^2} \cdot w^{i^2} \cdot w^{-(k-i)^2}.$$

The first term, w^{k^2}, does not depend on i, thus we can pull it out of the sum. The second term must still be multiplied with a_i and the last term is what causes (2.1) to have the form of a convolution.

Therefore, we can rewrite (2.1) as follows

$$X_k = w^{k^2} \cdot \sum_{i=0}^{n-1} a_i \cdot w^{i^2} \cdot w^{-(k-i)^2}, \quad 0 \leq k \leq n-1. \tag{2.2}$$

We define three sequences b_k, r_i, and s_i, such that

$$b_k = w^{k^2}, \qquad r_i = a_i w^{i^2}, \qquad s_i = w^{-i^2} \qquad \text{for } 0 \leq i, k \leq n-1.$$

At this point, the summation in (2.2) is exactly a convolution of r_i and s_i, and

$$X_k = b_k \cdot \sum_{i=0}^{n-1} r_i s_{k-i}, \quad \text{for } 0 \leq k \leq n-1. \tag{2.3}$$

We know from the previous section that a convolution can be carried out by FFT multiplication, and this particular convolution can be carried out with a FFT multiplication

of size at least $2n-1$. Thus, if we increase the length of r and s by zero padding to a highly composite length such as a power of two, then the convolution can be quickly performed by the Cooley-Tukey FFT in $O(n \log n)$ operations in F. This allows for prime length FFTs to be computed by a more efficient FFT algorithm.

Let m be the next power of two greater than or equal to $2n-1$. We must zero pad r and s to length m. We extend r_i of length n to a vector R_i of length m by zero padding in the usual way. However, zero padding s_i in the usual way is not sufficient. The s_{k-i} term in (2.3) forces both positive and negative values of i to be needed for s_i. Since $s_i = w^{-i^2}$, we have that $s_{-i \bmod n} = s_i$. Then, $-i = m - i$, by the periodic boundaries of the FFT.

To summarize, we extend r and s by letting:

$$R_i = \begin{cases} r_i, & \text{for } 0 \le i \le n-1 \\ 0, & \text{otherwise} \end{cases} \qquad S_i = \begin{cases} s_i, & \text{for } 0 \le i \le n-1 \\ s_{m-i}, & \text{for } m-n+1 \le i \le m-1 \\ 0, & \text{for } n \le i \le m-n \end{cases}$$

Now, we must execute an FFT multiplication of size m with input arrays R and S and output a vector c. The convolution has now been computed and so all there is left to do is to multiply b_k by c_k for all k and then we are done. Included in Algorithm 3 is pseudo code for Bluestein's algorithm.

At this time, we will provide an example of Bluestein's algorithm. Suppose that we have $n = 7, a = [1,2,3,4,3,2,1], w = 49$ and $p = 7 \cdot 2^4 + 1$. Then:

$$b = [1, 49, 106, 28, 28, 106, 49];$$
$$r = [1, 98, 92, 112, 84, 99, 49];$$
$$s = [1, 30, 16, 109, 109, 16, 30];$$
$$m \ge 2n - 1 = 16;$$
$$w1 = \text{a primitive } m\text{th root of unity} = 40;$$
$$R = [1, 98, 92, 112, 84, 99, 49, 0, 0, 0, 0, 0, 0, 0, 0, 0];$$
$$S = [1, 30, 16, 109, 109, 16, 30, 0, 0, 0, 30, 16, 109, 109, 16, 30];$$
$$c = [16, 99, 41, 95, 25, 52, 60, 15, 84, 44, 96, 43, 31, 29, 65, 11].$$
$$X = [16, 105, 52, 61, 22, 88, 2].$$

One can check that our output, X, is indeed equal to the X_k in (2.1).

Note that in our implementation of Bluestein's algorithm, we evaluated f at w^{2i} instead of w^i for $0 \le i \le n-1$. As we compute the powers of w^{2i}, we first only get the even exponents. Then, when the exponent of w becomes greater than n, because of the nature of the primitive nth root of unity, it is the same as that exponent mod n. If n is odd, then modding an

Input : A vector $a = [a_0, a_1, ..., a_{n-1}] \in \mathbb{F}_p^n$, which is the coefficients of a polynomial $f(x)$ in $\mathbb{F}_p[x]$ with prime p, the number of coefficients n, and a primitive nth root of unity, $w \in \mathbb{F}_p$.

Output: $X = [f(1), f(w), f(w^2), ..., f(w^{n-1})]$, where $f(w^k) = \sum_{i=0}^{n-1} a_i w^{2ik}$.

1 $W \leftarrow [w^0, w^1, ..., w^{n-1}]$
2 **for** i **from** 0 **to** $n-1$ **do**
3 $\quad t \leftarrow i^2 \bmod n$
4 $\quad b_i \leftarrow W_t$
5 $\quad r_i \leftarrow a_i \cdot b_i \in \mathbb{F}_p$
6 $\quad t \leftarrow -t \bmod n$
7 $\quad s_i \leftarrow W_t$
8 **end**
9 $m \leftarrow$ The next power of two $\geq 2n-1$
10 $w1 \leftarrow$ A primitive mth root of unity
11 Set $R_i = r_i$ for $0 \leq i \leq n-1$; and set $R_i = 0$ for $n \leq i < m$
12 Set $S_i = s_i$ for $0 \leq i \leq n-1$; set $S_i = s_{m-i}$ for $m-n+1 \leq i \leq m-1$; and set $S_i = 0$ for $n \leq i \leq m-n$
13 $c \leftarrow \text{FFTMult}(m, R, S, w1, p)$
14 $X \leftarrow [b_0 \cdot c_0, b_1 \cdot c_1, ..., b_{n-1} \cdot c_{n-1}] \in \mathbb{F}_p^n$
15 **return** X

Algorithm 3: Bluestein's Algorithm

even exponent produces an odd exponent, thereby giving us all the powers of w. Thus, if we wish to convert X_k into A_k, that is the regular DFT, then, for odd n, we can use the following method: $X_0 = A_0, X_1 = A_2, X_2 = A_4, ..., X_{(n-1)/2} = A_{n-1}, X_{(n+1)/2} = A_1$ $X_{(n+3)/2} = A_3, ..., X_{n-1} = A_{n-2}$. The even case is disregarded here, as one would normally use an FFT algorithm that is more efficient for this type of n.

2.3.1 Bluestein Complexity

To simplify the cost analysis and the input for the timings, we will assume that $n = 2^k$, which implies that $m = 2n$, here and in the next section.

Let $T(n)$ be the number of arithmetic operations in F that Bluestein's algorithm performs. There are n multiplications each in lines $1, 5$, and 14. Then, in line 13, there is an FFT multiplication of size m. Recall that $M(n) = \frac{9}{2}(n \log n)$, by Theorem 2.5. Hence,

$$\begin{aligned} T(n) &\leq M(m) + 3n \\ &\leq M(2n) + O(n) \\ &\leq \frac{9}{2}(2n \log 2n) + O(n) \\ &\leq 9n \log 2n + O(n) \in O(9n \log 2n) \in O(n \log n). \end{aligned}$$

Therefore, Bluestein's algorithm requires $O(n \log n)$ arithmetic operations in F.

2.3.2 Bluestein Timings

We have implemented the FFT and Bluestein's algorithm in Maple and in C. Code is provided in Appendix A and Appendix B.

We randomly generate a vector, a, of size n which are the coefficients of f, where $n = 2^k$ and $8 \leq k \leq 24$. We will work over the field $\mathbb{F}_p$, where $p = 7 \cdot 2^{26} + 1$. Then, we compute w, a primitive n root of unity. This will be used as input for Bluestein's algorithm in Maple and in C to allow for comparisons. The timings were run on an Intel Xeon E5 2660 CPU with 64 gigabytes of RAM.

n	Maple	Growth Factor	C	Growth Factor	Maple/C
2^{10}	293 ms	N/A	0 ms	N/A	N/A
2^{11}	568 ms	1.94	1 ms	N/A	568
2^{12}	1.23 s	2.17	2 ms	2.00	615
2^{13}	2.62 s	2.13	4 ms	2.00	655
2^{14}	5.60 s	2.14	9 ms	2.25	622
2^{15}	12.22 s	2.18	20 ms	2.22	611
2^{16}	26.05 s	2.13	42 ms	2.10	620
2^{17}	56.23 s	2.16	89 ms	2.12	632
2^{18}	2.05 min	2.19	188 ms	2.11	654
2^{19}	4.39 min	2.14	386 ms	2.05	682
2^{20}	9.11 min	2.08	815 ms	2.11	671
2^{21}	18.68 min	2.05	1.71 s	2.10	655
2^{22}	39.42 min	2.11	3.60 s	2.11	657
2^{23}	1.37 hr	2.09	7.54 s	2.09	654
2^{24}	2.77 hr	2.02	15.64 s	2.07	638

Table 2.1: Comparing the timings of Bluestein's Algorithm in Maple and in C

The first column of Table 2.1 displays values of n ranging from 2^{10} up to 2^{24}. The second and fourth columns present the time it took to run Bluestein's algorithm in Maple and in C at that value of n. The third and fifth columns show, after n is doubled, the factor by which the computation time increases. The last column divides the time needed in Maple

by the time needed in C, which tells us how much faster C was able to execute the same instance of input as Maple.

For Maple, as n doubles, the factor by which the time increases ranges from 1.94 to 2.19, with an average of 2.1. For C, the factor by which the time increases, as n is doubled, ranges from 2.00 to 2.25, with an average of 2.1.

Thus, for both Maple and C, we can see that the time taken increases by just a factor of two each time n is doubled, which suggests that these are both $O(n \log n)$ algorithms.

C was faster than Maple for all n. The average number in the last column is 638. This means that, on average, C was able to execute the same instance of Bluestein's algorithm 638 times faster than Maple!

The reason why the execution speed in C is so much faster than in Maple is due to the fact that C is a compiled language and Maple is an interpreted language.

Compiled languages use a compiler to translate a program into machine language that a computer can execute. Every time a change is made to the program, it needs to be recompiled. These types of languages also give the user more control over memory management and access to special hardware instructions. On the other hand, interpreted languages execute a program directly, without compiling it, and they have to interpret what the user has actually written. Interpreted languages are usually much slower than compiled languages. This is because each line of an interpreted program must be translated, interpreted, and executed every time it is run and this slows the whole process down. In general, interpreted languages are slower than compiled languages when they must do lots of small pieces of work like arithmetic with small integers, such as in the FFT. Interpreted languages perform better when they must do big pieces of work like multiplying polynomials or matrices. This is because the interpreter overhead is low compared with the work being done.

Maple and other interpreted languages also have other benefits. Mainly, Maple is more intuitive than C and less complicated to use. It is easier to revise and debug than C, since it doesn't have to recompile the program every time. Maple is interactive in that it allows users to input data directly into commands, such as factor, gcd, etc., which will then produce output from that input. This is not the case in C, where a user must write a complete program before any output can be seen. Both programming languages have advantages and disadvantages, and the reader should be mindful of them as they choose which one works best for them.

Chapter 3

Fast Polynomial Division

Let $f(x) = a_0 + a_1x + ... + a_{n-1}x^{n-1} + a_nx^n \in F[x]$ and $g(x) = b_0 + b_1x + ... + b_{m-1}x^{m-1} + b_mx^m \in F[x]$ be polynomials such that $n \geq m$ and $m \geq 0$. Then, for all f, g with $g \neq 0$ there exist unique polynomials $q, r \in F[x]$ such that $f = gq + r$, where either $r = 0$ or the degree of r is less than the degree of g.

To compute the quotient, q, or remainder, r, using the standard polynomial long division algorithm, we would need $(n-m+1)m$ multiplications in F in the worst case. This is because there are at most $n - m + 1$ steps in the division algorithm and each step does at most m multiplications. If $n = 2m$, then the algorithm performs at most $(2m-m+1)m = m^2+1 \in O(m^2) = O(n^2)$ multiplications in F. In this section, we will improve that complexity to $O(n \log n)$ using the FFT-based multiplication algorithm.

To begin, we will define the reciprocal polynomials of f and g as

$$f^r(x) = x^n f(1/x) = a_0x^n + a_1x^{n-1}... + a_{n-1}x + a_n$$

$$g^r(x) = x^m g(1/x) = b_0x^m + b_1x^{m-1}... + b_{m-1}x + b_m.$$

Observe that f^r and g^r are merely the polynomials f and g but with their coefficients reversed. To illustrate, suppose $f(x) = 2 + 3x + 4x^2 + 5x^3$. Then, $f^r(x) = x^3 f(1/x) = x^3(2 + 3(1/x) + 4(1/x)^2 + 5(1/x)^3) = 2x^3 + 3x^2 + 4x + 5$.

Thus, we can rewrite $f = gq + r$ as $f^r = g^r q^r + x^{(n-m+1)}r^r$. This implies that $f^r = g^r q^r \mod x^{n-m+1}$.

The definition of $f \mod x^n$ was given previously. A similar concept occurs for power series. If $f = f^\star + O(x^n)$, then $f^\star$ is known as an order n approximation of f. For example, if $f(x) = 1 + x + x^2 + x^3$ then $f \mod x^2$ is $f^\star = 1 + x$ and $f^\star$ is an order 2 approximation of f, where $1 + x + x^2 + x^3 = 1 + x + O(x^2)$.

Now, g^r is invertible mod x^{n-m+1} as a power series since it has a nonzero constant coefficient [7]. Therefore,

$$q^r = f^r \frac{1}{g^r} \bmod x^{n-m+1}.$$

From here, we can easily acquire the quotient and remainder by calculating $q = (q^r)^r$ and $r = f - gq$, using fast multiplication for the product gq. Thus, our task is to find a fast algorithm to compute the inverse of g^r to an order $N = n-m+1$ approximation. This can be done using Newton's method for power series inversion. Recall Newton's iteration method for solving an equation $t(y) = 0$ that starts with an estimate y_0. In our case, $y_0 = b_m^{-1}$, where b_m is the constant term of g^r, since $tb_m^{-1} = g^r(x) - b_m = 0 \bmod x$. We can determine all subsequent estimates with the formula

$$y_i = y_{i-1} - \frac{t(y_{i-1})}{t'(y_{i-1})} \bmod x^{2^i} \text{ for } 0 \leq i \leq \log_2 N.$$

To compute $1/g^r$, we use the fact that $t(y) = g^r - y^{-1} \Rightarrow a(1/g^r) = 0$ and $t'(y) = y^{-2}$. Therefore

$$y_i = y_{i-1} - \frac{g^r - y_{i-1}^{-1}}{y_{i-1}^{-2}} = 2y_{i-1} - g^r y_{i-1}^2. \tag{3.1}$$

We can rewrite (3.1) as follows

$$y_i = 2y_{i-1} - g^r y_{i-1}^2 = y_{i-1} + y_{i-1}(1 - g^r y_{i-1}) \bmod x^{2^i}. \tag{3.2}$$

A naive implementation of Newton inversion would use (3.1), whereas we will use (3.2) in Algorithm 4. To make Newton's method efficient when N is not a power of two, we compute $1/g^r$ recursively to mod $x^{\lceil N/2 \rceil}$.

Input : A polynomial $g(x)$ in $\mathbb{F}_p[x]$, p is a prime, with $g(0) = b_0 \neq 0$, and an integer $n > 0$.
Output: $g^{-1} \bmod x^n \in \mathbb{F}_p[x]$.

```
1  if n = 1 then
2  |  return g_0^{-1} ∈ F_p
3  end
4  m ← ⌈n/2⌉
5  a ← g mod x^m
6  y ← FastNewton(a, m, p)
7  z ← (1 − g × y)/x^m
8  z ← y × z mod x^{n−m}
9  y ← y + x^m z
10 return y
```

Algorithm 4: FastNewton

The naive implementation of FastNewton using (3.1) would have replaced lines $7, 8$ and 9 with the condensed single line $2y - gy^2 \bmod x^n$, which returns the correct answer, but also requires a larger than necessary FFT multiplication, thus increasing the cost.

Let $I(n)$ be the number of arithmetic operations in F for computing $1/g^r \bmod x^n$ using Newton's method. Recall that $M(n)$ is the cost of multiplying two polynomials of degree at most n in F. Then, if $n = 2^k$ and the naive implementation is used, $I(n) = I(n/2) + M(n/2) + M(n) + O(n)$. The $I(n/2)$ comes from the recursive call. When we square y, we are actually multiplying two degree $n/2 - 1$ polynomials and so this is where we get $M(n/2)$. After squaring y, the result has degree $n - 2$. This forces the cost of multiplying the result by g to be $M(n)$. Finally, there is some linear work, including $2 \cdot y$ and the subtraction. Solving this recurrence relation for $I(n)$ gives $I(n) < 3M(n) + O(n)$.

In FastNewton, we multiply g and y together, which have degree $n/2$ and $n/2 - 1$, respectively. Hence, the cost of this multiplication is $M(n/2)$. The result of $g \times y$ is a degree $n - 1$ polynomial that we divide by $x^{n/2}$ in line 7, and so the z polynomial has degree $n - 1 - n/2 = n/2 - 1$. We know that this division is exact because, after line 6, it must be that $g \times y = 1 \bmod x^{n/2}$. Consequently, we then multiply two polynomials of degree $n/2 - 1$ in line 8, which has a cost of $M(n/2)$. We also require some $O(n)$ work. The base of the recursion is $I(1) = 1$ since, if $n = 1$, we only need one division to obtain g_0^{-1}. Thus, for FastNewton, we have that $I(n) = I(n/2) + M(n/2) + M(n/2) + O(n)$. Note that, if we assume that $M(n)$ is superlinear, i.e. $M(n) \notin O(n)$, then this implies that $2M(n/2) < M(n)$ [7]. That is, two multiplications of degree $n/2$ polynomials costs less than one multiplication of degree n polynomials. Now, we can solve the recurrence relation as follows:

$$
\begin{aligned}
I(n) &\leq I(n/2) + 2M(n/2) + O(n) \\
&\leq I(n/2) + M(n) + O(n) \\
&\leq I(n/4) + M(n/2) + M(n) + O(n) \\
&\vdots \\
&\leq I(1) + M(2) + M(4) + M(8) + ... + M(n/4) + M(n/2) + M(n) + O(n) \\
&\leq 1 + 2M(4) + M(8) + M(16) + ... + M(n/4) + M(n/2) + M(n) + O(n) \\
&\leq M(8) + M(8) + M(16) + ... + M(n/4) + M(n/2) + M(n) + O(n) \\
&\leq 2M(8) + M(16) + M(32) + ... + M(n/4) + M(n/2) + M(n) + O(n) \\
&\vdots \\
&\leq 2M(n/2) + M(n) + O(n) \\
&\leq M(n) + M(n) + O(n) \\
&\leq 2M(n) + O(n)
\end{aligned}
$$

Therefore, we have improved the cost of FastNewton by a factor of $\frac{3}{2}$.

Next, we will provide pseudo code for a fast division implementation in Algorithm 5.

Input : Polynomials f and g in $\mathbb{F}_p[x]$, where $g \neq 0$, g is monic, and p is a prime.
Output: The remainder r and quotient q in $\mathbb{F}_p[x]$ such that $f = gq + r$ with degree$(r) <$ degree(g) or $r = 0$.

1 $n \leftarrow$ degree(f)
2 $m \leftarrow$ degree(g)
3 **if** $n < m$ **then**
4 | **return** $(f, 0)$
5 **end**
6 $a \leftarrow g^r \bmod x^{n-m+1}$
7 $b \leftarrow$ Compute $a^{-1} \bmod x^{n-m+1}$ using Algorithm 4
8 $c \leftarrow f^r \times b \bmod x^{n-m+1}$
9 $q \leftarrow c^r$
10 $r \leftarrow f - g \times q$
11 **return** (r, q)

Algorithm 5: FastDivision

For the runtime of FastDivision, consider Theorem 3.1.

Theorem 3.1. *Let $D(n)$ be the number of arithmetic operations required for calculating the remainder of $f \div g$, where the degrees of f and g are $2n-2$ and $n-1$, respectively.*

$$D(n) \leq 4M(n) + O(n) \in O(n \log n).$$

Proof. In FastDivision, we call FastNewton once as well as compute one polynomial multiplication in line 8 of degree $2n-2$ by degree $n-1$ and one in line 12 of degree $n-1$ by degree $n-1$. Hence, using Theorem 2.5, we find the recurrence relation for FastDivision is:

$$\begin{aligned} D(n) &\leq I(n) + M(n) + M(n) \\ &\leq 2M(n) + O(n) + 2M(n) \\ &\leq 4M(n) + O(n) \in O(n \log n) \end{aligned}$$

□

And so, FastDivision requires work equivalent to at most four polynomial multiplications. Thus, using Algorithm 2, we can find the remainder of two polynomials in $O(n \log n)$ arithmetic operations in F.

Chapter 4

Fast Evaluation

We have seen in Sections 2.0 and 2.3 that we can evaluate f at the powers of a primitive nth root of unity w in $O(n \log n)$ time. What if we wish to evaluate f at n arbitrary points, $u_0, ..., u_{n-1}$? How fast can we compute $f(u_i) = v_i$ for $0 \leq i < n$? Recall that Horner's method can do this in $O(n^2)$ arithmetic operations in F. In this chapter, we prove the following theorem.

Theorem 4.1. *Let $E(n)$ be number of arithmetic operations in F needed to evaluate a degree $n-1$ polynomial f at n arbitrary points, $u_0, ..., u_{n-1}$. Then,*

$$E(n) < 11(n \log^2 n) + O(n \log n) + O(n) \in O(n \log^2 n).$$

4.1 The Subproduct Tree

In order to implement the fast evaluation algorithm, we must first build a subproduct tree. A subproduct tree is a complete binary tree of products of polynomials.

We build a subproduct tree using the evaluation points by starting from the leaves and working our way up to the root. Each node of the tree represents a monic polynomial that is constructed as the product of its two children. The leaves are the linear polynomials $x - u_i$ for $0 \leq i < n$. Refer to Figure 4.1 to view the subproduct tree.

The height of a tree is the number of edges in the longest simple downward path from root to leaf. A leaf node will have height zero. The height of a complete binary tree with n leaves is $\log n$ [5].

Before we move on, we will first introduce Lemma 4.2 which will be very helpful for bounding the cost of the product tree algorithm.

Lemma 4.2. *Let $n = 2^k$ and a, b, d be natural numbers with $b > 0$ and let c be a positive real number. Let S and T be functions where $S(n/2) = c \cdot (n \log_2 n)$, and*

$$T(1) = a, \quad T(n) \leq 2T(n/2) + bS(n/2) + dn$$

Then, we have:

$$T(n) < b \cdot \frac{1}{4} S(n) \log\, n + b \cdot \frac{1}{4} S(n) + d(n \log n) + na$$

Proof. We know that $n = 2^k$, which means that $k = \log_2 n$. Therefore, we see that $S(n/2) = c \cdot (n \log_2 n) = c \cdot 2^k \cdot k$. We can now more easily solve the recurrence for $T(n)$:

$$\begin{aligned}
T(n) &= 2T(n/2) + b \cdot c \cdot 2^k \cdot k + d \cdot 2^k \\
2T(n/2) &\leq 4T(n/4) + b \cdot 2 \cdot c \cdot 2^{k-1} \cdot (k-1) + d \cdot 2^{k-1} \\
&= 4T(n/4) + b \cdot c \cdot 2^k \cdot (k-1) + d \cdot 2^{k-1} \\
4T(n/4) &\leq 8T(n/8) + b \cdot 4 \cdot b \cdot c \cdot 2^{k-2} \cdot (k-2) + d \cdot 2^{k-2} \\
&= 8T(n/8) + b \cdot c \cdot 2^k \cdot (k-2) + d \cdot 2^{k-2} \\
&\vdots \\
2^{k-2}T(3) &\leq 2^{k-1}T(2) + b \cdot 2^{k-2} \cdot c \cdot 2^2 \cdot 2 + d \cdot 4 \\
&= 2^{k-1}T(2) + b \cdot c \cdot 2^k \cdot 2 + d \cdot 4 \\
2^{k-1}T(2) &\leq 2^k T(1) + b \cdot 2^{k-1} \cdot c \cdot 2 \cdot 1 + d \cdot 2 \\
&= 2^k \cdot a + c \cdot 2^k \cdot 1 + d \cdot 2 \\
&= na + b \cdot c \cdot n + d \cdot 2
\end{aligned}$$

Now, adding both sides of the inequalities and canceling equal terms, we obtain:

$$\begin{aligned}
T(n) &\leq b \cdot c \cdot n \cdot \sum_{i=0}^{k-1} (k-i) + d \cdot 2^k \cdot k + na \\
&= b \cdot c \cdot n \cdot (k^2/2 + k/2) + d(n \log n) + na \\
&= b \cdot [\frac{1}{2}(c \cdot n \cdot k^2) + \frac{1}{2}(c \cdot d \cdot k)] + d(n \log n) + na \\
&= b \cdot [\frac{1}{2}(c \cdot (n \log n) \log n) + \frac{1}{2}(c \cdot (n \log n))] + d(n \log n) + na \\
&= b \cdot [\frac{1}{2} S(n/2) \log n + \frac{1}{2} S(n/2)] + d(n \log n) + na \\
&< b \cdot \frac{1}{4} S(n) \log n + b \cdot \frac{1}{4} S(n) + d(n \log n) + na
\end{aligned}$$

Thus, $T(n) < b \cdot \frac{1}{4} S(n) \log\, n + b \cdot \frac{1}{4} S(n) + d(n \log n) + na$. □

We are now ready to see how to explicitly build the subtree.

4.1.1 Building Up the Subtree

Suppose that $u_0, ..., u_{n-1} \in F$ are the distinct evaluation points for a degree $n-1$ polynomial $f(x)$ and k is the height of the subproduct tree, denoted M. Then, if n is a power of two, $k = \log_2 n$. The idea of the subtree is to split the evaluation points into two equal halves and proceed recursively with each half. To start building the subtree, let $m_i = x - u_i$ for $0 \leq i < n$ be the leaves of the tree. Thus, for n evaluation points, we will have n leaves. We specify the polynomial $M_{i,j}$ to be the product of its two children. It is situated at height i of the tree and j nodes from the left for $0 \leq i \leq k$ and $0 \leq j \leq 2^{k-i} - 1$. Each leaf is defined by the polynomial $M_{0,j} = m_j = x - u_j$, while the root of the tree is defined by the largest polynomial, $M_{k,0} = \prod_{i=0}^{n-1} m_i$. Explicitly, we have that

$$M_{i,j} = m_{j \cdot 2^i} \cdot m_{j \cdot 2^i + 1} \cdots m_{j \cdot 2^i + (2^i - 1)} = \prod_{0 \leq l \leq 2^i - 1} m_{j \cdot 2^i + l}.$$

Hence, each $M_{i,j}$ is a subproduct with 2^i factors of $m = \prod_{0 \leq l \leq n-1} m_l = M_{k,0}$ and can be found recursively for all i, j using

$$M_{0,j} = m_j, \quad M_{i,j} = M_{i-1,2j} \times M_{i-1,2j+1}.$$

Proof of correctness of Algorithm 6 follows directly from above. We can see the layout of the subproduct tree in Figure 4.1.

Since we are working in a field and the evaluation points are distinct, this means that each $M_{i,j}$ in Figure 4.1 is a monic squarefree polynomial and its zero set is the jth node from the left on level i.

We will now give an example of the subproduct tree for $n = 4$. Let the evaluation points be: $u_0 = 4, u_1 = 3, u_2 = 2$, and $u_3 = 1$. Then, the subtree for those data points is given in Figure 4.2.

Suppose that $p = 97$. In Figure 4.3, we will show what the Figure 4.2 subproduct tree looks like in expanded form in $\mathbb{F}_p$.

The general method for building the subtree is presented as Algorithm 6 and called BUST.

Note that the subproduct tree does not rely on the polynomial being evaluated, it only relies on the evaluation points. As such, the construction of the subtree is considered a precomputation step in the fast evaluation algorithm. If there are multiple polynomials that must be evaluated at the same evaluation points, then we only need to execute BUST once and reuse it for the different polynomials.

Our implementation of BUST is recursive. If $n = 1$, we return $x - u_0 \in \mathbb{F}_p$. Then, we recursively call BUST twice to compute the left and right subtree. The first, denoted L, has

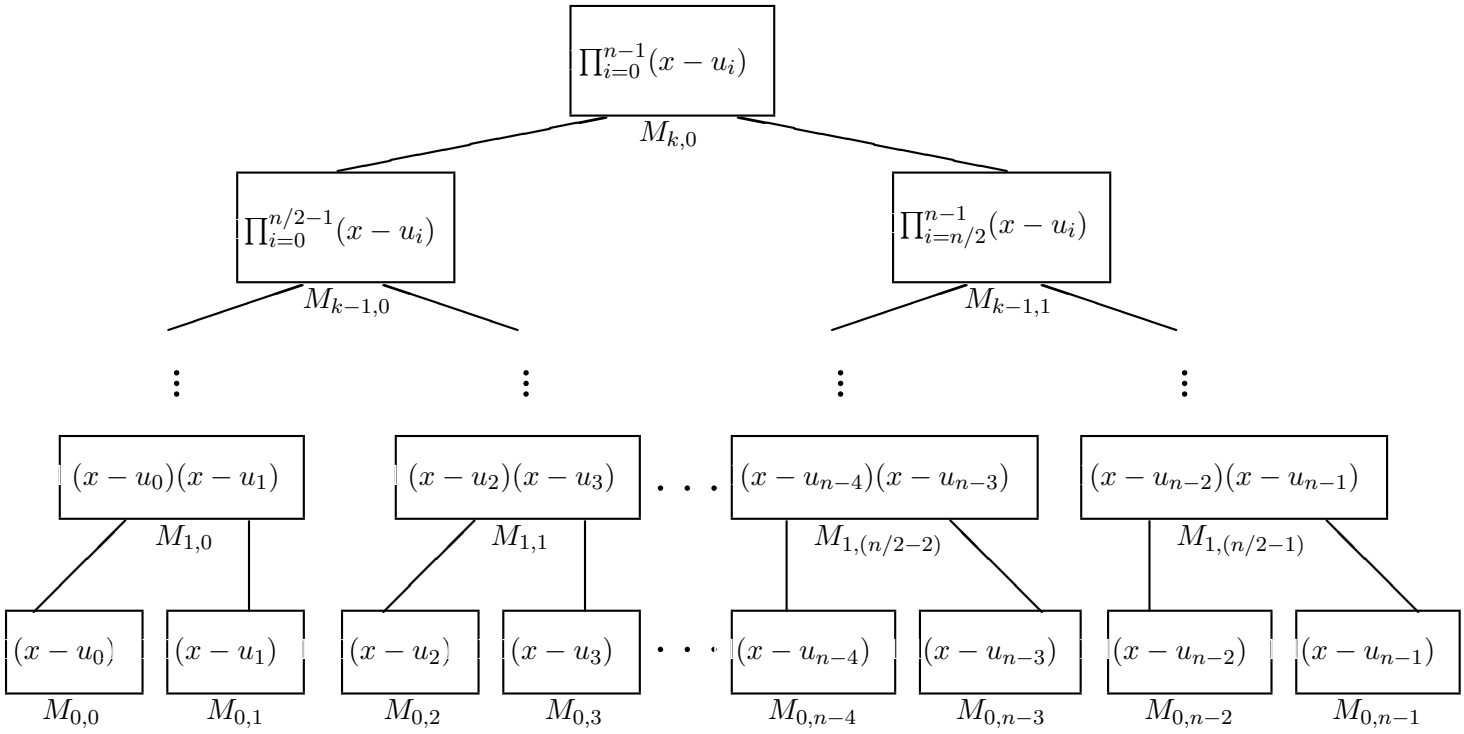

Figure 4.1: Subproduct Tree

input $n/2, [u_0, ..., u_{n/2-1}]$ and p. And the second, denoted R, has input $n/2, [u_{n/2}, ..., u_{n-1}]$ and p. Next, we let f be the product of the two newest polynomials added to the tree, found in L and R, both of which will have degree $n/2$. Finally, we return the subtree $[f, L, R]$.

Now, we can examine the running time for BUST using the recursive method.

Let $B(n)$ be the number of arithmetic operations needed to compute the subproduct tree. BUST can be executed with two recursive calls of size $n/2$ and one multiplication of polynomials of degree $n/2$. Thus, we have the recurrence relation:

$$B(n) = 2B(n/2) + M(n/2)$$

For $n = 1$, we must return $x - u_0$ and so $B(1) = 1$. If we let $M(n/2) = c \cdot (n \log_2 n)$ for some constant c, then we are able to use Lemma 4.2 to solve this recurrence. In this case, $a = 1, b = 1$, and $d = 0$ and we find that:

$$\begin{aligned} B(n) &< 1 \cdot \frac{1}{4}M(n)\log n + 1 \cdot \frac{1}{4}M(n) + 0 \cdot (n \log n) + 1 \cdot n \\ &= \frac{1}{4}M(n)\log n + \frac{1}{4}M(n) + O(n) \in O(n \log^2 n) \end{aligned}$$

The complexity analysis of BUST is summarized in Theorem 4.3.

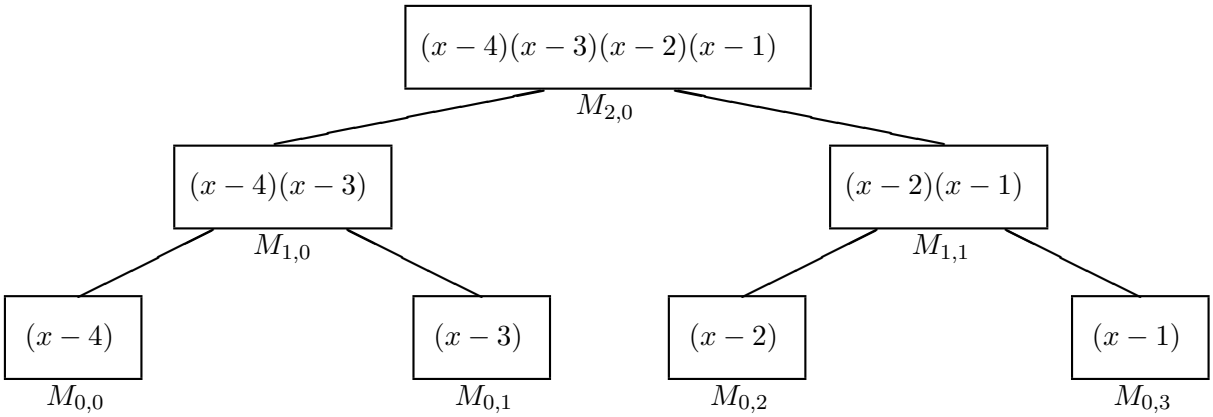

Figure 4.2: Example of Subproduct Tree when $n = 4$

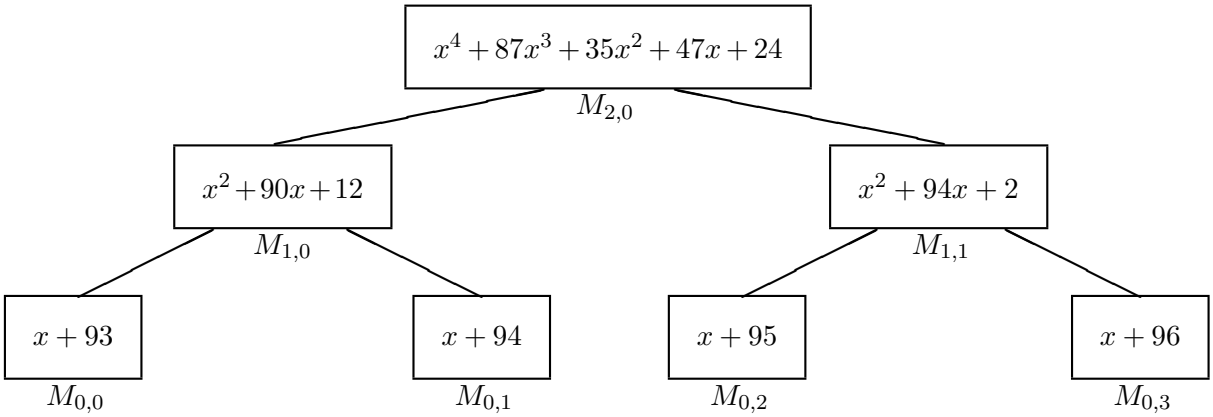

Figure 4.3: Example of Subproduct Tree in $\mathbb{F}_p$ when $n = 4$ and $p = 97$

Theorem 4.3. *Let $B(n)$ be number of arithmetic operations in F required to build up the subtree. Then,*

$$B(n) < \frac{1}{4}M(n)\log\, n + \frac{1}{4}M(n) + O(n) \in O(n\, \log^2\, n).$$

Hence, we can compute the subproduct tree in $O(n\, \log^2\, n)$ arithmetic operations in F.

4.1.2 Dividing Down the Subtree

With our subproduct tree in hand, we are now equipped to tackle fast multipoint evaluation with a simple divide and conquer algorithm based on the Chinese Remainder Theorem (CRT). Suppose that we have a polynomial $f(x) = a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1} \in F[x]$ and we want to evaluate f at the distinct points $u_0, ..., u_{n-1}$. Let $m_i = x - u_i$ for $0 \leq i < n$ as before. Since we specified that the evaluation points are distinct, this implies that m_i

Input : $n = 2^k$, and the evaluation points $u = [u_0, u_1, ..., u_{n-1}] \in \mathbb{F}_p^n$ for a prime p.
Output: The polynomials $M_{i,j}$ for $0 \leq i \leq k$ and $0 \leq j \leq 2^{k-i} - 1$.

```
for i from 0 to n − 1 do
    M_{0,i} ← (x − u_i)
end
for i from 1 to k do
    for j from 0 to 2^{k−i} − 1 do
        M_{i,j} ← M_{i−1,2j} × M_{i−1,2j+1}
    end
end
```

Algorithm 6: BUST

will be pairwise coprime for all i. Thus, the system of congruences: $f(x) \equiv f(u_i) \bmod m_i$, will have a solution by the CRT.

Let us take a moment to consider $f \bmod m_i$, where $m_i = x - u_i$. We know that when we mod a polynomial f by $(x - u_i)$ we are just taking the remainder after dividing f by m_i. We also know that this division results in unique polynomials q and r such that: $f = qm_i + r$, where $r = 0$ or degree$(r) <$ degree (m_i). Equivalently, we have:

$$f(u_i) = q(u_i)m_i(u_i) + r(u_i) = q(u_i)(u_i - u_i) + r(u_i) = q(u_i)0 + r(u_i) = r(u_i) = f \bmod m_i.$$

Therefore, $f \bmod (x - u_i) = f(u_i)$ for $0 \leq i < n$. We know that $f \bmod m_i \in F$ because it must have degree less than the degree of m_i and the m_i's all have degree one.

Hence, if we compute $f \bmod m_i$ for all i, then we will have evaluated f at n distinct points, as we wanted. However, an issue arises. Note that the polynomial f has degree $n-1$ and we are dividing it by a degree one polynomial. Each of these n divisions requires $O(n)$ multiplications in F. Therefore, this method of evaluation costs $O(n^2)$ arithmetic operations in F, which we do not want. The way to fix this is to use our precomputed subproduct tree. Instead of dividing f by the m_i's, we will **recurse down the tree**. Thus, we begin the algorithm by defining:

$$r_0 = f \bmod \prod_{i=0}^{n/2-1} m_i = f \bmod M_{k-1,0} \text{ and } r_1 = f \bmod \prod_{i=n/2}^{n-1} m_i = f \bmod M_{k-1,1}.$$

Pseudo code for dividing down the subtree is shown in Algorithm 7.

Input : $n = 2^k$, a polynomial $f \in \mathbb{F}_p[x]$ for a prime p, and the polynomials from the subtree $M_{i,j}$ for $0 \leq i \leq k$ and $0 \leq j \leq 2^{k-i} - 1$.
Output: $v = [f(u_0), ..., f(u_{n-1})] \in \mathbb{F}_p^n$.

```
1  if n = 1 then
2  |  return f
3  end
4  r_0 ← f mod M_{k-1,0}
5  r_1 ← f mod M_{k-1,1}
6  ML ← the subtree rooted at M_{k-1,0}
7  MR ← the subtree rooted at M_{k-1,1}
8  L ← DDST(n/2, r_0, ML, p) ∈ F_p^{n/2}
9  R ← DDST(n/2, r_1, MR, p) ∈ F_p^{n/2}
10 return [L, R]
```

Algorithm 7: DDST

To be explicit about what Algorithm 7 is doing: line 8 computes $r_0(u_0), ..., r_0(u_{n/2-1})$ and line 9 computes $r_1(u_{n/2}), ..., r_1(u_{n-1})$ and then, in the 10th line, we simply return $r_0(u_0), ..., r_0(u_{n/2-1})$, $r_1(u_{n/2}), ..., r_1(u_{n-1}) = f(u_0), ..., f(u_{n-1})$. Also note that lines 2 and 3 both require a fast division to improve on the $O(n^2)$ cost.

At this time we will provide an example of how we traverse down the subtree.

Example 1. Let $n = 4, p = 97, f(x) = 4 + 3x + 2x^2 + x^3$ and let the evaluation points be: $u_0 = 4, u_1 = 3, u_2 = 2$, and $u_3 = 1$. We start be computing f mod $M_{1,0}$ and f mod $M_{1,1}$ and we find the remainders: $r_0 = 54x + 90$ and $r_1 = 16x + 91$. Then we recursively call the algorithm twice; once with input $n/2 = 2, r_0$, and the subtree rooted at $M_{1,0}$ and the second with input $n/2 = 2, r_1$, and the subtree rooted at $M_{1,1}$. Now, we calculate:

$$r_0 = 54x + 90 \bmod (x - 4) = 15, \quad r_1 = 54x + 90 \bmod (x - 3) = 58$$
$$r_0 = 16x + 91 \bmod (x - 2) = 26, \quad r_1 = 16x + 91 \bmod (x - 1) = 10$$

The algorithm calls itself again four times. The first recursion has input $n/2 = 1, r_0 = 15$ and the subtree rooted at $M_{0,0}$. Notice that n is now one, which is the base of the recursion. Therefore, at this point, we return $f = r_0 = 15$. Similarly, we return f for the other three recursions and then we are done with the evaluation. Thus, we get: $f(u_0) = 15, f(u_1) = 58, f(u_2) = 26$, and $f(u_3) = 10$. This concludes the example.

Example 1 is showcased in Figure 4.4.

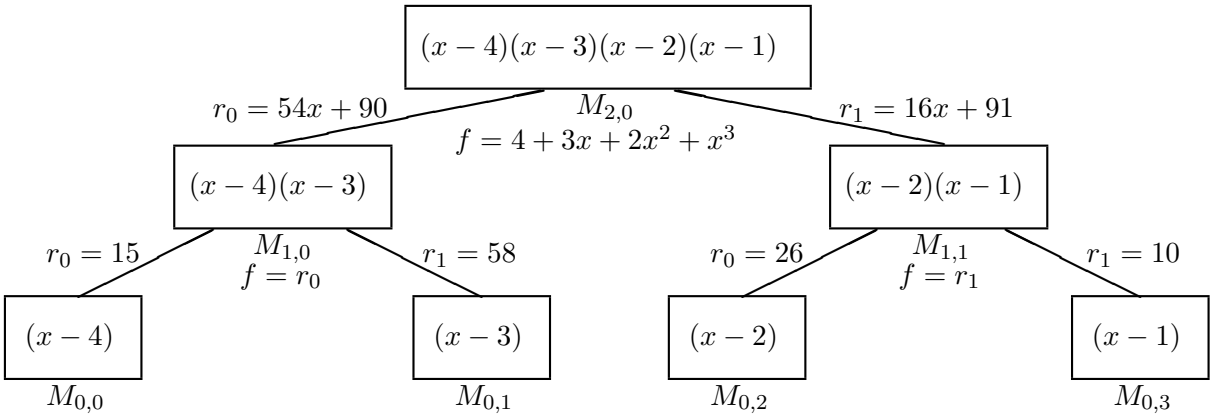

Figure 4.4: Example of DDST when $n = 4$ and $p = 97$

The correctness for Algorithm 7 can be proved by induction on $k = \log_2 n$. For the base case, we have $k = 0$, which implies f is a constant. Thus, DDST will return the correct answer, f, on line 2. For the induction hypothesis, we will assume that, for $k \geq 1$, lines 8 and 10 are correct. Suppose that when we divide f by $M_{k-1,0}$ and $M_{k-1,1}$, we get $f = q_0 M_{k-1,0} + r_0$ and $f = q_1 M_{k-1,0} + r_1$, respectively. Thus, evaluating f at u_i, gives: $f(u_i) = q_0(u_i)M_{k-1,0}(u_i) + r_0(u_i) = r_0(u_i) \in \mathbb{F}_p$ for $0 \leq i < n/2$ and $f(u_i) = q_1(u_i)M_{k-1,1}(u_i) + r_1(u_i) = r_1(u_i) \in \mathbb{F}_p$ for $n/2 \leq i < n$. This completes the proof.

We will now determine the runtime of DDST.

Let $C(n)$ be the number of arithmetic operations that Algorithm 7 does. To start, if $n = 1$, no operations are performed, and so $C(1) = 0$. Then, the algorithm has two recursive calls of size $n/2$ and each time there are two divisions of f by $M_{i,j}$ and $M_{i,j+1}$, both of which have degree $n/2$. That is, twice we divide a polynomial of degree $n - 1$ by a polynomial of degree $n/2$. Let $D(n/2)$ be the cost of one of those divisions. Therefore, the recurrence relation is:

$$C(n) = 2C(n/2) + 2D(n/2)$$

By Theorem 3.1, we let $D(n/2) = c \cdot (n \log_2 n)$ for some constant c, which enables us to use Lemma 4.2 to solve the above recurrence. In this situation, $a = 0, b = 2$, and $d = 0$, and we have:

$$\begin{aligned} C(n) &< 2 \cdot \frac{1}{4} D(n) \log n + 2 \cdot \frac{1}{4} D(n) + 0 \cdot (n \log n) + n \cdot 0 \\ &= \frac{1}{2} D(n) \log n + \frac{1}{2} D(n) \in O(n \log^2 n) \end{aligned}$$

The runtime of DDST is summarized in Theorem 4.4.

Theorem 4.4. *Let $C(n)$ be number of arithmetic operations in F needed to divide down the subtree. Then,*

$$C(n) < \frac{1}{2}D(n)\log\ n + \frac{1}{2}D(n) \in O(n\ \log^2\ n).$$

Thus, we can divide down the subproduct tree in $O(n\ \log^2\ n)$ operations in F.

4.2 Fast Multipoint Evaluation

To complete our fast multipoint evaluation algorithm, we simply need to combine the two procedures, BUST and DDST, into one. This is shown in Algorithm 8.

Input : $n = 2^k$, the polynomial $f \in \mathbb{F}_p[x]$, in which p is a prime, and
$u = [u_0, u_1, ..., u_{n-1}] \in \mathbb{F}_p^n$, where u is the set of evaluation points.
Output: $v = [f(u_0), f(u_1), ..., f(u_{n-1})] \in \mathbb{F}_p^n$.

```
1 T ← BUST(n, u, p)
2 v ← DDST(n, f, T, p)
3 return v
```

Algorithm 8: FastEval

Observe that FastEval does not use the polynomial $M_{k,0} = \prod_{i=0}^{n-1}(x - u_i)$. However, we will need it in the chapters for fast interpolation and fast Vandermonde. Hence, for our implementation of BUST, we built the entire subproduct tree, including $M_{k,0}$. If the reader wishes to have two versions of BUST− one for interpolation and Vandermonde that computes the whole subtree and another for evaluation that only computes the subtree up to $M_{k-1,0}$ and $M_{k-1,1}$− they are free to do so.

We will now determine the running time for fast evaluation.

Let $E(n)$ be the number of arithmetic operations performed by FastEval. The procedure only makes one call to BUST and one call to DDST. Thus, the equation for $E(n)$ is: $E(n) = B(n) + C(n)$, where $B(n)$ and $C(n)$ is the cost of BUST and DDST, respectively. Recall Theorem 2.5, 3.1, 4.3 and 4.4, then:

$$
\begin{aligned}
E(n) &= B(n) + C(n) \\
&< \frac{1}{4}M(n)\log\, n + \frac{1}{4}M(n) + O(n) + \frac{1}{2}D(n)\log\, n + \frac{1}{2}D(n) \\
&\leq \frac{1}{4}M(n)\log\, n + \frac{1}{4}M(n) + O(n) + \frac{1}{2}\cdot(4M(n) + O(n))\log\, n + \frac{1}{2}\cdot(4M(n) + O(n)) \\
&\leq \frac{9}{4}M(n)\log\, n + \frac{9}{4}M(n) + O(n\,\log\, n) + O(n) \\
&\leq \frac{9}{4}\cdot\frac{9}{2}(n\,\log\, n)\log\, n + \frac{9}{4}\cdot\frac{9}{2}(n\,\log\, n) + O(n\,\log\, n) + O(n) \\
&\leq \frac{81}{8}(n\,\log^2\, n) + \frac{81}{8}(n\,\log\, n) + O(n\,\log\, n) + O(n) \\
&< 11(n\,\log^2\, n) + O(n\,\log\, n) + O(n) \in O(n\,\log^2\, n)
\end{aligned}
$$

Therefore, we have proved Theorem 4.1 given at the beginning of this section and shown how to evaluate a degree $n-1$ polynomial at n distinct points in $O(n\,\log^2\, n)$ arithmetic operations in F.

Before we move on, we should address an assumption we made. Throughout this section, we assumed that the degree$(f) < n$, what happens if the degree$(f) \geq n$? Does FastEval still work? The answer is: yes, the algorithm still works, however there is an extra step required and an increase in cost. We would need to compute $r(x) = f(x) \bmod M_{k,0}$, where $M_{k,0}$ is the polynomial at the root of the subproduct tree. Then, degree$(r) < n$, or $r = 0$, and we are back in the scenario we can handle. Thus, we need one additional division of f by $M_{k,0}$.

4.2.1 FastEval Timings

We have implemented FastEval in Maple. Code is provided in Appendix A.

We will work over the field $\mathbb{F}_p$, where $p = 7 \cdot 2^{26} + 1$. Let $n = 2^k$ and $8 \leq k \leq 18$. We generate a vector, $u \in \mathbb{F}_p^n$ with $u_i \in [0, p)$ chosen at random and distinct. Then we randomly generate a polynomial of degree $n-1$. This will be used as input for FastEval as well as a quadratic version of evaluation which allows them to be compared. For the quadratic timings, we used the `Eval` command in Maple n times. The timings were run on an Intel Xeon E5 2660 CPU with 64 gigabytes of RAM.

The first column of Table 4.1 displays values of n ranging from 2^8 up to 2^{18}. The second and sixth columns present the time it took to execute the quadratic algorithm and FastEval, respectively, at that value of n. The fourth and fifth columns display the amount of time needed for BUST and DDST in FastEval. The third and seventh columns show, after n is doubled, the factor by which the computation time increases.

n	Quadratic	Growth Factor	BUST	DDST	FastEval	Growth Factor
2^8	5 ms	N/A	7 ms	70 ms	79 ms	N/A
2^9	19 ms	3.80	16 ms	119 ms	141 ms	1.78
2^{10}	79 ms	4.16	34 ms	221 ms	263 ms	1.87
2^{11}	295 ms	3.73	58 ms	454 ms	517 ms	1.97
2^{12}	1.09 s	3.70	109 ms	1.04 s	1.03 s	1.99
2^{13}	4.20 s	3.85	181 ms	2.11 s	2.30 s	1.98
2^{14}	17.57 s	4.18	363 ms	5.04 s	5.42 s	2.36
2^{15}	1.19 min	4.06	829 ms	10.27 s	11.11 s	2.05
2^{16}	4.96 min	4.17	1.58 s	24.16 s	25.76 s	2.32
2^{17}	20.88 min	4.21	3.35 s	49.90 s	53.32 s	2.07
2^{18}	1.42 hr	4.08	7.08 s	1.71 min	1.83 min	2.06

Table 4.1: The timings of FastEval in Maple

We can see that the quadratic algorithm's execution time increases approximately by a factor of four each time n is doubled, which implies that it is indeed a quadratic time algorithm.

For FastEval, as n doubles, the execution time increases on average by a factor of 2.1. FastEval became faster than the quadratic algorithm at $n = 2^{12}$. When $n = 2^{18}$, FastEval was 47 times faster than the quadratic algorithm.

Notice that the time needed for BUST to compute was small in comparison to the overall time needed for FastEval and that most of the computation time was in DDST. Recall that, in the recurrence relation for DDST, there is simply two recursive calls and two divisions. Therefore, if we wish to improve the timings for FastEval, we would have to somehow speed up the fast division algorithm.

Chapter 5

Fast Polynomial Interpolation

Given distinct $u_0, ..., u_{n-1} \in F$, and arbitrary $v_0, ..., v_{n-1} \in F$, there exists the unique polynomial $f(x) \in F[x]$ of degree less than n such that $f(u_i) = v_i$ for $0 \leq i < n$. Recall from the introduction that the requirement $f(u_i) = v_i$ yields a linear system of equations that can be expressed in matrix form with $Va = v$, where V is a Vandermonde matrix of order n and a is the unknown coefficient vector for $f(x)$.

The determinant of a Vandermonde matrix is given by

$$\det(V) = \prod_{0 \leq i < j < n} (u_j - u_i).$$

It is nonzero if and only if all u_i are distinct. In our case, the determinant of V is nonzero since we chose the u_i to be distinct, and thus $Va = v$ has a unique solution. Therefore, $f(x)$ exists and is unique.

The Lagrange interpolant, $L_i(x)$, is defined by

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{x - u_j}{u_i - u_j}. \tag{5.1}$$

Then, the property

$$L_i(u_j) = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j \end{cases}$$

follows immediately from (5.1). In Lagrange interpolation, the interpolating polynomial $f(x)$ takes the form

$$f(x) = \sum_{i=0}^{n-1} v_i L_i(x) = \sum_{i=0}^{n-1} v_i \prod_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{x - u_j}{u_i - u_j}. \tag{5.2}$$

Lagrange interpolation requires $7n^2 - 8n + 1 = O(n^2)$ operations in F [7]. In this section, we will show how to improve this to $O(n \log^2 n)$ operations. Let

$$M(x) = \prod_{j=0}^{n-1} x - u_j \quad \text{and} \quad t_i = \prod_{j \neq i} \frac{1}{u_i - u_j}.$$

Then, we can rearrange $L_i(x)$ as follows:

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{x - u_j}{u_i - u_j} = \prod_{j \neq i} \frac{M(x)}{(u_i - u_j)(x - u_i)} = \frac{t_i M(x)}{x - u_i}$$

Now, we can rewrite (5.2) like so:

$$f(x) = \sum_{i=0}^{n-1} v_i L_i(x) = \sum_{i=0}^{n-1} \frac{v_i t_i M(x)}{x - u_i}$$

The naive way to compute t_i would be to invert and multiply each pair of $u_i - u_j$ and divide the degree n polynomial $M(x)$ by the linear polynomial $(x - u_i)$, for a total cost of $O(n^2)$ arithmetic operations in F. Instead, we will take $M'(x) = \sum_{0 \leq j < n} M(x)/(x - u_i)$, which is the formal derivative of $M(x)$, and note that $M(x)/(x - u_i)$ vanishes at all points u_j with $i \neq j$. Thus,

$$M'(u_i) = \frac{M}{x - u_i}\Big|_{x=u_i} = \frac{1}{t_i}$$

Therefore, given $M(x)$, which is the largest polynomial at the root of our subproduct tree, we can compute all the t_i's with one evaluation of $M'(x)$ at n distinct evaluation points. By Theorem 4.1, we know that this costs $O(n \log^2 n)$ arithmetic operations in F. We also require $O(n)$ operations in F to compute $M'(x)$.

Let $n = 2^k$ and $c_i = v_i t_i$ for $0 \leq i < n$. Then, given the polynomials $M_{i,j}$ from the subtree, we can use a divide and conquer algorithm to recursively find:

$$r_0 = \sum_{i=0}^{n/2-1} \frac{c_i(M_{k-1,0})}{x - u_i} \quad \text{and} \quad r_1 = \sum_{i=n/2}^{n-1} \frac{c_i(M_{k-1,1})}{x - u_i}$$

Next, we obtain f using $f = M_{k-1,1} r_0 + M_{k-1,0} r_1$ and then we return f. Thus, we can implement fast interpolation with two algorithms. The first is the core of fast interpolation and outputs $f(x) = \sum_{0 \leq i < n} (c_i M)/(x - u_i)$. The second one pulls everything together by providing the input for the first algorithm. Pseudo code for these two algorithms is provided in Algorithm 9 and Algorithm 10.

Input : $n = 2^k, c = [v_0 t_0, v_1 t_1, ..., v_{n-1} t_{n-1}] \in \mathbb{F}_p^n$ for a prime p, and the polynomials $M_{i,j}$ for $0 \leq i \leq k$ and $0 \leq j \leq 2^{k-i} - 1$ from the subproduct tree.

Output: $f = \sum_{i=0}^{n-1} \frac{c_i M(x)}{x - u_i} \in \mathbb{F}_p[x]$ where $M = M_{k,0}$.

1 **if** $n = 1$ **then**
2 | **return** c_0
3 **end**
4 $c1 \leftarrow [c_0, ..., c_{n/2-1}]$
5 $c2 \leftarrow [c_{n/2}, ..., c_{n-1}]$
6 $r_0 \leftarrow$ Call InterpWork$(n/2, c1, M_{k-1,0}, p)$ to compute $\sum_{i=0}^{n/2-1} \frac{c_i(M_{k-1,0})}{x - u_i}$
7 $r_1 \leftarrow$ Call InterpWork$(n/2, c2, M_{k-1,1}, p)$ to compute $\sum_{i=n/2}^{n-1} \frac{c_i(M_{k-1,1})}{x - u_i}$
8 $f \leftarrow M_{k-1,1} \times r_0 + M_{k-1,0} \times r_1 \in \mathbb{F}_p[x]$
9 **return** f;

Algorithm 9: InterpWork

Input : $n = 2^k, v = [v_0, ..., v_{n-1}] \in \mathbb{F}_p^n$ for a prime p, and $u = [u_0, ..., u_{n-1}] \in \mathbb{F}_p^n$, where u_i is distinct for $0 \leq i < n$.

Output: The unique polynomial $f \in \mathbb{F}_p[x]$ of degree less than n such that $f(u_i) = v_i$ for all i.

1 $T \leftarrow$ Call BUST(n, u, p) to compute the subproduct tree.
2 $M \leftarrow M_{k,0}$
3 $s \leftarrow$ Call DDST(n, M', T, p) to evaluate $M'(x)$ at u_i for $0 \leq i < n$.
4 **for** i **from** *0* **to** $n - 1$ **do**
5 | $t \leftarrow s_i^{-1} \in \mathbb{F}_p$
6 | $c_i \leftarrow v_i \cdot t$
7 **end**
8 $f \leftarrow$ InterpWork(n, c, T, p)
9 **return** f

Algorithm 10: FastInterp

The correctness for Algorithm 9 can be proved by induction on $k = \log_2 n$. For the base case, if $k = 0$ then $M(x) = x - u_0$, which implies that $\sum_{0 \leq i < n}(c_i(x - u_i))/(x - u_i) = (c_0(x - u_0))/(x - u_0) = c_0$. Thus, on line 2, InterpWork will return the correct answer, c_0. For the induction hypothesis, if $k \geq 1$, we will assume that lines 6 and 7 are correct. Then, since $M(x) = M_{k-1,0} \times M_{k-1,1}$, Algorithm 9 will return the correct answer in line 9. This completes the proof.

We will now provide an example for FastInterp, by continuing Example 1.

Example 2. Suppose that $u = [4, 3, 2, 1]$, $v = [15, 58, 26, 10]$, $f(x) = 4 + 3x + 2x^2 + x^3$ and $p = 97$. First, we call BUST which computes the subproduct tree shown in Figure 4.3 and let $M(x) = M_{2,0}$. Then, we call DDST with input M' and the subtree, which outputs t_i^{-1} and we find that:

$$t = [6^{-1}, 95^{-1}, 2^{-1}, 91^{-1}] = [81, 48, 49, 16] \in \mathbb{F}_{97}^4$$
$$c = [v_0 t_0, v_1 t_1, v_2 t_2, v_3 t_3] = [51, 68, 13, 63] \in \mathbb{F}_{97}^4$$

Next, we call InterpWork and we compute:

$$\begin{aligned}
\sum_{0 \le i < 2} \frac{c_i(M_{k-1,0})}{x - u_i} &= \frac{c_0(M_{1,0})}{x - u_0} + \frac{c_1(M_{1,0})}{x - u_1} = \frac{51(x-4)(x-3)}{(x-4)} + \frac{68(x-4)(x-3)}{(x-3)} \\
&= 51(x-3) + 68(x-4) = 22x + 60 \in \mathbb{F}_{97} = r_0. \\
\sum_{2 \le i < 4} \frac{c_i(M_{k-1,1})}{x - u_i} &= \frac{c_2(M_{1,1})}{x - u_2} + \frac{c_3(M_{1,1})}{x - u_3} = \frac{13(x-2)(x-1)}{(x-2)} + \frac{63(x-2)(x-1)}{(x-1)} \\
&= 13(x-1) + 63(x-2) = 76x + 55 \in \mathbb{F}_{97} = r_1.
\end{aligned}$$

Finally, we multiply:

$$\begin{aligned}
M_{k-1,1} \times r_0 + M_{k-1,0} \times r_1 &= (x^2 + 94x + 2)(22x + 60) + (x^2 + 90x + 12)(76x + 55) \\
&= 4 + 3x + 2x^2 + x^3.
\end{aligned}$$

Thus, $f = 4 + 3x + 2x^2 + x^3 \in \mathbb{F}_{97}[x]$ and we have found the original polynomial. This concludes Example 2.

On a final note, recall that interpolating a polynomial of degree $n-1$ from its values at n points is equivalent to solving an n by n Vandermonde system. Hence, our example can be expressed in matrix form as:

$$\underbrace{\begin{bmatrix} 1 & 4 & 16 & 64 \\ 1 & 3 & 9 & 27 \\ 1 & 2 & 4 & 8 \\ 1 & 1 & 1 & 1 \end{bmatrix}}_{V} \underbrace{\begin{bmatrix} 4 \\ 3 \\ 2 \\ 1 \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} 15 \\ 58 \\ 26 \\ 10 \end{bmatrix}}_{v}$$

5.1 Fast Interpolation Complexity

Let $T(n)$ be the number of arithmetic operations done by InterpWork. The algorithm has two recursive calls of size $n/2$ and each time there are two multiplications of a degree $n/2$ polynomial by a polynomial of degree less than $n/2$. There is also an addition of two polynomials of degree not more than $n-1$. Therefore, the recurrence relation for $T(n)$ is:

$$T(n) \leq 2T(n/2) + 2M(n/2) + 1 \cdot n$$

To begin, if $n = 1$, we return c_0, and so $T(1) = 0$. Then, we let $M(n/2) = c \cdot (n \log_2 n)$ for some constant c, which enables us to use Lemma 4.2 to solve the above recurrence. In this situation, $a = 0$, $b = 2$, and $d = 1$, and we have:

$$\begin{aligned} T(n) &< 2 \cdot \frac{1}{4} M(n) \log n + 2 \cdot \frac{1}{4} M(n) + 1(n \log n) + n \cdot 0 \\ &= \frac{1}{2} M(n) \log n + \frac{1}{2} M(n) + O(n \log n) \in O(n \log^2 n) \end{aligned}$$

Hence, we can compute $f(x) = \sum_{0 \leq i < n} \frac{c_i M(x)}{x - u_i}$ in $O(n \log^2 n)$ operations in F.

Let $I(n)$ be the number of arithmetic operations in F performed by FastInterp. The algorithm makes one call to BUST, one call to DDST, and one call to InterpWork. We require n multiplications to compute M', n inverses for line 5, and n multiplications for line 6. Thus, the equation for $I(n)$ is: $I(n) = B(n) + C(n) + T(n) + 3n$, where $B(n)$, $C(n)$ and $T(n)$ are the cost of BUST, DDST, and InterpWork, respectively. Recall Theorem 2.5, 3.1, 4.3 and 4.4. Now:

$$\begin{aligned} I(n) &= B(n) + C(n) + T(n) + 3n \\ &< \frac{1}{4} M(n) \log n + \frac{1}{4} M(n) + \frac{1}{2} D(n) \log n + \frac{1}{2} D(n) + \frac{1}{2} M(n) \log n + \frac{1}{2} M(n) + O(n \log n) + O(n) \\ &\leq \frac{3}{4} M(n) \log n + \frac{3}{4} M(n) + \frac{1}{2} \cdot (4M(n) + O(n)) \log n + \frac{1}{2} \cdot (4M(n) + O(n)) + O(n \log n) + O(n) \\ &\leq \frac{11}{4} M(n) \log n + \frac{11}{4} M(n) + O(n \log n) + O(n) \\ &\leq \frac{11}{4} \cdot \frac{9}{2} (n \log n) \log n + \frac{11}{4} \cdot \frac{9}{2} (n \log n) + O(n \log n) + O(n) \\ &\leq \frac{99}{8} (n \log^2 n) + \frac{99}{8} (n \log n) + O(n \log n) + O(n) \\ &< 13(n \log^2 n) + O(n \log n) + O(n) \in O(n \log^2 n) \end{aligned}$$

Therefore, we have proved the following theorem.

Theorem 5.1. *Let $I(n)$ be number of arithmetic operations in F needed to interpolate a polynomial of degree $n-1$. Then,*

$$I(n) < 13(n \log^2 n) + O(n \log n) + O(n) \in O(n \log^2 n).$$

Thus, with $O(n \log^2 n)$ arithmetic operations in F, we can interpolate a degree $n-1$ polynomial from n distinct points.

5.2 FastInterp Timings

We have implemented FastInterp in Maple. Code is provided in Appendix A.

We will work over the field $\mathbb{F}_p$, where $p = 7 \cdot 2^{26} + 1$. Let $n = 2^k$ and $8 \leq k \leq 18$. We randomly generate two vectors, $v \in \mathbb{F}_p^n$ and $u \in \mathbb{F}_p^n$ with $v_i, u_i \in [0, p)$ and the u_i are distinct. This will be used as input for FastInterp as well as a quadratic version of interpolation which allows them to be compared. For the quadratic algorithm, the `Interp(...) mod p` command in Maple was used, which employs Newton interpolation. The timings were run on an Intel Xeon E5 2660 CPU with 64 gigabytes of RAM.

n	Quadratic	Growth Factor	BUST	InterpWork	DDST	FastInterp	Growth Factor
2^8	2 ms	N/A	7 ms	8 ms	61 ms	85 ms	N/A
2^9	6 ms	3.00	15 ms	15 ms	120 ms	152 ms	1.79
2^{10}	21 ms	3.50	28 ms	29 ms	212 ms	276 ms	1.82
2^{11}	73 ms	3.48	49 ms	53 ms	446 ms	557 ms	2.02
2^{12}	277 ms	3.79	91 ms	102 ms	1.06 s	1.31 s	2.35
2^{13}	1.09 s	3.94	169 ms	207 ms	2.50 s	2.92 s	2.23
2^{14}	4.32 s	3.96	360 ms	433 ms	4.62 s	5.50 s	1.88
2^{15}	17.50 s	4.05	832 ms	856 ms	9.97 s	11.94 s	2.17
2^{16}	1.19 min	4.08	1.57 s	2.19 s	22.12 s	26.18 s	2.19
2^{17}	5.15 min	4.32	3.58 s	4.61 s	48.65 s	57.86 s	2.21
2^{18}	21.18 min	4.11	7.52 s	9.34 s	1.63 min	1.98 min	2.05

Table 5.1: The timings of FastInterp in Maple

The first column of Table 5.1 displays values of n ranging from 2^8 up to 2^{18}. The second and seventh columns present the time it took to execute the quadratic algorithm and FastInterp, respectively, at that value of n. The fourth, fifth, and sixth columns display

the amount of time needed for BUST, InterpWork, and DDST in FastInterp. The third and eighth columns show, after n is doubled, the factor by which the computation time increases.

We can see that the quadratic algorithm's execution time increases approximately by a factor of four each time n is doubled, which implies that it is indeed a quadratic time algorithm.

For FastInterp, as n doubles, the execution time increases on average by a factor of 2.1. The n at which FastInterp became faster than the quadratic algorithm was 2^{15}. When $n = 2^{18}$, the quadratic algorithm took eleven times as long to compute as FastInterp.

Notice that the time computation time for BUST and InterpWork was small in comparison to the overall time needed for FastInterp. Once again, most of the computation time was in DDST.

Observe that, at small n, Maple's `Interp(...) mod` p command performs rather well. At $n = 2^8$, the quadratic algorithm was 43 times faster than FastInterp. Thus, it would be a good idea to use the `Interp(...) mod` p command until around $n = 2^{15}$ and then switch to FastInterp.

Chapter 6

Solving Transposed Vandermonde Systems

Let $f(x) = a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1} \in F[x]$ and $f(u_i) = v_i$ for $0 \leq i < n$. This section will present a fast algorithm that solves a transposed Vandermonde linear system of equations, $V^T a = v$, for the unknown coefficient vector a of the polynomial f. We can express this in matrix form as follows

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ u_0 & u_1 & u_2 & \cdots & u_{n-1} \\ u_0^2 & u_1^2 & u_2^2 & \cdots & u_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_0^{n-1} & u_1^{n-1} & u_2^{n-1} & \cdots & u_{n-1}^{n-1} \end{bmatrix}}_{V^T} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}}_{a} = \underbrace{\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-1} \end{bmatrix}}_{v}.$$

One application of transposed Vandermonde systems is that they need to be solved in sparse interpolation algorithms, like in [1]. In 1990, Zippel [16] prensented an $O(n^2)$ algorithm to solve a transposed Vandermonde system. Our goal in this section will be to prove Theorem 6.1.

Theorem 6.1. *Let $V(n)$ be number of arithmetic operations in F required to solve a transposed Vandermonde system. Then,*

$$V(n) < 20(n \log^2 n) + O(n \log n) + O(n) \in O(n \log^2 n).$$

First, let $(V^T)^T = V$, where V is a Vandermonde matrix of order n. Now, we have:

$$V^T a = v \Rightarrow (V^{-1})^T V^T a = (V^{-1})^T v \Rightarrow Ia = (V^{-1})^T v \Rightarrow a = (V^{-1})^T v.$$

where I is the identity matrix of order n. Let $C_j(x) = c_{0,j} + c_{1,j}x + ... + c_{n-1,j}x^{n-1}$, where the coefficients are the jth column $c_j = (c_{0,j}, c_{1,j}, ..., c_{n-1,j})^T$ of V^{-1}. Then, C_j evaluated at u_i has the same value as the (i,j)th element of $VV^{-1} = I$. That is,

$$C_j(u_i) = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j \end{cases}.$$

Recall that this is the property of the Lagrange interpolant. Thus,

$$C_j(x) = \prod_{\substack{k=0 \\ k \neq j}}^{n-1} \frac{x - u_k}{u_j - u_k}, \text{ for } 0 \leq j < n$$

and

$$a = (V^{-1})^T v = \begin{bmatrix} c_{0,0}v_0 + c_{1,0}v_1 + \cdots + c_{n-1,0}v_{n-1} \\ c_{0,1}v_0 + c_{1,1}v_1 + \cdots + c_{n-1,1}v_{n-1} \\ \vdots \\ c_{0,n-1}v_0 + c_{1,n-1}v_1 + \cdots + c_{n-1,n-1}v_{n-1} \end{bmatrix}. \tag{6.1}$$

Consider the polynomials:

$$D(x) = v_{n-1}x + v_{n-2}x^2 + ... + v_1x^{n-1} + v_0x^n$$
$$C_j(x)D(x) = G(x) = g_0x + g_1x^2 + ... + g_{n-2}x^{n-1} + g_{n-1}x^n + ... + g_{2n-2}x^{2n-1}.$$

Since $C_j(x)D(x) = G(x)$, this implies that $g_{n-1} = c_{0,j}v_0 + c_{1,j}v_1 + ... + c_{n-1,j}v_{n-1}$. Thus, g_{n-1} has the same form as a_j in (6.1), and so a_j is the coefficient of x^n in $G(x)$.

As in the previous section, let

$$M(x) = \prod_{j=0}^{n-1} x - u_j \quad \text{and} \quad t_j = \prod_{j \neq i} \frac{1}{u_j - u_i}.$$

Then, we can compute a_j quickly without computing all of $G(x)$ by noticing that

$$C_j(x) = \frac{t_j M(x)}{x - u_j}.$$

Suppose that

$$H(x) = M(x)D(x) = h_0x^1 + h_1x^2 + h_2x^3 + ... + h_{2n-1}x^{2n-2} + h_{2n-1}x^{2n}.$$

The coefficient of x^n in the quotient of $H(x)/(x - z)$, can be found with

$$Q_n(z) = h_n + h_{n+1}z + ... + h_{2n-2}z^{n-2} + h_{2n-1}z^{n-1}.$$

Now, we have that $Q_n(u_j) = a_j t_j^{-1}$ for all j. And so, at this point, we have a degree $n-1$ polynomial, $Q_n(x)$, that must be evaluated at n evaluation points, namely $u_0, ..., u_{n-1}$. By Theorem 4.1, this costs $O(n \log^2 n)$ arithmetic operations in F.

We find t_j in the same way as the interpolation section, by taking the derivative, $M'(x) = \sum_{j=0}^{n-1} M(x)/(x-u_j)$, and realizing that $M/(x-u_i)$ vanishes at all points u_j with $i \neq j$, and so $M(u_j)$ computes t_j^{-1}. Therefore, if we evaluate $M'(x)$ at the points $u_0, ..., u_{n-1}$, then all of the t_j's can be obtained.

A summary of FastVandermonde follows. We input the vector $u = [u_0, ..., u_{n-1}]$ whose entries make up the transposed Vandermonde matrix V^T and the vector $v = [v_0, ..., v_{n-1}]$. First, compute the polynomial $M(x)$ using Algorithm 6: BUST. Let $H(x)$ be defined as above. We calculate $H(x)$ using fast multiplication and then build the polynomial $Q_n(x)$ using the coefficients of $H(x)$. Next, obtain $r_i = a_i t_i^{-1}$ and t_i^{-1} by evaluating $Q_n(x)$ and $M'(x)$ at the entries of u. Lastly, we return the coefficients $a_i = r_i \cdot t_i$ for $0 \leq i < n$. Pseudo code for FastVandermonde is included in Algorithm 11.

Input : $n = 2^k, v = [v_0, ..., v_{n-1}] \in \mathbb{F}_p^n$ for a prime p, and $u = [u_0, ..., u_{n-1}] \in \mathbb{F}_p^n$, whose entries make up the transposed Vandermonde matrix V^T.

Output: The coefficient vector $a = [a_0, ..., a_{n-1}] \in \mathbb{F}_p^n$, where $V^T a = v$.

1 $T \leftarrow$ Call BUST(n, u, p) to compute the subproduct tree.
2 $M \leftarrow M_{k,0}$
3 $D \leftarrow v_0 x^n + v_1 x^{n-1} + ... + v_{n-2} x^2 + v_{n-1} x$
4 Compute $H = M \times D$, where
  $H = h_0 x^1 + h_1 x^2 + h_2 x^3 + ... + h_{2n-1} x^{2n-2} + h_{2n-1} x^{2n}$.
5 Read off the coefficients of H to get $Q = h_n + h_{n+1} z + ... + h_{2n-2} z^{n-2} + h_{2n-1} z^{n-1}$.
6 $r \leftarrow$ Call DDST(n, Q, T, p) to evaluate Q at u_i for $0 \leq i < n$.
7 $s \leftarrow$ Call DDST(n, M', T, p) to evaluate M' at u_i for $0 \leq i < n$.
8 **for** i **from** *0* **to** $n-1$ **do**
9 $t \leftarrow s_i^{-1} \in \mathbb{F}_p$
10 $a_i \leftarrow r_i \cdot t$
11 **end**
12 **return** a

Algorithm 11: FastVandermonde

Now, we will present an example for FastVandermonde, continuing Example 1 and 2.

Example 3. Let $u = [4, 3, 2, 1]$, $v = [15, 58, 26, 10]$, and $p = 97$. First, we compute the subproduct tree shown in Figure 4.3 by calling BUST and then we assign $M(x) = M_{2,0} = x^4 + 87x^3 + 35x^2 + 47x + 24$. Next, we compute the polynomials:

$$D(x) = 10x + 26x^2 + 58x^3 + 15x^4$$
$$H(x) = 46x + 27x^2 + 54x^3 + 16x^4 + 60x^5 + 68x^6 + 5x^7 + 15x^8$$
$$Q_n(x) = 60 + 68x + 5x^2 + 15x^3$$

Now, we call DDST once with $Q_n(x)$ and again with M' and then we find:

$$r = [14, 35, 45, 51] \in \mathbb{F}_{97}^4$$
$$t = [6^{-1}, 95^{-1}, 2^{-1}, 91^{-1}] = [81, 48, 49, 16] \in \mathbb{F}_{97}^4$$
$$a = [r_0 t_0, r_1 t_1, r_2 r_2, r_3 t_3] = [67, 31, 71, 40] \in \mathbb{F}_{97}^4$$

Therefore, we have solved the transposed Vandermonde system and found the coefficient vector $a = [67, 31, 71, 40] \in \mathbb{F}_{97}^4$. This concludes the example.

In matrix form, Example 3 is expressed as:

$$\underset{V^T}{\begin{bmatrix} 1 & 1 & 1 & 1 \\ 4 & 3 & 2 & 1 \\ 16 & 9 & 4 & 1 \\ 64 & 27 & 8 & 1 \end{bmatrix}} \underset{a}{\begin{bmatrix} 67 \\ 31 \\ 71 \\ 40 \end{bmatrix}} = \underset{v}{\begin{bmatrix} 15 \\ 58 \\ 26 \\ 10 \end{bmatrix}}$$

6.1 FastVandermonde Complexity

Let $V(n)$ be the number of arithmetic operations in F that FastVandermonde executes. Algorithm 11 makes one call to BUST, two calls to DDST, and performs one FFT multiplication of two degree n polynomials. We need n multiplications each to compute $r_i \cdot t$ and the derivative M' as well as n inverses in line 9. Therefore, the formula for $V(n)$ is: $V(n) = B(n) + 2C(n) + M(n) + 3n$, where $B(n)$ and $C(n)$ are the cost of BUST, and DDST, respectively. Recall Theorem 2.5, 3.1, 4.3 and 4.4. Then,

$$\begin{aligned}
V(n) &= B(n) + 2C(n) + M(n) + 3n \\
&< \frac{1}{4}M(n)\log n + \frac{1}{4}M(n) + O(n) + D(n)\log n + D(n) + M(n) + 3n \\
&\leq \frac{1}{4}M(n)\log n + \frac{5}{4}M(n) + O(n) + (4M(n) + O(n))\log n + 4M(n) + O(n) + 3n \\
&\leq \frac{17}{4}M(n)\log n + \frac{21}{4}M(n) + O(n \log n) + O(n) \\
&\leq \frac{17}{4} \cdot \frac{9}{2}(n \log n)\log n + \frac{21}{4} \cdot \frac{9}{2}(n \log n) + O(n \log n) + O(n) \\
&\leq \frac{153}{8}(n \log^2 n) + \frac{189}{8}(n \log n) + O(n \log n) + O(n) \\
&< 20(n \log^2 n) + O(n \log n) + O(n) \in O(n \log^2 n)
\end{aligned}$$

Thus, we have proved Theorem 6.1 and we are now able to solve a transposed Vandermonde system with $O(n \log^2 n)$ arithmetic operations in F.

6.2 FastVandermonde Timings

We have implemented FastVandermonde in Maple. Code is provided in Appendix A.

We will work over the field $\mathbb{F}_p$, where $p = 7 \cdot 2^{26} + 1$. Let $n = 2^k$ and $8 \leq k \leq 18$. We randomly generate two vectors, $v \in \mathbb{F}_p^n$ and $u \in \mathbb{F}_p^n$ with $v_i, u_i \in [0, p)$ and the u_i are distinct. This will be used as input for FastVandermonde as well as a quadratic version of solving transposed Vandermonde systems to allow for comparisons. For the quadratic timings, we coded Zippel's algorithm [16]. The timings were run on an Intel Xeon E5 2660 CPU with 64 gigabytes of RAM.

n	Quadratic	Growth Factor	BUST	DDST 1	DDST 2	FastVandermonde	Growth Factor
2^8	16 ms	N/A	7 ms	66 ms	54 ms	142 ms	N/A
2^9	65 ms	4.06	16 ms	117 ms	102 ms	248 ms	1.75
2^{10}	276 ms	4.25	31 ms	241 ms	224 ms	515 ms	2.08
2^{11}	1.14 s	4.13	62 ms	529 ms	451 ms	1.16 s	2.25
2^{12}	4.78 s	4.19	108 ms	1.01 s	933 ms	2.40 s	2.07
2^{13}	18.17 s	3.80	184 ms	2.31 s	2.09 s	4.86 s	2.02
2^{14}	1.32 min	4.34	382 ms	4.65 s	4.48 s	9.86 s	2.03
2^{15}	5.16 min	3.92	901 ms	9.62 s	9.47 s	22.17 s	2.25
2^{16}	20.40 min	3.95	2.07 s	20.53 s	20.59 s	47.23 s	2.13
2^{17}	1.36 hr	4.01	4.60 s	46.12 s	46.23 s	1.66 min	2.11
2^{18}	5.55 hr	4.08	9.29 s	1.59 min	1.58 min	3.47 min	2.09

Table 6.1: The timings of FastVandermonde in Maple

The first column of Table 6.1 displays values of n ranging from 2^8 up to 2^{18}. The second and seventh columns present the time it took to execute the quadratic algorithm and FastVandermonde at that value of n. The third and eighth columns show, after n is doubled, the factor by which the computation time increases. The fourth column displays the execution time for BUST in FastVandermonde. The fifth and sixth columns show the amount of time needed for the first and second execution of DDST in FastVandermonde, respectively.

For the quadratic algorithm, we can see that the time it takes for that algorithm to run increases approximately by a factor of four each time n is doubled, which implies that it is indeed a quadratic time algorithm.

For FastVandermonde, the execution time increases on average by a factor of 2.1, as n is doubled. FastVandermonde overtook the quadratic algorithm in execution speed when $n = 2^{12}$. When $n = 2^{18}$, FastVandermonde was 96 times faster than the quadratic algorithm.

Notice again that the time needed for BUST to compute was small in comparison to the overall time needed for FastVandermonde. Most of the computation time was in the two executions of DDST.

Chapter 7

Conclusion

In this project, we were able to implement three algorithms: FastEval, FastInterp, and FastVandermonde, each of which requires $O(n \log^2 n)$ arithmetic operations in F. This is a nice upgrade from their classical quadratic running times. All three algorithms make use of the subproduct tree and assume a fast multiplication algorithm for $\mathbb{F}_p[x]$. The timings we found during our experiments reflect the complexities we expected the algorithms to run.

A natural extension of this project would be to code these algorithms in C and compare the timings to Maple. As seen in Bluestein's section, there will most likely be a significant improvement in the timings. Another extension of this project would be implementing the fast Chinese remaindering algorithm detailed in [7]. It also utilizes the subproduct tree. However, the reader should be aware that this algorithm also requires a fast Euclidean algorithm.

Bibliography

[1] M. Ben-Or and P. Tiwari. A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proc. of STOC '20*, ACM Press, 301–309, 1988.

[2] L.I. Bluestein. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics*. 451-455, 1970.

[3] A. Borodin and R. Moenck, Fast Modular Transforms. *Journal of Computer and System Sciences 8*, 366–386, 1974.

[4] J. W. Cooley and J. W. Tukey, An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation* 297–301, 1965.

[5] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms* The MIT Press, 2009.

[6] C. Fiduccia, Polynomial evaluation via the division algorithm: The fast Fourier transform revisited. In *Proc. 4th Symp. on Theory of Computing*, ACM Press, 88–93, 1972.

[7] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 291–297, 2013.

[8] J. Gauthier. Fast Multipoint Evaluation on n Arbitrary Points. (Master's Project) *Simon Fraser University*, 2017.

[9] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra* Springer US, 2007.

[10] E. Horowitz, A fast method for interpolation of polynomials using preconditioning. *Information Processing Letters 1*, 157–163, 1972.

[11] E. Kaltofen and L. Yagati. Improved Sparse Multivariate Polynomial Interpolation Algorithms. *International Symposium on Symbolic and Algebraic Computation*, 467–474, 1988.

[12] M. Law, M. Monagan, A parallel implementation for polynomial multiplication modulo a prime. In *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation* 78-86, 2015

[13] J. Lipson, Chinese remainder and interpolation algorithms. In *Proc. of the 2nd Syrup. on Symbolic and Algebraic Manipulation*, ACM Press, 372–391,1971.

[14] Maple 2018. Maplesoft, a division of Waterloo Maple Inc., Waterloo, Ontario. `https://www.maplesoft.com/`

[15] R. Moenck and A. Borodin, Fast modular transforms via division. In *Proc. of the 13th Annual Syrup. on Switching and Automata Theory*, Oct. 1972.

[16] R.E. Zippel. Interpolating polynomials from their values. *J. Symbolic Comput.*, **9**(3):375–403, 1990.

Chapter 8

Appendix A: Maple Code

```
FastFT:= proc(n,A,w,p) #Fast Fourier Transform of size n.
local n2,b,c,B,C,i,wi,T,D;
  if n=1 then return A; fi;
  D:=Array(1..n);
  if n=2 then D[1] := A[1]+A[2] mod p; D[2] := A[1]-A[2] mod p; return D; fi;
  n2:=n/2;
  b:=Array(1..n2); c:=Array(1..n2);
   for i from 0 to n2-1 do
     b[i+1]:=A[2*i+1];
     c[i+1]:=A[2*i+2];
   od;
  B:=FastFT(n2,b,w^2 mod p,p);
  C:=FastFT(n2,c,w^2 mod p,p);
  wi:= 1;
   for i from 0 to n2-1 do
     T:=wi*C[i+1] mod p;
     D[i+1]:=B[i+1]+T mod p;
     D[n2+i+1]:=B[i+1]-T mod p;
     wi:=wi*w mod p;
   od;
  return D;
end:

poweroftwo:=proc(la,lb) #The next power of two > degree(a) + degree(b) + 1.
  local c,d,m;
  m:= (la-1)+(lb-1)+1; d:= 2;
    while (d < m) do  d:= 2*d; od;
  return d;
end:
```

```
FFTMult:= proc(a,la,b,lb,omega,p)  #FFT Multiplication.
  local A,B,Afft,Bfft,Cfft,C,da,db,n,i,nv,w;
  if la <> lb or type(log[2](la),integer)=false then
    n:= poweroftwo(la,lb);
    A:= Array(1..n);
    B:= Array(1..n);
    for i from 1 to la do A[i] := a[i]; od;
    for i from 1 to lb do B[i] := b[i]; od;
    w:=nthroot(n,p);
  else
    n:=la;
    A:= a;
    B:= b;
    w:=omega;
  fi;
  Cfft:= Array(1..n);
  Afft:= FastFT(n,A,w,p);
  Bfft:= FastFT(n,B,w,p);
    for i from 1 to n do
      Cfft[i] := Afft[i]*Bfft[i] mod p;
    od;
  C:= FastFT(n,Cfft,1/w mod p,p);
  nv:=1/n mod p;
    for i from 1 to n do
      C[i]:=nv*C[i] mod p;
    od;
  return C;
end:


NextPOT:=proc(n) #The next power of two >= 2n-1.
  local m;
  m := 2; while m<2*n-1 do m := 2*m; od;
  return m;
end:
```

```
Bluestein:=proc(a,n,w,p)  #Bluestein's algorithm
local W,i,t,b,s,r,S,R,c,X,m,w1;
  W:=Array(1..n);
  W[1]:=1; W[2]:=w;
    for i from 3 to n do
      W[i]:=W[i-1]*w mod p;
    od;
  b:=Array(1..n); s:=Array(1..n);
  b[1]:=W[1]; b[2]:=W[2];
  s[1]:=W[1]; s[2]:=w^(-1) mod p;
    for i from 3 to n do
      t:=(i-1)^2 mod n;
      b[i]:=W[t+1];
      t:=-t mod n;
      s[i]:=W[t+1];
    od;
  r:=Array(1..n);
    for i from 1 to n do
      r[i]:=a[i]*b[i] mod p;
    od;
  m:=NextPOT(n); w1:=nthroot(m,p);
  R:=Array(1..m):
    for i from 1 to n do
      R[i]:=r[i];
    od:
  S:=Array(1..m):
    for i from 1 to n do
      S[i]:=s[i];
    od:
    for i from m-n+2 to m do
      S[i]:=s[m-i+2]
    od:
  c:=FFTMult(R,m,S,m,w1,p);
  X:=Array(1..n);
    for i from 1 to n do
      X[i]:=b[i]*c[i] mod p;
    od;
  return X;
end:
```

```
FastNewton:=proc(g,x,n,p) #Fast Newton inversion.
local m,y,a,b,i,c,z,t,y1;
  if n=1 then return 1/g mod p; fi;
  m:=ceil(n/2);
  a:=convert(taylor(g,x,m),polynom) mod p;
  y:=FastNewton(a,x,m,p);
  z:=Expand((1-g*y)/x^m) mod p;
  z:=Expand(y*z) mod p;
  z:=convert(taylor(z,x,n-m),polynom) mod p;
  y:=y+Expand(x^m*z) mod p;
return y;
end:


RecipPoly:=proc(f,n,p) #Computes the reciprocal polynomial.
local res;
  res:=Expand(x^n*subs(x=1/x,f)) mod p:
return(res);
end:


FastRem:=proc(f,g,x,p) #Fast Division algorithm that only returns the remainder.
local n,m,q,a,fr,b,c,r;
  n:=degree(f);
  m:=degree(g);
  if n < m then return f; fi;
  a:=RecipPoly(g,m,p);
  a:=convert(taylor(a,x,n-m+1),polynom) mod p;
  b:=FastNewton(a,x,n-m+1,p);
  fr:=RecipPoly(f,n,p);
  c:=Expand(fr*b) mod p;
  c:=convert(taylor(c,x,n-m+1),polynom) mod p;
  q:=RecipPoly(c,degree(c),p);
  r:=f-Expand(g*q) mod p;
return r;
end:
```

```
BUST:=proc(n,u,x,p)  #Building Up the Subproduct Tree
local L,R,f;
   if n = 1 then return [x-u[1] mod p] fi;
   L := BUST(n/2,u[1..n/2],x,p);
   R := BUST(n/2,u[n/2+1..n],x,p);
   f := Expand(L[1]*R[1]) mod p;
   [f,L,R];
end:


DDST:=proc(n,f,M,p) #Dividing Down The Subproduct Tree
local r0,r1;
  if n = 1 then return f; fi;
  r0:=FastRem(f,M[2][1],x,p);
  r1:=FastRem(f,M[3][1],x,p);
  [op(DDST(n/2,r0,M[2],p)), op(DDST(n/2,r1,M[3],p))];
end:


FastEval:=proc(n,f,u,p) #Fast Multipoint Evaluation.
local T,v;
  T:=BUST(n,u,x,p);
  v:=DDST(n,f,T,p);
return v;
end:


InterpWork:=proc(n,c,M,p)
local q,r0,r1,q0,q1,i,f,k;
  if n = 1 then return c[1]; fi;
  r0:=InterpWork(n/2,c[1..n/2],M[2],p);
  r1:=InterpWork(n/2,c[n/2+1..n],M[3],p);
  f:=Expand(r0*M[3][1]+r1*M[2][1]) mod p;
return f;
end:
```

```
FastInterp:=proc(n,v,u,p) #Fast Interpolation.
local T,M,s,i,c,f,t;
  T:=BUST(n,u,x,p);
  M:=T[1];
  s:=DDST(n,diff(M,x),T,p);
  c:=Array(1..n);
    for i from 1 to n do
      t:=1/s[i] mod p;
      c[i]:=Expand(v[i]*t) mod p;
    od;
  f:=InterpWork(n,convert(c,list),T,p);
return f;
end:


FastVandermonde:=proc(n,u,v,p)
#Fast method for solving a transposed Vandermonde system.
local T,M,D,MD,H,Q,s,a,t,i,r;
  T:=BUST(n,u,x,p);
  M:=T[1];
  D:=add(v[i]*x^(n-i+1),i=1..n) mod p;
  MD:=Expand(M*D) mod p;
  H:=[coeffs(MD)];
  Q:=add(H[i]*x^(n-i),i=1..n);
  r:=DDST(n,Q,T,p);
  s:=DDST(n,diff(M,x),T,p);
  a:=Array(1..n);
    for i from 1 to n do
      t:=1/s[i] mod p;
      a[i]:=Expand(r[i]*t) mod p;
    od;
  return convert(a,list);
end:
```

```
SlowEval:=proc(f,u,n,p) #O(n^2) method for multipont evaluation.
local i,R;
R:=Array(1..n);
 for i from 1 to n do
  R[i]:=Eval(f,x=u[i]) mod p;
od;
return R;
end:


Interp(u,v,x) mod p; #O(n^2) method for interpolation.


VandermondeSolveMODP1:=proc(v,u,p)
#O(n^2) method for solving a transposed Vandermonde system.
local t,i,j,M,x,a,q,r,s,Q,temp;
  t:=numelems(v);
  M:=modp1(ConvertIn(1,x),p);
    for i to t do
      temp:=modp1(ConvertIn(x-m[i],x),p);
      M:=modp1(:-Multiply(temp,M),p);
    od;
  a:=Vector(t);
    for j to t do
      Q:=modp1(Quo(M,ConvertIn(x-m[j],x)),p);
      r:=1/modp1(Eval(Q,m[j]),p) mod p;
      Q:=modp1(ConvertOut(Q),p); #dense list#
      s:=add(v[i]*Q[i],i=1..t) mod p;
      a[j]:=r*s mod p;
    od;
  a:=convert(a,list);
return a;
end:
```

Chapter 9

Appendix B: C Code

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

MUL64s computes $a \cdot b \in \mathbb{F}_p$.

```
LONG MUL64s(LONG a, LONG b, LONG p);
```

INV64s computes $c^{-1} \in \mathbb{F}_p$.

```
LONG INV64s( LONG c, LONG p);
```

getomega64s returns a primitive n root of unity in $\mathbb{F}_p$.

```
LONG getomega64s( LONG p, LONG n);
```

MultFFT64s computes $A = A \times B \in \mathbb{F}_p$ using an FFT where A, B, and W are arrays of size m, where $m = 2^k$, and w is a primitive mth root of unity.

```
void MultFFT64s( LONG *A, LONG *B, LONG m, LONG w, LONG *W, LONG p);
```

```
void ExtendBlueR(LONG *r, LONG n, LONG *R, LONG m)
{//Extend r of length n to R of length m, where m >= 2n-1 and m = 2^k.
   LONG i;
      for(i=0; i<n; i++) R[i] = r[i];
      for(i=n; i<m; i++) R[i] = 0;
   return;
}


void ExtendBlueS(LONG *s, LONG n, LONG *S, LONG m)
{//Extend s of length n to S of length m, where m >= 2n-1 and m = 2^k.
   LONG i;
      for(i=0; i<n; i++) S[i] = s[i];
      for(i=m-n+1; i<=m-1; i++) S[i] = s[m-i];
      for(i=n; i<=m-n; i++) S[i] = 0;
   return;
}


LONG NextPow2(n)
{//Returns the next power of 2 that is >= 2n-1.
    LONG t, m;
    t = 2*n-1;
       for(m=2; m<t; m = 2*m);
    return m;
}
```

```
void Bluestein(LONG *a, LONG n, LONG w, LONG *T, LONG p)
{  //a is the coefficient vector of length n,
   //w is a primitive nth root of unity,
   //T is an array of size 4n.
    LONG i,t,m,w1;
    LONG *b, *s, *r, *W, *R, *S, *W1;
       if(n==1) return;
    b = T; s = T+n; r = s+n; W = r+n;
    W[0] = 1;
    W[1] = w;
       for( i=2; i<n; i++ ) W[i] = MUL64s(w,W[i-1],p);
    b[0] = W[0];
    b[1] = W[1];
    s[0] = W[0];
    s[1] = INV64s(w,p);
       for(i=2; i<n; i++)
          { t = MUL64s(i,i,n);
            b[i] = W[t];
            t = MUL64s(n-1,t,n);
            s[i] = W[t];
          }
       for(i=0; i<n; i++) r[i] = MUL64s(a[i],b[i],p);
    m = NextPow2(n);
    w1 = getomega64s(p,m);
    R = array64s(m);
    ExtendBlueR(r,n,R,m);
    S = array64s(m);
    ExtendBlueS(s,n,S,m);
    W1 = array64s(m);
    MultFFT64s(R,S,m,w1,W1,p);
      for(i=0; i<n; i++) a[i] = MUL64s(b[i],R[i],p);
    free(R);
    free(S);
    free(W1);
    return;
}
```