Fast Multipoint Evaluation On n Arbitrary Points

by

Justine Gauthier

B.Sc., University of King's College, 2015

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

in the
Department of Mathematics
Faculty of Science

Approval

Name:	**Justine Gauthier**
Degree:	**Masters of Science (Mathematics)**
Title:	***Fast Multipoint Evaluation On n Arbitrary Points***
Examining Committee:	**Chair:** N/A

Michael Monagan
Supervisor
Professor

Nils Bruin
Co-Supervisor
Professor

Date Defended: August 17, 2017

Abstract

The Fast Fourier Transform evaluates a polynomial of degree less than n, at n powers of a primitive nth root of unity, in $O(n \log n)$ arithmetic operations. What if we wish to evaluate such a polynomial at n arbitrary points? Using Horner's method, this will take as many as $O(n^2)$ multiplications. This project will present and analyse a recursive algorithm which evaluates a polynomial of degree less n, at n arbitrary points, using only $O(n \log^2 n)$ arithmetic operations. This improvement allows fast multipoint evaluation at arbitrary points to be used in subquadratic algorithms. The implementation and running time of the algorithm in C will be explored.

Keywords: Fast Polynomial Evaluation, Subquadratic Algorithms, Fast Fourier Transform, Computer Algebra

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Suppose you are given a polynomial, $f(x) \in R[x]$, where R is a commutative ring with unity. You are also given $n \in \mathbb{N}$ points from the ring R. You wish to evaluate f at these n points either as a sub-procedure of a bigger problem, or as a stand alone inquiry. If n is small, then you may not care how the problem is approached. However, if n is, say, a million, it is worth while to use an algorithm which does not take hours to run. Formally, we are trying to solve the following problem:

Problem (Multipoint Evaluation)**.** *Suppose R is a commutative ring with unity. Given $n \in \mathbb{N}$, and $u_0, \ldots, u_{n-1} \in R$, and $f \in R[x]$ of degree less than n, compute*

$$f(u_0), \ldots, f(u_{n-1}).$$

This work will present a subquadratic algorithm, FastEval, to solve the multipoint evaluation problem under the assumption that R contains an nth root of unity, and $2^{-1} \in R$. This algorithm is presented in the paper *Evaluating polynomials at many points* by Borodin and Munro in 1971 [1]. The main idea of the algorithm comes from this paper, while the implementation in C was done in collaboration with Michael Monagan. The algorithm is also described in [5].

Classical polynomial evaluation using Horner's method for each u_i would solve this problem in $O(n^2)$ operations in R. It can be proven that one evaluation requires at least $O(n)$ multiplications. This fact may seem to imply that n evaluations would require $O(n^2)$ operations in R, however, the mass-production of evaluations can lead to significantly fewer arithmetic operations. The algorithm presented in this work will require at most $O(n \log^2 n)$ operations in R.

In order to give a proper overview of the work, the first chapter of this report will explore some of the algorithms which helped to make FastEval fast. Chapter 2 will present FastEval by first giving an overview of how the algorithm works, presenting a short example, and then digging into the mechanics of the algorithm. Timings of FastEval compared to the classical evaluation algorithm will be presented. The final chapter will discuss errors made during the work of this report, in hopes that the reader will avoid making the same mistakes. This chapter will also explore possible future works, as well as an interpolation algorithm which reuses some of the work computed by FastEval. We will end with a brief conclusion.

We now wish to make some assumptions, which will help to make our discussion more digestible. The final chapter will address these assumptions, demonstrating that the algorithm does in fact solve our general problem. The first assumption that we wish to make, is to assume $R = \mathbb{F}$ is a field which supports the Fast Fourier Transform. If $R = \mathbb{F}_p$, we will assume that p is a Fourier prime. These requirements will be discussed when the Fourier Transform is presented in the background material. For now, this assumption allows us to multiply polynomials using the Fast Fourier Transform in $O(n \log n)$ operations in R. Secondly, we will assume that n, the number of evaluation points, is a power of 2, and that each evaluation point is unique. Lastly, we will assume that $f \in R[x]$ is a polynomial of degree less than n. These assumptions will assist the discussion of the complexity of the algorithm.

1.1 Background

This chapter will review important algorithms required for evaluation. First, we will explore the classical polynomial evaluation method using Horner's form. Next, we will discuss the Fast Fourier Transform, both as an evaluation technique, and as a way to perform fast polynomial multiplication. Finally, we will look at the Newton iteration for dividing polynomials.

1.1.1 Classic Evaluation - Horner's Method

Consider the polynomial $f(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$, with $a_i \in R$ a ring. We can rewrite the polynomial $f(x)$ into the nested form, or Horner's form,

$$f(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-2} + x(a_{n-1}))) \cdots)).$$

To evaluate $f(x)$ at $x = \alpha$ for some $\alpha \in R$, we would first multiply $\alpha \cdot a_{n-1}$. Next, we would add a_{n-2} to αa_{n-1}, and then multiply by α again. Continuing on, we will eventually add a_0 to the result of $\alpha(a_1 + \alpha(a_2 + \alpha(a_3 + \cdots + \alpha(a_{n-2} + \alpha(a_{n-1}))) \cdots)$. We can see that evaluating $f(x)$ at α requires $n-1$ multiplications in the ring R, and $n-1$ additions. This results in $O(n)$ operations in the ring. This method of evaluation is called Horner's method.

If we want to evaluate f at n points using this method, we would need $n \cdot (n-1)$ multiplications, and $n \cdot (n-1)$ additions. Therefore, using Horner's method to evaluate a polynomial of degree $n-1$ at n arbitrary points requires $O(n^2)$ work.

Horner's method will be used in this project to compare and measure the improvements of the algorithm being presented. All comparisons made to classical evaluation will refer to Horner's method.

1.1.2 The Fast Fourier Transform

Let $n = 2^k \in \mathbb{N}$, $f(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$, where $a_i \in R$, where R is a commutative ring with unity. We say ω is a primitive nth root of unity if $\omega^n = 1$ and $\omega^k \neq 1$ for all $0 < k < n$. The R-linear map

$$DFT_\omega : \begin{cases} R^n \longrightarrow R^n \\ f \longmapsto (f(1), f(\omega), \ldots, f(\omega^{n-1})) \end{cases}$$

is called the Discrete Fourier Transform. It evaluates the polynomial f at the powers of ω, a primitive nth root of unity. The Fast Fourier Transform, or FFT, computes the Discrete Fourier Transform in $\frac{1}{2}n\log_2 n$ multiplications in R. The FFT takes advantage of the properties of the primitive nth root of unity presented in the following Lemma.

Lemma 1.1.1. *Let $\omega \in R$ be a primitive nth root of unity. Then*

- $\omega^j = -\omega^{j+n/2}$,
- ω^2 *is a primitive $n/2$th root of unity,*
- $\omega^0 + \omega^1 + \cdots + \omega^{n-1} = 0$,
- ω^{-1} *is a primitive nth root of unity.*

By evaluating f at $\omega^0, \omega^1, \ldots, \omega^{n-1}$, we can cut down on the number of operations by first noticing f can be written as

$$f(x) \quad = \quad [a_0 + a_2x^2 + \cdots + a_{n-2}x^{n-2}] + x[a_1 + a_3x^2 + \cdots + a_{n-1}x^{n-2}].$$

Let $b(x) = a_0 + a_2x + \cdots + a_{n-2}x^{n/2-1}$, and let $c(x) = a_1 + a_3x + \cdots + a_{n-1}x^{n/2-1}$. Then

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

Using the property that $\omega^j = -\omega^{j+n/2}$, we see that $b(x^2)$ evaluated at $x = \omega^j$ is identical to $b(x^2)$ evaluated at $x = \omega^{j+n/2}$, thus saving about half of the arithmetic operations in R. This technique is repeated recursively, forming the general idea behind the Fast Fourier Transform. We now present pseudo code for the algorithm, and examine its running time.

Algorithm 1 Fast Fourier Transform (FFT)

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $a = [a_0, a_1, \ldots, a_{n-1}] \in R^n$ and a primitive $n$th root of
    unity $\omega \in R$.
Output: $DFT_\omega(f) = \big(f(1), f(\omega), \ldots, f(\omega^{n-1})\big) \in R^n$ where $f = \sum_{j=0}^{n-1} a_i x^i$.
1: if $n = 1$ then return: $a_0 \in R$
2: $b \longleftarrow [a_0, a_2, \ldots, a_{n-2}]$
3: $c \longleftarrow [a_1, a_3 \ldots, a_{n-1}]$
4: $B \longleftarrow FFT(n/2, b, \omega^2)$
5: $C \longleftarrow FFT(n/2, c, \omega^2)$
6: $y \leftarrow 1$
7: for $i = 0, 1, \ldots, n/2 - 1$ do
8:     $T \leftarrow y \cdot C_i$
9:     $a_i \leftarrow B_i + T$
10:    $a_{i+n/2} \leftarrow B_i - T$
11:    $y \leftarrow y \cdot \omega$
12: return: $[a_0, \ldots, a_{n-1}] \in R^n$.
```

Let $T(n)$ be the number of multiplications in R done by Algorithm 1. If $n = 1$, then the input polynomial is an element of R, and is returned. Therefore, $T(1) = 0$. Next, the algorithm does two recursions of size $n/2$. These calls require one multiplication of to find ω^2. After the recursive calls, the algorithm performs $n/2$ multiplications of y times c_i, and $n/2$ multiplications of y times ω, all of which are done in the ring R. Therefore,

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

Using Maple's rsolve command, we find that $T(n) = n \log_2 n + n - 1 \in O(n \log n)$.

An optimization presented in a paper by Law and Monagan [4] allows us to further cut down on the multiplications required to compute the $DFT_\omega(f)$. By pre-computing the necessary powers of ω in an array, namely ω^i for $0 \le i < n/2.$, we cut down on over half of the multiplications in the loop. Without having to calculate ω^2 for the recursive calls, or updating $y = y \cdot \omega$ at every step, we have

$$T(n) = 2T(n/2) + \frac{n}{2}.$$

When solved with $T(1) = 0$, we have $T(n) = \frac{1}{2} n \log_2 n$.

Along with the Fast Fourier Transform, we also have the Inverse Fourier Transform. As one may expect from the name, the Inverse Fourier Transform takes as input $n = 2^k$, $DFT_\omega(f) = \big(f(1), f(\omega), \ldots, f(\omega^{n-1})\big) \in R^n$, and ω^{-1}. The Inverse FFT performs an interpolation on the outputs of f evaluated at the powers of ω. What is more surprising, is that the same algorithm which evaluates the polynomial, can also interpolate it. Consider the evaluation

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2n-2} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & \omega^{n-1} & \omega^{2n-2} & \dots & \omega^{(n-1)^2} \end{bmatrix}}_{V_\omega} \underbrace{\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{n-1} \end{bmatrix}}_{F} = \underbrace{\begin{bmatrix} f(1) \\ f(\omega) \\ f(\omega^2) \\ \vdots \\ f(\omega^{n-1}). \end{bmatrix}}_{F_\omega}$$

Suppose we know F_ω, and want to compute F. Using a classic matrix inversion to find $(V_\omega)^{-1}$ and then calculating $(V_\omega)^{-1} F_\omega = F$ would require $O(n^3)$ operations in the ring. Recall that if ω is a primitive nth root of unity, then so too is ω^{-1}.

Proposition 1.1.1. *$V_\omega \cdot V_{\omega^{-1}} = nI$, which implies $(V_\omega)^{-1} = \frac{1}{n} V_{\omega^{-1}}$.*

Proof. Let W be the product of $V_\omega \cdot V_{\omega^{-1}}$. Consider the elements on the diagonal of W that derive from the jth row of V_ω and the jth column of $V_{\omega^{-1}}$. We have

$$\sum_{i=0}^{n-1} \omega^{2ij} \cdot \omega^{-2ij} = \sum_{i=0}^{n-1} 1 = n.$$

Thus, each element on the diagonal of W is n. Knowing that $\omega^0 + \omega^1 + \cdots + \omega^{n-1} = 0$, it is not hard to show that all other elements will be zero. Therefore, $V_\omega \cdot V_{\omega^{-1}} = W = nI$. The implication follows. $\square$

We can now use the fact that

$$F = V_\omega^{-1} F_\omega = \frac{1}{n} \left(V_{\omega^{-1}} F_\omega \right) = \frac{1}{n} DFT_{\omega^{-1}}(F_\omega) \in R^n.$$

This result allows us to use the Fast Fourier Transform to multiply polynomials efficiently as follows:

Algorithm 2 FFT multiplication

```
Input: $a = [a_0, a_1, \ldots, a_{n-1}], b = [b_0, \ldots, b_{n-1}] \in R^n$, $R$ a ring that supports the FFT,
    $n = 2^k$ for some $k \in \mathbb{N}$, and a primitive $n$th root of unity $\omega \in R$.
Output: $c = ab \in R^n$
1: $A \leftarrow FFT(n, \omega, a)$, $B \leftarrow FFT(n, \omega, b)$
2: $C \leftarrow [A_0 \cdot B_0, A_1 \cdot B_1, \ldots, A_{n-1} \cdot B_{n-1}]$
3: $C \leftarrow FFT(n, C, \omega^{-1})$
4: $c = n^{-1} \cdot C$
5: return: $[c_0, c_1, \ldots, c_{n-1}] \in R^n$.
```

Algorithm 2 requires 3 FFT calls of size n, where n is the smallest power of two greater than the sum of the degrees of the inputs. In this report, we will refer to the work required to compute a Fast Fourier Transform of size n as $\mathsf{FFT}(n)$. It also requires the pointwise multiplication of the vectors A and B, which is n multiplications in R. Finally, each element of c must be multiplied by n^{-1}. Therefore, the work required to multiply two polynomials with an FFT multiplication of size n is

$$3\mathsf{FFT}(n) + 2n = \frac{3}{2} n \log_2 n + O(n).$$

For a polynomial multiplication whose product has degree less than n, we will refer to the work required as $\mathsf{M}(n)$. It will be assumed that all polynomial multiplications are done using the FFT, hence

$$\mathsf{FFT}(n) = \frac{1}{2} n \log_2 n, \text{ and } \mathsf{M}(n) = \frac{3}{2} n \log_2 n + O(n).$$

Note that not every field has a primitive nth root of unity.

Theorem. *The finite field $\mathbb{Z}_p$ has a primitive nth root of unity if and only if n divides $p-1$.*

Proof. The following proof is similar to a proof given in *Algorithms for Computer Algebra.* [2]

Suppose ω is a primitive nth root of unity in $\mathbb{Z}_p$. We call the point set $\{1, \omega, \ldots, \omega^{n-1}\}$ the set of Fourier points. The Fourier points form a cyclic subgroup of the multiplicative group $\mathbb{Z}_p$. We know from group theory that the cardinality of a multiplicative subgroup must divide the size of the group. Since $\{1, \omega, \ldots, \omega^{n-1}\}$ has n elements, and $\mathbb{Z}_p$ has $p-1$, it follows that n divides $p-1$.

Suppose n divides $p-1$. We know from field theory that the field $\mathbb{Z}_p$ is cyclic, and there exists $\varphi(p-1)$ generators. Let $\alpha \in \mathbb{Z}_p$ be a generator such that $\alpha^{p-1} = 1$ and

$$\mathbb{Z}_p^* = \{1, \alpha, \ldots, \alpha^{p-2}\}$$

If we choose $\omega = \alpha^{(p-1)/n}$, it follows that

$$\omega^n = \left(\alpha^{(p-1)/n}\right)^n = 1$$

by the choice of α. Therefore, ω is an nth root of unity. Furthermore, for $0 < k < n$, $\omega^k = 1$ would contradict α being a generator for $\mathbb{Z}_p$. Hence ω is a primitive nth root of unity. □

We say a prime p is a Fourier prime if $p-1$ is divisible by a large power of two. Working in a field F_p, where p is a Fourier prime, allows us to use FFT multiplication on polynomials of many different degrees. For example our implementation was done over the field $\mathbb{Z}_p$, where $p = 15 \cdot 2^{27} + 1$, the largest 31 bit Fourier prime.

1.1.3 Newton Iteration

Let $a, b \in F[x]$, where F is a field, and

$$a(x) = a_0 + a_1 x + \cdots + a_{2n-2}x^{2n-2}, \quad b(x) = b_0 + b_1 x + \cdots + b_{n-1}x^{n-1}.$$

Suppose we wish to divide a by b such that $a = qb + r$, where $q \in F[x]$, and the degree of $r \in F[x]$ is either zero or less than the degree of b. To perform this division using a classical long division algorithm may require $O(n^2)$ operations in F. Can we compute the quotient of a divided by b in fewer than $O(n^2)$ operations? Let

$$a^*(x) = x^{2n-2}a(1/x) = a_0 x^{2n-2} + \cdots + a_{2n-2}, \quad b^*(x) = x^{n-1}b(1/x) = b_0 x^{n-1} + \cdots + b_{n-1}.$$

We call a^* and b^* the reciprocal polynomials of a and b, respectively. This manipulation results in reversing the order of the coefficients of the original polynomials. It follows that

$$a(x) = q(x)b(x) + r(x) \iff a^*(x) = q^*(x)b^*(x) + x^{n-1+\lambda}r^*(x),$$

for $\lambda \geq 1$. If we can find the quotient, q^*, we can then compute the remainder, r, via $a - qb$. We want to calculate the inverse of b^* to the smallest order necessary to be able to compute the quotient, which has degree $n-1$.

We say a power series $\bar{y}(x)$ is an order n approximation of $y(x)$ if

$$\bar{y}(x) = y(x) + O(x^n) \text{ or } \bar{y}(x) \equiv y(x) \mod x^n.$$

Therefore, it is sufficient to calculate the first $n-1$ terms of the power series of $1/b^*$, and then multiply the result by a^* to determine q^*. Let n be a power of two. Newton's method for power series inversion begins with an initial approximation, $y_0 = 1/b_{n-1}$, or the inverse of the constant term. Next, for k from 1 to $\log_2 n$, we have

$$y_k \equiv 2y_{k-1} - y_{k-1}^2 b^* \mod x^{2^k}.$$

Notice that from the previous iteration, we have $y_{k-1}b^*(x) \equiv 1 \mod x^{2^{k-1}}$. At each step, the Newton iteration multiplies y_{k-1}, which is of degree less than 2^{k-1} by itself, and then multiplies the product by b^*, which is of degree less than 2^k. This work is followed by a subtraction. This requires one FFT multiplication of size 2^k, followed by a second FFT multiplication of size 2^{k+1}. Let $I(n)$ be the number of arithmetic operations in F required to compute $1/b^*$ to an order n approximation. Note that $I(1) = 1$, the work required to compute the inverse. If $n = 2^k$ then

$$\begin{aligned} I(2^k) &\le I(2^{k-1}) + \mathsf{M}(2^k) + \mathsf{M}(2^{k+1}) + O(n) \\ &< I(2^{k-1}) + \frac{1}{2}\mathsf{M}(2^{k+1}) + \mathsf{M}(2^{k+1}) + O(n) \\ &< \frac{3}{2}\mathsf{M}(2^{k+1}) + \frac{3}{2}\mathsf{M}(2^{k+1}) + \cdots + \frac{3}{2}\mathsf{M}(2) + O(n) \\ &< 3\mathsf{M}(2^{k+1}) + O(n). \end{aligned}$$

Hence, the Newton iteration calculates the inverse of b^* to $O(x^n)$ using work which is equivalent to no more than three degree $2n$ polynomial multiplications, plus some linear work. Once we have the inverse of b^*, we are left to calculate the quotient. We calculate

$$q^* = a^* \cdot (1/b^*) \mod O(x^n),$$

using another FFT based multiplication. Finally, we reverse q^* back to q, and calculate $r = a - bq$. Multiplying b and q requires a final FFT multiplication, for a total of 5 FFT multiplications. The work done after calculating $1/b^* \mod x^n$ will be discussed in the following chapter. For now we are only interested in the inversion. Next, we present an improvement which decreases the size of the larger FFT multiplication.

The Middle Product Optimization

Let $n = 2^k$. First observe that

$$y_k = 2y_{k-1} - y_{k-1}^2 b^* \mod x^{2^k} = y_{k-1} + y_{k-1}(1 - b^* y_{k-1}) \mod x^{2^k}.$$

Normally, to multiply a polynomial of degree $2n-1$ by a polynomial of degree $n-1$ the FFT multiplication procedure requires an array which can hold at least $3n$ elements in the field. Since the FFT requires the input size to be a power of two, we are required to use an FFT multiplication of size $4n$. Knowing that $1 - y_k b^* \equiv 0 \mod x^{2^k}$, we are able to predict that the product $y_k b^*$ will have the following form

$$\begin{aligned} y_k \cdot b^* &= 1 + 0 \cdot x + \cdots + 0 \cdot x^{n-1} + m_0 \cdot x^n + m_1 \cdot x^{n+1} + \cdots + m_{n-1} \cdot x^{2n-1} + O(x^{2n}) \\ &= 1 + 0 \cdot x + \cdots + 0 \cdot x^{n-1} + x^n \underbrace{\left(m_0 + m_1 \cdot x + \cdots + m_{n-1} \cdot x^{n-1}\right)}_{m(x)} + O(x^{2n}) \\ &= 1 + 0 \cdot x + \cdots + 0 \cdot x^{n-1} + x^n m(x) + x^{2n} \cdot \left(c_0 + c_1 \cdot x + \cdots + c_{n-1} \cdot x^{n-1}\right) \end{aligned}$$

where $m_i, c_i \in R$ for $i = 0 \ldots k$ and $m(x) \in R[x]$.

Proposition 1.1.2. *Given $a(x), b(x) \in R[x]$ of degree less than n, $n = 2^k \in \mathbb{N}$, the Fast Fourier Transform multiplication procedure of size n computes $c(x) = a(x) \cdot b(x) \mod x^n - 1$.*

Proof. Recall the R-linear map DFT_ω which computes the Discrete Fourier Transform. First, we define the convolution of two polynomials, $a = \sum_{j=0}^{n-1} a_j x^j$ and $b = \sum_{k=0}^{n-1} b_k x^k$ in $R[x]$ as the polynomial

$$c = a *_n b = \sum_{0 \le \ell < n} c_\ell x^\ell \in R[x]$$

where

$$c_\ell = \sum_{j+k \equiv \ell \mod n} a_j b_k = \sum_{0 \le j < n} a_j b_{\ell - j} \text{ for } 0 \le \ell < n.$$

This notion of convolution is equivalent to polynomial multiplication in the ring $R[x]/\langle x^n - 1\rangle$. The indices on a and b should be regarded mod n. Consider the Discrete Fourier Transform of such a convolution. We have $a * b = ab + q \cdot (x^n - 1)$ for some $q \in R[x]$. Then

$$(a * b)(\omega^j) = a(\omega^j)b(\omega^j) + q(\omega^j)(\omega^{jn} - 1) = a(\omega^j)b(\omega^j)$$

for $0 \le j < n$. Therefore, we can say that $DFT_\omega(a * b) = DFT_\omega(a) \cdot DFT_\omega(b)$ where $\cdot$ represents pointwise multiplication of the vectors, $DFT_\omega(a)$ and $DFT_\omega(b)$.

Therefore, the FFT calculates the product $c(x) = a(x) \cdot b(x) \mod x^n - 1$. □

In other words, if the degree of the product of two polynomials exceeds n, then the coefficients wrap back around. The coefficient on x^n is added to the constant term, the coefficient on x^{n+1} is added to the coefficient in front of x, and so on. While calculating y_{k+1}, or y to order $2n$, the output of the FFT multiplication of size $4n = 2^{k+2}$ would give an output of

$$\begin{aligned} y_k \cdot b^* = 1 + 0x + \cdots + 0x^{n-1} + m_0 \cdot x^n + \cdots + m_{n-1} \cdot x^{2n-1} \\ + c_0 \cdot x^{2n} + \cdots + c_{n-1} \cdot x^{3n-2} + 0 \cdot x^{3n-1} + \cdots + 0 \cdot x^{4n} \end{aligned}$$

for $m_i, c_i \in R$. To compute $1 - y_k b^* \mod x^{2n}$, we are only interested in the middle polynomial, or the middle product, $m(x)$, defined above as

$$m(x) = m_0 \cdot + \cdots + m_{n-1} \cdot x^{n-1}.$$

Consider using an FFT multiplication of size $2n$. We get

$$y_k \cdot b^* \mod x^{2n} = (1 + c_0) + c_1 \cdot x + \cdots + c_{n-1} \cdot x^{n-1} + m_0 \cdot x^n + \cdots + m_{n-1} \cdot x^{2n-1}.$$

Therefore, we can extract the $m(x)$ polynomial from the top half of the output of a size $2n$ FFT multiplication.

Recall $I(n)$ was defined as the cost of computing $1/b^*$ to an order n approximation. This improvement allows us to perform two FFT multiplications of equal size. Hence,

$$\begin{aligned}I(n) = I(2^k) &\leq I(2^{k-1}) + \mathsf{M}(2^k) + \mathsf{M}(2^k) + O(2^k)\\ &< I(2^{k-1}) + 2\mathsf{M}(2^k) + O(2^k)\\ &< 2\mathsf{M}(2^k) + 2\mathsf{M}(2^{k-1}) + \cdots + 2\mathsf{M}(2) + O(2^k) + O(2^{k-1}) + \cdots + O(1)\\ &< 4\mathsf{M}(2^k) + O(2^k) < 2\mathsf{M}(2^{k+1}) + O(2^k).\end{aligned}$$

This improvement reduces the number of FFT multiplication calls by reducing the size of the FFT multiplications from $2n$ to size n. Note that the work required to compute $4\mathsf{M}(n)$ is less than the work required to compute $2\mathsf{M}(2n)$. Prior to this improvement, we required at least $3\mathsf{M}(2^{k+1}) + O(2^k)$ work. The middle product optimization is accredited to Hanrot, Quercia and Zimmermann [3].

Chapter 2

FastEval: The Algorithm

2.1 Overview

The fast evaluation algorithm presented in this section requires first building a binary tree of products of polynomials, which we will refer to as the subproduct tree. The tree is built from the leaves up, and relies only on the evaluation points, not on the polynomial being evaluated. Therefore, this work may be computed once and re-used to evaluate other polynomials at the same evaluation points. We will see later that this work can also be re-used in other algorithms.

After the tree has been computed, the algorithm makes use of the Chinese Remainder theorem by considering the input polynomial modulo the polynomials in the subproduct tree.

2.1.1 Multiplying up the Tree

Let $u_0, \ldots, u_{n-1}$ be given points in the ring R. The first step of the algorithm is building the subproduct tree, see Figure 1. To build the tree, we start with the polynomials $x - u_i$ for $0 \leq i < n$ as the leaves. Each node represents a monic polynomial that is constructed as the product of its children. The polynomial $M_{i,j}$ resides at height i, j nodes from the left, and is the product of all the leaves that lay underneath it. The root of the tree represents $M_{k,0} = \prod_{i=0}^{n-1}(x - u_i)$, and each leaf represents $M_{0,j} = x - u_j$. Note that the largest polynomial, $M_{k,0}$, is not computed because we do not use it in the computation. We include it in the discussion only to complete the binary tree. However, in implementation we only compute up to $M_{k-1,0}$ and $M_{k-1,1}$.

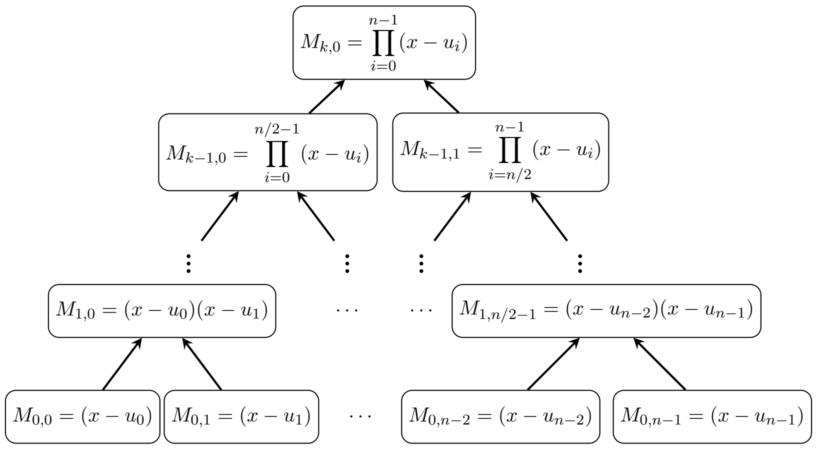
$M_{k,0} = \prod_{i=0}^{n-1}(x - u_i)$
$M_{k-1,0} = \prod_{i=0}^{n/2-1}(x - u_i)$
$M_{k-1,1} = \prod_{i=n/2}^{n-1}(x - u_i)$
$M_{1,0} = (x - u_0)(x - u_1)$
$M_{1,n/2-1} = (x - u_{n-2})(x - u_{n-1})$
$M_{0,0} = (x - u_0)$
$M_{0,1} = (x - u_1)$
$M_{0,n-2} = (x - u_{n-2})$
$M_{0,n-1} = (x - u_{n-1})$

Figure 1: Subproduct Tree

If R is a commutative ring with unity, and $u_0, \ldots, u_{n-1} \in R$ are distinct, then each polynomial $M_{i,j}$ in Figure 1 is the monic square-free polynomial whose zero set is the jth node from the left at level i. The following pseudo code gives a general method for building the subproduct tree. If the FFT multiplication is used for polynomial multiplications, then we obtain a subquadratic time algorithm for arbitrary points $u_0, \ldots, u_{n-1} \in R$.

Algorithm 3 Building up the Subproduct tree

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $u_0, \ldots, u_{n-1} \in R$.
Output: The polynomials $M_{i,j}$ for $0 \le i \le k$ and $0 \le j < 2^{k-i}$.
1: for $j = 0, \ldots, n-1$ do $M_{0,j} \leftarrow (x - u_i)$
2: for $i = 1, \ldots, k$ do
3:     for $j = 0, \ldots, 2^{k-i} - 1$ do $M_{i,j} \leftarrow M_{i-1,2j} \cdot M_{i-1,2j+1}$
```

Before discussing the correctness, termination, or running time of this step of the algorithm, we wish to present the rest of the algorithm, motivating the work already presented.

2.1.2 Dividing down the Tree

Once the subproduct tree has been computed, we can begin to evaluate f with the fast multipoint evaluation algorithm, which we call FastEval. The algorithm is a straight forward divide-and-conquer algorithm which utilizes the Chinese Remainder Theorem over $R[x]$.

Let $R = \mathbb{Z}_p$ for p a Fourier prime. For $0 \le i < n$, let $m_i = x - u_i$, and define the canonical ring homomorphism

$$\pi_i : R \longrightarrow R/\langle m_i \rangle,$$
$$\pi_i(f) = \quad f \mod m_i.$$

Recall that the composition of ring homomorphisms is again a ring homomorphism. It follows that

$$\begin{aligned}\chi = \pi_0 \times \cdots \times \pi_{n-1} : R \to R/\langle m_0\rangle \times \cdots \times \langle m_{n-1}\rangle,\\ \chi(f) = (f \mod m_0, \ldots, f \mod m_{n-1})\end{aligned}$$

is also a ring homomorphism.

Consider what it means to take $f \mod m_i$. We are essentially dividing f by $(x - u_i)$ and keeping the remainder. Notice that we chose the moduli in such a way that f evaluated at u_i is

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

This is equivalent to saying that $f(u_i) = f(x) \mod (x - u_i)$. Since $f \mod m_i$ must have degree less than the degree of m_i, and each m_i is linear, it follows that $f \mod m_i \in R$. Therefore,

$$\begin{aligned}&\chi : R \to R/\langle m_0\rangle \times \cdots \times \langle m_{n-1}\rangle,\\ &\chi(f) = (f(u_1), \ldots, f(u_{n-1})).\end{aligned}$$

We now have a method for evaluating a polynomial at n points. However, dividing a polynomial of degree $n - 1$ by n linear polynomials is still $O(n^2)$, as each division requires roughly n multiplications in the ring. We can save work by performing larger divisions, rather than n linear divisions. This is where our precomputed subproduct tree becomes useful. Instead of dividing f by the leaves of the tree, we will recurse down the tree. First, let

$$r_0 = f \mod \prod_{i=0}^{n/2-1} (x - u_i) = M_{k-1,0} \text{ and } r_1 = f \mod \prod_{i=n/2}^{n-1} (x - u_i) = M_{k-1,1}.$$

Next, call the algorithm on inputs $r_0, n/2$ and the subtree rooted at $M_{k-1,0}$ and again on inputs $r_1, n/2$ and the subtree rooted at $M_{k-1,1}$. Since the subproduct tree is a binary tree of height $\log_2 n$, we will reduce the number of polynomial divisions required to compute an evaluation to $O(\log n)$. The exact number of operations required will be explored in more detail in the following sections.

Algorithm 4 Dividing down the Subproduct tree

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $f \in R[x]$ of degree less than $n$, and the subproducts $M_{i,j}$
Output: $f(u_0), \ldots, f(u_{n-1}) \in R$.
1: if $n = 1$ then return $f \in R$
2: $r_0 \leftarrow f \mod M_{k-1,0}$
3: $r_1 \leftarrow f \mod M_{k-1,1}$
4: call the algorithm with input $r_0, n/2$ and the subtree rooted at $M_{k-1,0}$ to compute
   $r_0(u_0), \ldots, r_0(u_{n/2-1})$
5: call the algorithm with input $r_1, n/2$ and the subtree rooted at $M_{k-1,1}$ to compute
   $r_1(u_{n/2}), \ldots, r_1(u_{n-1})$
6: return $r_0(u_0), \ldots, r_0(u_{n/2-1}), r_1(u_{n/2}), \ldots, r_1(u_{n-1})$
```

Finally, we combine the two procedures, building up and dividing down, to create the final algorithm:

Algorithm 5 FastEval: Fast multipoint evaluation

```
Input: $n = 2^k$ for some $k \in \mathbb{N}$, $f \in R[x]$ of degree $n-1$, $u_0, \ldots, u_{n-1} \in R$.
Output: $f(u_0), \ldots, f(u_{n-1}) \in R$.
1: call Algorithm 3 with inputs $n, u_0, \ldots, u_{n-1}$
2: call Algorithm 4 with inputs $f, n$, and the subproducts $M_{i,j}$
```

2.2 Proof of Correctness

The subproduct tree is built from the leaves up to the root. Let $m_i = x - u_i$ for all $0 \leq i < n$. Now, let

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

It follows that each $M_{i,j}$ is a subproduct with 2^i factors of $M_{k,0} = \prod_{0 \leq \ell < n} m_\ell$, and satisfies for each i, j the recursive equations

$$M_{0,j} = m_j, \quad \text{and} \quad M_{i+1,j} = M_{i,2j} \cdot M_{i,2j+1}.$$

Proof of correctness of the pre-computation follows directly from this.

The correctness of Algorithm 4 was discussed in the overview, but will be proven by induction on $k = \log_2 n$. If $k = 0$, then f is constant, and Algorithm 4 will return f at step 1. Assume $k \geq 1$, and take steps 3 and 4 to be correct by the inductive hypothesis. Let q_0 be the quotient of f divided by $M_{k-1,0}$, and q_1 the quotient of f divided by $M_{k-1,1}$.

Evaluating f at u_i gives:

$$f(u_i) = \begin{cases} q_0(u_i) \cdot M_{k-1,0}(u_i) + r_0(u_i) = r_0(u_i) & \text{if } 0 \leq i < n/2 \\ q_1(u_i) \cdot M_{k-1,1}(u_i) + r_1(u_i) = r_1(u_i) & \text{if } n/2 \leq i < n \end{cases}$$

Correctness follows immediately.

2.3 Example

Let $f(x) = 4x^3 + 3x^2 + 2x + 1 \in \mathbb{Z}_{97}$ and suppose we wish to evaluate f at the following $n = 4$ points:

$$u_0 = 1, u_1 = 2, u_2 = 3, u_3 = 4.$$

The corresponding subproduct tree is shown in Figure 2 below.

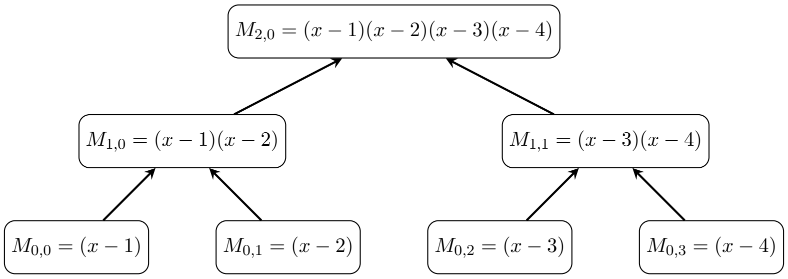

Figure 2: Subproduct Tree for Evaluation points $u_0 = 1, u_1 = 2, u_2 = 3, u_3 = 4$.

Construction of the tree is clear from Figure 2. Each node is the product of its children. Note that the root, $M_{2,0} = (x-1)(x-2)(x-3)(x-4)$, would not be computed in implementation.

Once the tree has been constructed, we begin Algorithm 4, which is illustrated by Figure 3. First, we start by dividing f by $M_{1,0}$ to find the remainder $r_0 = 39x + 68$, and then by $M_{1,1}$ to find the remainder $r_1 = 74x + 17$. We then call the algorithm recursively using inputs $r_0, n/2$ and again on $r_1, n/2$.

Next, the algorithm computes the remainders of r_0 and r_1 divided by the children of the nodes used in the last step. Specifically, it calculates

$$\begin{aligned} r_0 &= 39x + 68 \mod (x-1) = 10, & r_1 &= 39x + 68 \mod (x-2) = 49 \\ r_0 &= 74x + 17 \mod (x-3) = 45, & r_1 &= 74x + 17 \mod (x-4) = 22 \end{aligned}$$

The algorithm will attempt another recursion. Since the moduli are degree one, we know the results will all be constant terms in the ring, satisfying the base of recursion for $n = 1$. The algorithm returns these results, and the evaluation is complete.

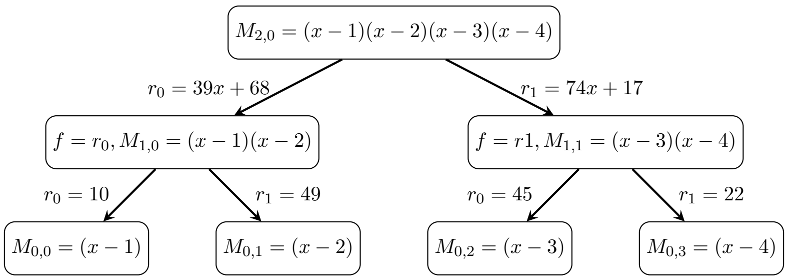

Figure 3: Dividing down for Evaluation points $u_0 = 1, u_1 = 2, u_2 = 3, u_3 = 4$.

We have successfully mapped f to its evaluations:

$$\begin{aligned}\chi : \mathbb{Z}_{97}[x] &\longrightarrow \mathbb{Z}_{97}/\left(\langle x-1\rangle \times \langle x-2\rangle \times \langle x-3\rangle \times \langle x-4\rangle\right),\\ \chi\left(4x^3 + 3x^2 + 2x + 1\right) &= (10, 49, 45, 22).\end{aligned}$$

2.4 Complexity and Implementation

FastEval is a significant improvement to naive polynomial evaluation. Creating an evaluation algorithm to run in subquadratic time allows it to be used as a subroutine in modular algorithms. The importance of having all subroutines within the algorithm be subquadratic can not be over emphasised. In this section, we will analyse the number of operations required to run FastEval, and discuss the challenges of implementing the code in subquadratic time. Errors which arose during the process will be discussed in the following chapter, in hopes of preventing others from making the same mistakes. Finally, some data will be presented as proof of the subquadratic complexity, highlighting the actual improvement in timing from a quadratic algorithm to an algorithm that runs in $O(n \log^2 n)$.

2.4.1 Multiplying Up the Tree

The pre-computation stage begins at the bottom of the tree. At the mth step, the algorithm must multiply $n/2^m = 2^{k-m}$ pairs of polynomials, each of degree 2^{m-1}. We are able to cut down on the size of our FFT multiplications in this step by using Proposition 1.1.2, and knowing that each product will be a monic polynomial of degree 2^m. Normally, to compute a product of degree 2^m, we would need an FFT of size 2^{m+1}, in order to fit all $2^m + 1$ coefficients. We can multiply using an FFT of size 2^m because we know that the coefficient on x^{2^m} will be a one, and it will end up in front of x^0. Therefore, we can subtract the one from the constant to reveal the solution. It follows that the mth step requires 2^{k-m} FFT multiplications of size 2^m. Recall that we let $\mathsf{M}(n)$ represent the number of arithmetic operations in R required to run the FFT multiplication whose product has degree less than n. Since calls to the FFT multiplication algorithm require more than linear work, we use the fact that $\mathsf{M}(n) > 2\mathsf{M}(n/2)$, to see

$$\mathsf{M}\left(2^m\right) = \mathsf{M}\left(\frac{n}{2^{k-m}}\right) < \frac{1}{2^{k-m}}\mathsf{M}\left(n\right) \implies 2^{k-m}\mathsf{M}\left(\frac{n}{2^{k-m}}\right) < \mathsf{M}\left(n\right),$$

for $0 \leq m < k$. Therefore, the total work required to compute the subproduct tree is

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

The pre-computation step takes less than $\log_2 n$ polynomial multiplications of size n. The case where the FFT is not supported in the ring will be discussed later.

2.4.2 Dividing Down the Tree

The most expensive part of the entire evaluation algorithm is taking $f \mod M_{i,j}$ for each node $M_{i,j}$ in the subproduct tree. The algorithm runs recursively, first calling a subroutine to divide f by $M_{i,j}$ and $M_{i,j+1}$. The algorithm then calls itself twice on the subtrees rooted at $M_{i,j}$ and $M_{i,j+1}$. Let $T(n)$ be the number of operations in R required to completely run the algorithm through the tree. Let $D(n)$ be the number of operations in R required by the fast division algorithm to divide a polynomial of degree $n-1$ by a polynomial of degree $n/2$. It follows that $T(n) = 2T(n/2) + 2D(n)$. When $n = 1$, we have a zero degree polynomial and the algorithm returns the input value. Therefore, $T(1) = 0$. To solve this relation, we must first examine the division routine.

As discussed in the background material, the division algorithm used in this work utilizes two fast algorithms to compute the remainder in subquadratic time. Taking as inputs two polynomials $a, b \in R[x]$, the procedure calls the Newton inversion routine to compute the inverse of the quotient. Previously, we examined the number of operations in R required to compute the inverse of the reciprocal polynomial $b^* \in R[x]$. We concluded that the algorithm required $2\mathsf{M}(2n) + O(n)$, or roughly two FFT multiplications of size $2n$. We have $\mathsf{M}(n) = \frac{3}{2}n \log_2 n + O(n)$, implying that the inversion requires $6n \log_2(2n)$ operations in R, plus some linear work. Can we again improve on this work?

Recall that at each step of the iteration, we must compute

$$y_k = 2y_{k-1} - y_{k-1}^2 b^* \mod x^{2^k} = y_{k-1} + y_{k-1}(1 - b^* y_{k-1}) \mod x^{2^k}.$$

We have already improved on the multiplication of $b^* y_k$ by reducing the size of the FFT needed to compute the product. We make another improvement by noting that we are calculating the forward Discrete Fourier Transform of y_k for two multiplications, $b^* y_k$ and $y_k(1 - b^* y_k)$, both of the same size. Thus, instead of using 6 calls to the FFT procedure to do the two multiplications, we can save the work done to transform y_k, resulting in only 5 calls to the FFT. We also save work by computing the powers of ω and ω^{-1} once. Recall that $I(n)$ is defined as the number of arithmetic operations in R required to compute $1/b^*$ to an order n approximation. We have reduced this number to

$$\begin{aligned} I(n) &< I(n/2) + 5\mathsf{FFT}(2n) + O(n) \\ &< 10\mathsf{FFT}(2n) + O(n) \\ &\approx 3\frac{1}{3}\mathsf{M}(n). \end{aligned}$$

Therefore, the Newton inversion is a recursive algorithm whose work is no more than 10 calls of size $2n$ to the Fast Fourier Transform to compute b^* to order x^n. In Algorithm 4, at the ith level of recursion, we are dividing a polynomial of degree $2^{k-i} - 1$ by a polynomial from the subproduct tree of degree 2^{k-1-i}. Let $a \in R[x]$ be the polynomial being divided by $b \in R[x]$. For $n = 2^k$, a is degree $n - 1$, and the divisor, b, is degree $n/2$. To compute the inverse of b^* truncated to $O(x^{n/2})$, we need 10 Fast Fourier Transforms of size n. Since the FFT is quasilinear, this is less than 5 calls to the FFT of size $2n$.

Once the inverse is computed, the quotient is calculated by multiplying a^* by $1/b^*$ using an FFT multiplication routine of size n. We now have the reciprocal of the quotient, q^*, and can find q from q^* by reversing the coefficients. Finally, the remainder is computed by noticing that $r = a - bq$. This requires a final FFT multiplication of size n, and a subtraction. As discussed previously, each FFT multiplication requires two forward transforms and one backwards transform. Then we have,

$$\begin{aligned} D(n) &\le 10\mathsf{FFT}(n) + 2\mathsf{M}(n) + O(n) \\ &< 5\mathsf{FFT}(2n) + \mathsf{M}(2n) + O(n) \\ &\le 8\mathsf{FFT}(2n) = 8n\log_2 2n + O(n). \end{aligned}$$

We can now examine the time complexity of dividing down the tree. We have

$$\begin{aligned} T(n) &= 2T(n/2) + 2D(n) \le 2T(n/2) + 16\mathsf{FFT}(2n) \\ 2T(n/2) &\le 4T(n/4) + 2\cdot 16\mathsf{FFT}(n) \le 4T(n/4) + 16\mathsf{FFT}(2n) \\ 4T(n/4) &\le 8T(n/8) + 4\cdot 16\mathsf{FFT}(n/2) \le 8T(n/8) + 16\mathsf{FFT}(2n) \\ &\vdots \end{aligned}$$

Notice that on the m^{th} recursion, we are dividing by the polynomials at depth m on the tree. At this level, there are 2^m separate subtrees that the algorithm is running on, all of which are size $n/2^m$. To complete all the work required at the m^{th} step takes

$$2^m T(n/2^m) \leq 2^{m+1} T(n/2^{m+1}) + 16 \cdot 2^m \mathsf{FFT}(n/2^{m-1}).$$

However, $2^m \mathsf{FFT}(n/2^m) < \mathsf{FFT}(n)$, so we notice that the m^{th} step requires no more than $2^{m+1}T(n/2^{m+1}) + 16 \cdot \mathsf{FFT}(2n)$ work for all 2^m calls.

Therefore, Algorithm 4 takes a total of

$$T(n) = \log_2 n(16\mathsf{FFT}(2n)) + O(n \log n) = 16n \log_2^2 2n + O(n \log n) \in O(n \log^2 n)$$

operations in R.

Recall computing the subproduct tree requires less than $\log_2(n)\mathsf{M}(n)$ operations in the ring R. In total, including precomputations and evaluation, the FastEval requires less than

$$\begin{aligned}
&\log_2 n\mathsf{M}(n) + \log_2 n 16\mathsf{FFT}(2n) + O(n \log n)\\
\leq&\frac{3}{2} n \log_2^2(n) + 16n \log_2^2 2n + O(n \log n)\\
=&\frac{3}{2} n \log_2^2(n) + 16n(\log_2^2 n + 2\log_2 n + 1) + O(n \log n)\\
\leq&\frac{35}{2} n \log_2^2 n + O(n \log n)\\
<&17n \log_2^2 n + O(n \log n) \in O(n \log^2 n)
\end{aligned}$$

operations in R to evaluate a polynomial of degree $n-1$ at n arbitrary points.

2.4.3 Implementation

We implemented the algorithms above in C for 31 bit primes over the field $\mathbb{Z}_p$, where $p = 15 \cdot 2^{27} + 1$. Being clever in our implementation allowed us to save space, leading to a faster, more efficient algorithm.

One way we were able to save space was to notice that the subproduct tree contained only monic polynomials. Since the $M_{i,j}$ are monic on x, we do not need to store the leading term coefficient, 1. The bottom level contains n linear polynomials, and we only need to store the n constants. The next level contains $n/2$ degree 2 polynomials, each containing only two elements that need to be stored, for a total of n elements. Similarly, the two nodes just below the root are both degree $n/2$, and each contain $n/2$ elements to be stored. Hence, the whole tree fits in an array of size $\log_2 n$ by n. Had we stored the monic coefficients, the leaves would have required double the space. This would have lead to an array of size $\log_2 n$ by $2n$.

This improvement was also useful in the computation of the subproduct tree polynomials. Recall Proposition 1.1.2 says that given $a, b \in R[x]$ of degree less than n, the FFT multiplication algorithm generates $c = ab \in R[x]/(x^n - 1)$. While building the subproduct tree, we use FFT multiplications of size n to calculate polynomials of degree n. If

$$ab = c = c_0 + c_1 x + \cdots + c_m x^m$$

for $0 < n < m$, then the FFT of size n maps the coefficient c_ℓ to $(x^\ell \mod x^n)$. We know there will be a c_n which is mapped to $x^n \mod x^n = x^0$. We are left with an answer of degree $n-1$, whose constant is too large by one. We subtract this extra constant, and we expect to have to add x^n. However, since we are not going to store this value anyway, we can save ourselves the time. In the appendix, this FFT procedure has been included as "FFTmultshort". We also save space by recognizing that we do not need to save the root.

The middle product optimization was carefully implemented. The algorithm has a special FFT procedure which is called to compute $y_k b^*$ using the middle product optimization. Notice that we computed

$$y_k \cdot b^* \mod x^{2n} = (1 + c_0) + c_1 \cdot x + \cdots + c_{n-1} \cdot x^{n-1} + \underbrace{m_0 \cdot x^n + \cdots + m_{n-1} \cdot x^{2n-1}}_{x^n m(x)},$$

but what we are actually interested in computing is $(1 - b^* y_k)$. In theory, we need to calculate $y_k \cdot b^*$ as above, extract $m \in R[x]$, and set $y_k b^* = x^n m$. It then remains to calculate $1 - b^* y_k$. The procedure "newtonrec", which is included in the appendix, simply negates the coefficients of m, and discards the rest. This eliminates the work of extracting m and saves space by not requiring it to be copied.

The algorithm requires a temporary array of size $12n$ for computations. This array is used throughout the entire algorithm, including the precomputation. By only allocating space once, we avoid allocating heap space in recursive calls and do not have to worry about freeing space.

2.4.4 Data Results

We randomly generated polynomials of degree $n-1$ for $n = 2^k$, where $k = 5, \ldots, 21$ over the field $\mathbb{Z}_p$, where $p = 15 \cdot 2^{27} + 1$. For each value of n, the polynomial was evaluated at the points $U = [1, 2, \ldots, n]$, first using FastEval, and then using the classic algorithm n times. A higher degree of accuracy was necessary for input values less than 2^{10}.

Comparing Algorithms				
n	New Algorithm	Old Algorithm	Old/New	New/$(n \log_2^2 n)$
2^5	0.000083	0.000035	0.422	1.038×10^{-7}
2^6	0.000219	0.000157	0.717	9.505×10^{-8}
2^7	0.000611	0.000446	0.730	9.742×10^{-8}
2^8	0.001209	0.000950	0.786	7.379×10^{-8}
2^9	0.00284	0.00394	1.393	6.847×10^{-8}
2^{10}	0.00774	0.01612	2.083	7.559×10^{-8}
2^{11}	0.0249	0.0658	2.643	1.005×10^{-7}
2^{12}	0.067	0.269	4.015	1.136×10^{-7}
2^{13}	0.175	1.091	6.234	1.264×10^{-7}
2^{14}	0.444	4.429	9.975	1.383×10^{-7}
2^{15}	1.083	18.103	16.715	1.469×10^{-7}
2^{16}	2.618	73.437	28.051	1.560×10^{-7}
2^{17}	6.204	296.793	47.839	1.638×10^{-7}
2^{18}	14.137	1200.574	84.9242	1.664×10^{-7}
2^{19}	32.358	4874.245	150.635	1.710×10^{-7}
2^{20}	73.654	19750.093	268.147	1.756×10^{-7}
2^{21}	169.008	80266.321	474.926	1.827×10^{-7}

Figure 4: Time in seconds to evaluate a polynomial of degree $n-1$ at n arbitrary points over the field $\mathbb{Z}_p$ for $15 \cdot 2^{27} + 1$.

Figure 4 displays values of n from 2^5 up to 2^{21}, or from 32 to 2097152. The first two columns present the time in seconds that it took to run each algorithm on a randomly generated polynomial of degree $n-1$, at the points $1, \ldots, n-1$. The third column shows how many times slower the old algorithm ran compared to FastEval. The final column reaffirms our complexity discussion by comparing the first column to $n \log_2^2 n$. The time it took to run FastEval divided by $n \log_2^2 n$ seems to approach a constant, approximately 1.8×10^{-7}.

We can see that the time it takes for the old algorithm to run increases four times for every time the number n doubles. On the other hand, the timings for FastEval increase just under three times from 2^{11} to 2^{12}. The rate of change in timings levels out at just under 2.3 at the higher values of n. At 512 evaluation points, the two algorithms ran at practically the same speed. Running FastEval on a polynomial of degree $2^{20} - 1$ took less than 74 seconds, while the old algorithm took a staggering 19750 seconds, or approximately five and a half hours. On a polynomial of degree $2^{21} - 1$, FastEval took just under three minutes, while Horner's method took over 22 hours!

These timings reflect the complexities we expected the algorithm to run for both small and large values of n. The time it took for the evaluation with $n = 2^{20}$ was 2.27 times the time it took for $n = 2^{19}$. Doubling the inputs again, we see that $n = 2^{21}$ took 2.29 times as long as $n = 2^{20}$. Using our time complexity analysis, we predicted that the larger input would take 2.21 times the amount of time to run. The discrepancies in prediction versus practice are small.

It should be noted that the fast algorithms are very useful for large inputs, but are not always the fastest choice for small inputs. For this reason, if $n < 256$, the old quadratic algorithms are being run. This results in similar timings for n small, since most of the work for this example is being done using the same procedures in both cases. In fact, 256 is our break even point, where the fast algorithm actually does begin to perform faster than the classical algorithm.

In addition to comparing the new and old algorithm, it is worth commenting on the difference between the two major components of FastEval. The following figure shows clearly that Multiplying Up (Algorithm 3), is significantly faster than Dividing Down (Algorithm 4). Although both run in $O(n \log^2 n)$, the latter has a much larger coefficient. This is not surprising, as each division requires multiple multiplications.

The first column in the following figure displays the time it took to run Algorithm 3 with inputs n and $1, \ldots, n-1$. The second column shows the time it to to run Algorithm 4 on a randomly generated polynomial of degree $n-1$ with the precomputation done in Algorithm 3. The third column presents the change in time from the first algorithm to the second.

A breakdown of FastEval timings			
n	Multiplying Up	Dividing Down	Dividing/Multiplying
2^5	0.000020	0.000061	3.050000
2^6	0.000032	0.000101	3.156250
2^7	0.000068	0.000215	3.161764
2^8	0.000157	0.000535	3.407643
2^9	0.000512	0.001662	3.246094
2^{10}	0.001498	0.005343	3.566756
2^{11}	0.004061	0.019539	4.811377
2^{12}	0.010454	0.057278	5.479051
2^{13}	0.026055	0.152480	5.852236
2^{14}	0.062933	0.385680	6.128422
2^{15}	0.149907	0.952550	6.354272
2^{16}	0.353182	2.288893	6.480774
2^{17}	0.815458	5.389606	6.609299
2^{18}	1.869353	12.555766	6.716637
2^{19}	4.222501	28.78418	6.816855
2^{20}	9.507624	65.141750	6.851527
2^{21}	21.330361	147.357927	6.908365

Figure 5: A breakdown of time in seconds to evaluate a polynomial of degree $n - 1$ at n points

The data in Figure 5 fails to reinforce our theoretical timings, even though both routines take approximately 2.3 times longer to run when the inputs are doubled. The relative difference between the times in the left column and the times in the right column seems to approach 7. The time to divide down the tree with $n = 2^{21}$ is approximately 6.9 times the number of seconds required to build the subproduct tree. However, our analysis predicted the timings to differ by around 10.6 times. One reason for the difference between our theoretical findings and our actual running times lays in the implementation of the algorithm.

As was previously mentioned, some fast algorithms actually perform slower on small inputs than their quadratic alternative. For this reason, when n is small, we force the code to run quadratic polynomial multiplication and division algorithms. These algorithms both require approximately the same amount of work, and our analysis does not account for this. We assumed that each division would require the work of multiple multiplications.

Figure 5 is presented with greater accuracy than Figure 4. This was done to highlight the speed at which the subproduct tree is built.

Chapter 3

Additional Comments

3.0.1 Addressing our Assumptions

In the beginning of this work, we made a few assumptions. These assumptions allowed us to more easily discuss the algorithm and all its components. It is now time to remove these assumptions and discuss the consequences of their disappearance. Our first, and perhaps largest assumption, was to assume our commutative ring with unity was in fact a field. Suppose we are interested in evaluating a polynomial $f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}, a_i \in R$, where R is a commutative ring with unity. Perhaps R has elements which do not have multiplicative inverses. Another possible situation is that R is in fact a field, but it is a finite field $R = \mathbb{Z}_p$, where 2^k does not divide $p - 1$. Let's suppose R is some commutative ring with unity which does not support the FFT. This means that either R has no primitive nth root of unity, or $2^{-1} \notin R$. One option in this case is to compute the evaluations in a sufficient number of Fourier prime fields, and then use the Chinese Remainder Theorem to determine the true answer in $\mathbb{Z}$. This way, all the work is done over a field. This is particularly nice, as it allows the division algorithm to work as before. If the ring R does not guarantee each element has a multiplicative inverse, we cannot use Newton Iteration, because our initial guess, the inverse of the constant term, may not exist.

If we are working over a ring which does not support the FFT, another option available is to use a different subquadratic multiplication algorithm. One, which is explored in detail in [5], is Karatsuba's multiplication algorithm. This algorithm multiplies polynomials of degree less than $n = 2^k$ over a ring with at most $O(n^{1.59})$ ring operations. While this is a significant increase from $O(n \log^2 n)$ for large n, it is better than $O(n^2)$.

Next, we assumed that we were given $n = 2^k$ unique evaluation points. Since our evaluation map is a homomorphism, insisting on unique evaluation points does not create a loss of information. This is to be expected as evaluating a polynomial at the same points multiple times is redundant.

Suppose n is not a power of two. In general, we have two choices. We may either add "phantom points" to round up to the next power of two, or we may adjust the code so to have an "almost" binary tree. The second option leads to problems with the FFT, as it requires $n = 2^k$. Choosing to add extra points so that the input number is $2^{\lceil \log_2 n \rceil}$ makes the analysis of the algorithm similar, but the run time will be slower. However, we may be able to save work by noticing that the subtree whose leaves are all phantom points can be disregarded. For example, if half the points are phantom points, we need only to consider the subtree rooted at $M_{k-1,0}$.

Lastly, we assumed that $f \in R[x]$ was a polynomial of degree $n-1$. The algorithm is presented as a pair, an evaluation algorithm and an interpolation algorithm. The latter will be discussed in the following section. Therefore, it seems natural to talk about evaluating the polynomial at the number of points needed to interpolate it. However, the reader may choose to use the algorithm for different reasons, and may have a polynomial of degree less than $n-1$. Let $deg(f)$ be the degree of $f(x)$ with respect to x. If $n/2 - 1 < deg(f) < n-1$, then the FFT multiplications still require the same amount of space. This means that the algorithm will still take $O(n \log^2 n)$ operations in the ring. However, if the degree of f is less than half of n, and we wish to evaluate f at n points, it would be more time efficient to run the algorithm twice on $n/2$, than to run it on size n.

3.0.2 Errors to Learn From

As with any project, many mistakes were made, discovered, and corrected along the way. One mistake of note was underestimating the importance of having all subroutines being subquadratic. Because the division algorithm uses a Newton iteration, it is only able to perform divisions when the polynomial dividing has an invertible constant term. Since we are working over a field, all non-zero elements are invertible. Therefore, the division algorithm originally had a check for when the divisor b had a zero constant. When this occurred, the algorithm called a quadratic division algorithm to avoid dividing by zero.

Originally, we tested the algorithm on the evaluation points $0, p-1, p-2, \ldots, p-n+1 \in \mathbb{Z}_p$, for $p = 15 \cdot 2^{27} + 1$. This meant that the very first division divided f by $x(x-1)\cdots(x-n/2)$. This is a large division, which took $O(n^2)$ time. At each level of the algorithm, the quadratic routine would be called once, resulting in at least $n^2 \log_2 n$ arithmetic operations. The data we obtained showed the new algorithm was much faster than Horner's method, but the times for large n were increasing by more than a factor of 3 as n doubled. Running the algorithm on $n = 2^{20}$ took approximately 6 minutes, which was nearly 3.5 times slower than $n = 2^{19}$. Although this was an improvement, it was not as fast as $O(n \log^2 n)$.

Luckily, this problem was easily fixed. Evaluating a polynomial $f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ at $x = 0$ is trivially $f(0) = a_0$. If it is necessary to evaluate f at zero, it can be done by inspection. Therefore, we can require that the input evaluation points are non-zero. By adding in this requirement, and removing the check for zero constants, the algorithm improved to be subquadratic.

3.1 Interpolation and Further Work

Evaluating a polynomial $f \in R[x]$ of degree $n-1$ at n points is especially useful if one wishes to perform operations in the ring R instead of $R[x]$, and then interpolate the results back to $R[x]$. In this section, we will present an overview of an interpolation algorithm which uses work already done in FastEval. Note that interpolation requires f to be evaluated at n distinct points.

Problem (Interpolation)**.** *Suppose R is a commutative ring with unity. Given $n = 2^k$ for some $k \in \mathbb{N}$, and $u_0, \ldots, u_{n-1} \in R$ such that $u_i - u_j$ is a unit for $i \neq j$, and $v_0, \ldots, v_{n-1} \in R$, compute $f \in R[x]$ of degree less than n with*

$$\chi(f) = (f(u_0), \ldots, f(u_{n-1})) = (v_0, \ldots, v_{n-1}).$$

Given distinct $u_0, \ldots, u_{n-1}$ and the arbitrary $v_0, \ldots, v_{n-1}$ in a field F, Lagrange interpolation says that the unique polynomial $f \in F[x]$ which solves the interpolation problem takes the form

$$f = \sum_{0 \le i < n} v_i s_i m/(x - u_i),$$

where $m = (x - u_0) \cdots (x - u_{n-1})$, and

$$s_i = \prod_{j \neq i} \frac{1}{u_i - u_j}.$$

Although this technique requires a field, the condition that $u_i - u_j$ must be a unit allows us to work over a ring.

First, we compute the s_i. To invert and multiply each pair of $u_i - u_j$ would be costly. Instead, we will take a smarter approach. Note that the formal derivative of m is $m' = \sum_{0 \le j < n} m/(x - u_j)$, and $m/(x - u_i)$ vanishes at all points u_j with $i \neq j$. Therefore, we have

$$m'(u_i) = \frac{m}{x - u_i}|_{x = u_i} = \frac{1}{s_i}.$$

Therefore, we can compute all s_i by evaluating m' at the n evaluation points $u_0, \ldots, u_{n-1}$. Luckily, we know we can do this in $O(n \log^2 n)$ operations in R, plus the n inversions.

Once we have computed the s_i, we are already done - we have created a polynomial which solves the interpolation problem. However, the algorithm does not end here. In order to output a polynomial of the form $f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$, for $a_i \in R$, we must compute the products and then take the sum of $f = \sum_{0 \le i < n} v_i s_i m/(x - u_i)$. To do this, we will use our subproduct tree as follows:

Algorithm 6 Interpolation

Input: $n = 2^k$ for some $k \in \mathbb{N}$, $c_i = v_i \cdots_i$ for $i = 0, \dots, n-1$ where s_i is computed as above, $f(u_i) = v_i$ for $u_0, \dots, u_{n-1} \in R$, and the subproducts $M_{i,j}$

Output: $f = \sum_{0 \le i < n} c_i m/(x - u_i) \in R[x]$, where $m = (x - u_0) \cdots (x - u_{n-1})$.

```
1: if $n = 1$ then return: $c_0$
2: call the algorithm with input $r_0 = \sum_{0 \le i < n/2} c_i \frac{M_{k-1,0}}{x-u_i}$
3: call the algorithm with input $r_1 = \sum_{n/2 \le i < n} c_i \frac{M_{k-1,1}}{x-u_i}$
4: compute $M_{k-1,1} r_0 + M_{k-1,0} r_1$
5: return $f$
```

The reader is invited to refer to *Modern Computer Algebra* [5] for a discussion on correctness and run time. It is interesting to note that this algorithm, if the FFT is used, will also require no more than $O(n \log^2 n)$ operations in R. Implementing the interpolation algorithm is the natural next step of this project. This algorithm was coded in Maple, but time did not allow for C code to be completed. Another interesting algorithm which is presented in *Modern Computer Algebra* is a Fast Chinese remaindering algorithm for $R[x]$. This too utilizes a subproduct tree!

3.2 Conclusion

This project was successful in implementing a subquadratic algorithm for evaluating a polynomial of degree less than n, at n arbitrary points. Figure 4 shows the good improvement from the old classic algorithm. The break even point, the size required for FastEval to perform faster than Horner's method, was around 256. This is a nice result for an algorithm requiring $O(n \log^2 n)$ operations.

Bibliography

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

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

[3] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann. The middle product algorithm i. *Appl. Algebra Eng., Commun. Comput.*, 14(6):415–438, March 2004.

[4] Marshall Law and Michael Monagan. A parallel implementation for polynomial multiplication modulo a prime. In *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation*, PASCO '15, pages 78–86, New York, NY, USA, 2015. ACM.

[5] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 2013.

Appendix A

C Code

For the following algorithms, we assume $0 < p < 2^{32}$.

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

Add32s - Compute $a + b \in \mathbb{Z}_p$:

```
int add32s(int a, int b, int p);
```

Sub32s - Compute $a - b \in \mathbb{Z}_p$:

```
int sub32s(int a, int b, int p);
```

Neg32s - Compute $-a \in \mathbb{Z}_p$:

```
int neg32s(int a, int p);
```

SeriesMult - Series multiplication $C = A \cdot f \mod x^n \in \mathbb{Z}_p$:

```
void seriesmult( int n, int *A, int *f, int *C, int p){ //A*f = C to O(x^n)
    int i, k;
    LONG t, M;
    M = ((LONG) p) << 32;
    for(k = n-1 ; k>= 0 ; k--){
        i=0;t=0;
        while(i<k){
            t -= (LONG) A[i]*f[k-i]; i++;
            t -= (LONG) A[i]*f[k-i]; i++;
            t += (t>>63) & M;
        }
        if(i==k){
                t-= (LONG) A[i]*f[k-i];
                t+=(t>>63) & M;
        }
        t = -t;
        t+=(t>>63) & M;
        C[k] = t % p;
    }
    return;
}
```

FFT1 - Forward FFT transform of size n:

```
void FFT1( int *A, int n, int *W, int p){
        int i, n2, t, s, a, s2, temp, temp2;
        n2 = n/2;
        for( s=2; s <= n; s=2*s ){
        if(s==2){
    for(i=0; i < n; i+=s){
                temp = A[i]; temp2 = A[i+1];
                A[i] = add32s(temp,temp2,p);
                A[i+1] = sub32s(temp,temp2,p);
                                        }
                }
    else{
                s2 = s/2;int diff = n-s;
                for(int stemp=0; stemp < n; stemp = s+stemp){
                        for(i=0; i <s2; i++){
                                t = mul32s(W[i+diff], A[stemp+s2+i],p);
                                temp = A[i+stemp];
                                A[i+stemp] = add32s(temp,t,p);
                                A[s2+i+stemp]= sub32s(temp, t, p);
                        }
                        }
                }
        }
        return;
}
```

FFT2 - Backward FFT transform of size n:

```
void FFT2( int *A, int n, int *W, int p){
        int i, n2, t, s, a,s2, temp, temp2;
        n2 = n/2;
        for( s=n; s>1; s=s/2 ){
        if(s==2){
                for(i=0; i < n; i+=s){
                        temp = A[i]; temp2 = A[i+1];
                        A[i] = add32s(temp, temp2,p);
                        A[i+1] = sub32s(temp, temp2,p);
                }
        }
        else{
                s2 = s/2;int diff = n-s;
                for(int stemp=0; stemp < n; stemp = s+stemp){
                    for(i=0; i <s2; i++){
                            t= sub32s(A[i+stemp], A[s2+i+stemp], p);
                            temp = A[i+stemp]; temp2 = A[s2+i+stemp];
                            A[i+stemp] = add32s(temp,temp2,p);
                            A[s2+i+stemp] = mul32s(t,W[i+diff],p);
                            }
                    }
            }
        }
        return;
}
```

Algorithm 2 - FFT multiplication $C = A \cdot B$, where degree of $C < n$:

```
void FFTmult( int *A, int *B, int *C, int n, int da, int db, int * T, int p){
        //C = A*B, or A = A*B, or B = A*B
        //T must be sive 2n, and C must be size n
    int i, ninv, w, winv;
    if(n<256){polmul32s(A,B,C,da,db,p);return;}
    ninv = modinv32s(n,p);
    w = powmod(31, mul32s((p-1), ninv,p),p);
    winv = modinv32s(w,p);
    int *W, *TA, *TB;
    W = C; TA = T; TB = T+n;
    for(i = 0 ; i <= da ; i++){ TA[i] = A[i];}
    for(i = 0 ; i <= db ; i++){ TB[i] = B[i];}
    for(i = da+1 ; i < n ; i++){ TA[i] = 0; }
    for(i = db+1 ; i < n ; i++){ TB[i] = 0; }
    buildW( W, w, n, p);
    FFT2(TA, n, W, p);
    FFT2(TB, n, W, p);
    for(i = 0 ; i < n ; i++){ TA[i] = mul32s(TA[i],TB[i],p);}
    buildW(W,winv,n,p);
    FFT1(TA,n,W,p);
    for(i=0 ; i< n ; i++){ C[i] = mul32s(TA[i],ninv,p);}
}
```

FFTMultShort - FFT multiplication $C = A \cdot B$, where degree of $C \geq n$:

```
void FFTmultshort( int *A, int *B, int *C, int n,
                          int da, int db, int * T, int p){
    //C = A*B, or A = A*B, or B = A*B
    //T must be sive 2n, and C must be size n
    int i, ninv, w, winv;
    if(n<256){polmul32s(A,B,C,da,db,p); return;}
    ninv = modinv32s(n,p);
    w = powmod(31, mul32s((p-1), ninv,p),p);
    winv = modinv32s(w,p);
    int *W, *TA, *TB;
   W = C;TA = T;TB = T+n;
    for(i = 0 ; i <= da ; i++){ TA[i] = A[i];}
    for(i = 0 ; i <= db ; i++){ TB[i] = B[i];}
    for(i = da+1 ; i < n ; i++){ TA[i] = 0; }
    for(i = db+1 ; i < n ; i++){ TB[i] = 0; }
    buildW( W, w, n, p);
    FFT2(TA, n, W, p);
    FFT2(TB, n, W, p);
    for(i = 0 ; i < n ; i++){ TA[i] = mul32s(TA[i],TB[i],p);}
    buildW(W,winv,n,p);
    for(i=0 ; i < n ; i++)TA[i] = sub32s(TA[i], 1, p);
    FFT1(TA,n,W,p);
    for(i=0 ; i< n ; i++){ C[i] = mul32s(TA[i],ninv,p);};
}
```

NegMiddleProduct - Compute $C = y \cdot f$ using the middle product optimization:

```
void negmiddleproduct( int n, int m, int *y, int *f, int *C, int p){
  //y*f into C using middle product optimization
     int i,k; LONG t, M;
    M = ((LONG) p) << 32;
     for(k=m; k<n; k++){
          i=0; t=M;
          while(i<k){
               t-= (LONG) y[i]*f[k-i]; i++;
               t-= (LONG) y[i]*f[k-i]; i++;
               t += (t>>63) & M;
          }
          if(i==k){ t-= (LONG) y[i]*f[k-i]; t+=(t>>63) & M;}
          C[k-m] = t % p;
     }
     return;
}
```

Newtonrec - Compute $y = 1/f to O(x^n)$ using Newton inversion:

```
void newtonrec( int n, int *f, int *y,  int *T, int p){
     // T must be 4n
     int i,m, ninv, w, winv;
     if( n==1 ){ y[0] = modinv32s(f[0], p);   return; }
    m = n/2;
     newtonrec( m, f, y, T, p);
     for(i=m; i<n; i++){ y[i]=0;}
     //negative middle product
     if(n<256){negmiddleproduct(n,m,y,f,T,p); polmul32s(y,T,T,m-1,m,p); }
     else{
          ninv = modinv32s(n,p);
          w = powmod(31, mul32s((p-1), ninv,p),p);
          winv = modinv32s(w,p);
          int *W, *Winv, *TA, *TB;
         W = T; Winv = T+n; TA = T+2*n;TB = T+3*n;
          for(i = 0 ; i < n ; i++){ TA[i] = y[i]; TB[i] = f[i]; }
          buildW( W, w, n, p);
          FFT2(TA, n, W, p);
          FFT2(TB, n, W, p);
          for(i = 0 ; i < n ; i++){ TB[i] = mul32s(TA[i],TB[i],p);}
          buildW(Winv,winv,n,p);
          FFT1(TB,n,Winv,p);
          for(i = 0; i < m ; i++){ TB[i] = neg32s(mul32s(TB[i+m],ninv,p),p);}
          for(i = m; i < n ; i++){ TB[i] = 0;}
          FFT2(TB, n, W, p);
          for(i = 0 ; i < n ; i++){ TB[i] = mul32s(TA[i],TB[i],p);}
          FFT1(TB,n,Winv,p);
          for(i = 0 ; i < n ; i++){ T[i] = mul32s(TB[i],ninv,p);}
     }
     for(i=0; i < n-m; i++) {y[m+i]=T[i];}
     return;
}
```

Division - Compute A/B, and store the remainder in q:

```
int division( int n, int *A, int *B, int *q, int da, int *T, int p) {
     // Computes A/B and stores the remainder and quotient in q
     // Returns the degree of r
     //A deg n-1, B deg n/2, T must be of size 8n
          int i, dq, dr,m, np, db;
          db = n/2;
          if( db<2048 ) {
                  for(i=0; i<=da; i++){T[i] = A[i]; }
                  dr = poldiv32s( T, B, da, db, p );
          for(i=0 ; i <=dr ; i++){
                  q[i] = T[i];
                          return(dr);
          }
     int *TA, *TB, *C;     //Temporary arrays to store the reversed polys
    m = 2*n;
    dq = sub32s(da,db,p);   np = db; dr = db-1;
    C = T; TB = T+2*n ; TA = T+4*n;
     for(i=0; i < 8*n ; i++){ T[i] = 0;}
          for(i=0 ; i<=db ; i++){TB[i] = B[db-i];}
          newtonrec( np, TB, C, TA, p);  //inverts B to O(x^np) and puts it in C
          for(i=0 ; i<np ; i++){ TA[i] = A[da-i];}
          FFTmult(TA,C,TB,n,np-1,np-1,T+6*n,p);    /TA*T (revA*invB) into TB
          for(i=0 ; i<=dq ; i++){q[dq-i] = TB[i];}
          for(i=0 ; i<=dq ; i++){TA[i] = q[i]; }
          for(i=0 ; i<=db ; i++){TB[i] = B[i];}
          FFTmult(TA, TB, C, n, dq, db,T+6*n,p);  // Bq
     for( i = 0 ; i <=da; i++) TA[i] = q[i];
     dr = polsub32s(A, C, TA, da, da, p); //A–Bq = r
     for(i=0 ; i <=da ; i++){ q[i] = TA[i];}
     return(dr);
}
```

Algorithm 3 - Compute the subproduct tree:

```
void subprod(int n, int ** M, int *U, int *T, int p){ //Algorithm 3
    int i, j,m,l,lg,c,k;
    for(i = 0 ; i < n ; i++){ M[0][i]=neg32s(U[i],p); }
    int * temp1 = T + n; int * temp2 = T + 3*n;
    int * temp3 = T + 4*n;
    j = 1;                                    //Degree of poly in row k
    lg = log_2(n);
    for(k=0 ; k < (lg-1) ; k++){          // Row
        l = n/(2*j); c=0;                 // Number of polynomials in row k
        for(m=0 ; m < l ; m++) {
            temp1[j] = 1; temp2[j] = 1;
            for(i = 0 ; i < j ; i++){
                temp1[i] = M[k][i+2*m*j];
                temp2[i] = M[k][i+j+2*m*j];
            }
            FFTmultshort(temp1, temp2, temp3, 2*j, j,j,T, p);
            for(i=0; i < 2*j; i++){ M[k+1][i+c] = temp3[i];}
            c+=2*j;
                }
        j = 2*j;
    }
    return;
}
```

Algorithm 4 - Divide down the subproduct tree:

```
void downtree( int n, int k, int **f, int *T, int *T2, int st , int df,
int **M, int p){
//Algorithm 4
    int i,dr1,dr2, j;
    if (n==1 ){ return;}                  //Base of recursion
    if(df == 0){
        for(i=0 ; i < n ; i++){f[0][st+i] = f[k][st];}
        return;
    }
    for(i = 0 ; i<2*n ; i++){T[i] = 0;}
    T[n/2] = 1;                  //Retrieve polynomials from the precomputed tree
    for(i=st ; i< n/2 + st ; i++){
        T[i-st] = M[k-1][i];
    }
    for(i=0 ; i<8*n ; i++){T2[i] = 0;}
    dr1 =  division(n, f[k]+st, T, f[k-1]+st  , df, T2, p);
    for(i = st ; i < n/2 + st ; i++){T[i-st] = M[k-1][n/2+i];}
    T[n/2] = 1;
    dr2 =  division(n, f[k]+st, T, f[k-1]+(st+n/2), df, T2, p);
    downtree( n/2, k-1, f, T, T2,         st  , n/2-1, M, p);
    downtree( n/2, k-1, f, T, T2, (n/2)+(st), n/2-1, M, p);
}
```