ALGORITHMS FOR COMPUTING CYCLOTOMIC POLYNOMIALS

by

Andrew Arnold

B.Sc., University of British Columbia, 2007

THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN THE DEPARTMENT

OF

MATHEMATICS

SIMON FRASER UNIVERSITY

Spring, 2011

APPROVAL

Name: Andrew Arnold

Degree: Master of Science

Title of thesis: Algorithms for Computing Cyclotomic Polynomials

Examining Committee: **Dr. Stephen Choi**
Associate Professor of Mathematics (Chair) (Chair)

Michael Monagan
Senior Supervisor
Progressor of Mathematics

Peter Borwein
Supervisor
Professor of Mathematics

Petr Lisonek
Internal Examiner
Associate Professor of Mathematics

Date Approved: January 18, 2011

Abstract

The nth cyclotomic polynomial, $\Phi_n(z)$, is the minimal polynomial of the nth primitive roots of unity. We developed and implemented algorithms for calculating $\Phi_n(z)$ to study its coefficients.

The first approach computes $\Phi_n(z)$ using its discrete Fourier transform. The sparse power series (SPS) algorithm calculates $\Phi_n(z)$ as a truncated power series. We make three key improvements to the SPS algorithm, ultimately resulting in a fast recursive variant of the SPS algorithm. We make further optimizations to this recursive SPS algorithm to compute cyclotomic polynomials that require very large amounts of storage. These algorithms were used to find the least n such that $\Phi_n(z)$ has height greater than n^k, for $1 \leq k \leq 7$.

The big prime algorithm quickly computes $\Phi_n(z)$ for n having a large prime divisor p. This algorithm was used to search for families of $\Phi_n(z)$ of height 1.

I dedicate my thesis to my older brother Thomas, whom I will always look up to.

Acknowledgments

I'd like to thank the following people: Colin Beaumont, for lending ample computer processor time; Roman Pearce, for always being a good sounding board; Greg Fee, whose suggestion led to the sparse power series algorithm; and Michael Monagan, for introducing me to this rewarding research project in the first place. I'd also like to thank the CECM populace for tolerating my long-running abuse of computational resources and my parents, without whose support I probably would not have pursued this research.

Table of contents

List of tables

List of algorithms and procedures

List of figures

Chapter 1

Introduction and preliminaries

1.1 Introduction

The nth cyclotomic polynomial, $\Phi_n(z)$ is the minimal polynomial of the nth primitive roots of unity. It is a polynomial over $\mathbb{Z}$ with typically small integer coefficients. We let the order of $\Phi_n(z)$ be the number of distinct odd prime divisors of n. We are interested in the properties of the coefficients of $\Phi_n(z)$ and in particular the height of $\Phi_n(z)$, which we denote by $A(n) \in \mathbb{N}$. There are a number of open problems concerning $A(n)$ that are subject to active research.

One such open problem is to completely characterize flat cyclotomic polynomials. $\Phi_n(z)$ is said to be *flat* if $A(n) = 1$. Cyclotomic polynomials of order 1 are trivially flat and Bang proved that all those of order two are flat as well. Recent efforts by Bachman, Elder, and Kaplan [5, 12, 21] have helped classify infinite families of flat $\Phi_n(z)$ of order three. More recently Kaplan has found an infinite family of flat cyclotomic polynomials of order 4. It remains open, however, whether these families encompass all flat $\Phi_n(z)$ of orders three and four. Moreover, we currently do not know whether there exists a flat cyclotomic polynomial of order five or greater.

We are equally interested in cyclotomic polynomials of large height. Erdős [13] first proved that the heights of cyclotomic polynomials can be arbitrarily large. Erdos proved that, given $c > 0$, that there exist infinitely many n such that $A(n) > n^c$; Maier proved in addition that the set of such $n \in \mathbb{N}$ has positive lower density. We sought to answer: what is the least n for which $A(n) > n$? n^2? n^3? and so forth.

To help try to answer these questions, we developed and implemented a number of fast algorithms to calculate cyclotomic polynomials. The first method, described in Chapter 2, computes $\Phi_n(z)$ through a series of exact polynomial divisions made fast via the fast Fourier transform (FFT). This method was first implemented by Monagan [2]. In Section 2.4 we improve on this method by a constant factor. The sparse power series algorithm, which we describe in Chapter 3, computes $\Phi_n(z)$ as a truncated power series. We make a series of improvments to this algorithm to develop a recursive variant of the sparse power series algorithm, which appears to be our fastest means of computing most cyclotomic polynomials. These algorithms were used namely to study cyclotomic polynomials of large height.

The big prime algorithm, described in Chapter 4, was designed for calculating $\Phi_n(z)$ for n with a large prime divisor p (e.g. $p \approx \sqrt{n}$). This algorithm allows us to compute $A(n)$ without storing the entire set of coefficients of $\Phi_n(z)$, a limitation of our previous algorithms. This algorithm was used to search for $\Phi_n(z)$ of order five that were potentially flat. We, moreover, apply some ideas of the big prime algorithm to make a variation of the sparse power series algorithm which also does not require immediate storage for all the coefficients of $\Phi_n(z)$. This method was used to compute cyclotomic polynomials of large height that may require tens or hundreds of gigabytes to store in entirety. In Chapter 5 we compare our algorithms. We also present some of the cyclotomic polynomial data we have amassed.

1.2 The Möbius function and Euler's ϕ function

We define the number-theoretic functions μ and ϕ, which appear often in our discussion of cyclotomic polynomials.

Definition 1.1. *The Möbius function is the function $\mu : \mathbb{N} \to \{-1, 0, 1\}$ satisfying $\mu(n) = 1$ for squarefree n with an even number of prime factors, $\mu(n) = -1$ for squarefree n with an odd number of prime factors, and $\mu(n) = 0$ for nonsquarefree n.*

E.g. $\mu(6) = 1$. We note that μ is a multiplicative function. That is, if $\gcd(a, b) = 1$, then $\mu(ab) = \mu(a)\mu(b)$.

Definition 1.2. *The totient of a natural number n, $\phi(n)$, is the number of integers j such that $0 \leq j < n$ such that $\gcd(j, n) = 1$.*

Given $n = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}$, an integer with k distinct prime factors $p_1, p_2, \dots, p_k$, it is well known that

$$\phi(n) = \prod_{i=1}^{k} p^{e_k - 1}(p-1) = n \prod_{p|n} (1 - \tfrac{1}{p}), \tag{1.1}$$

where the latter product is over the k distinct primes p dividing n.

1.3 Cyclotomic polynomials

Definition 1.3. *An element ω of a field F is said to be an nth root of unity if $\omega^n = 1$; furthermore, we say such a root of unity is primitive if $\omega^n = 1$ and $\omega^k \neq 1$ for $0 < k < n$.*

That is, the nth primitive roots of unity are the nth roots of unity with multiplicative order n. The nth complex roots of unity in $\mathbb{C}$ are the values

$$e^{2\pi i j/n} \text{ for } 0 \leq j < n; \tag{1.2}$$

the complex primitive roots of unity are exactly those for which $\gcd(j, n) = 1$.

Definition 1.4. *The nth cyclotomic polynomial, $\Phi_n(z) \in \mathbb{C}[z]$, is the monic polynomial whose roots are the nth primitive roots of unity.*

$$\Phi_n(z) = \prod_{\substack{0 \leq j < n \\ \gcd(j,n)=1}} \left(z - e^{2\pi i j/n} \right). \tag{1.3}$$

We could similarly define $\Phi_n(z)$ over any field F containing nth primitive roots of unity. "Cyclotomic" is latin in origin, meaning "circle dividing". Indeed, all the roots of $\Phi_n(z)$ lie on the unit circle. The degree of $\Phi_n(z)$ is the totient of n, $\phi(n)$. As every nth root of unity has multiplicative order d for some unique positive d dividing n, it follows that

$$z^n - 1 = \prod_{d|n} \Phi_d(z), \tag{1.4}$$

where the product is taken over all positive d dividing n.

We let the *index* of $\Phi_n(z)$ be n, and the *order* of $\Phi_n(z)$ be the number of distint odd prime divisors dividing n. In addition we define $\Psi_n(z)$, the nth inverse cyclotomic polynomial.

Definition 1.5. *The nth inverse cyclotomic polynomial, $\Psi_n(z)$, is the polynomial satisfying $\Phi_n(z)\Psi_n(z) = z^n - 1$.*

It is immediate from definitions 1.4 and 1.5 that $\Psi_n(z)$ is the monic polynomial whose $n - \phi(n)$ roots are the nth *non-primitive* roots of unity. We note that both $\Phi_n(z)$ and $\Psi_n(z)$ have no repeated roots, a fact we use in some proofs in this chapter.

$$\Psi_n(z) = \prod_{\substack{0 \le j < n \\ \gcd(j,n) > 1}} \left(z - e^{2\pi i j/n}\right). \tag{1.5}$$

Moree [27] introduced inverse cyclotomic polynomials and was the first to study their properties. As

$$\frac{1}{\Phi_n(z)} = \Psi_n(z)(1 + z^n + z^{2n} + \dots), \tag{1.6}$$

we see that the coefficients of $\Psi_n(z)$ occur in the power series expansion of $1/\Phi_n(z)$.

The first six cyclotomic and inverse cyclotomic polynomials are as follows:

$$\begin{aligned}
&\Phi_1(z) = z - 1, && \Psi_1(z) = 1,\\
&\Phi_2(z) = z + 1, && \Psi_2(z) = z - 1,\\
&\Phi_3(z) = z^2 + z + 1, && \Psi_3(z) = z - 1,\\
&\Phi_4(z) = z^2 + 1, && \Psi_4(z) = z^2 - 1,\\
&\Phi_5(z) = z^4 + z^3 + z^2 + z + 1, && \Psi_5(z) = z - 1,\\
&\Phi_6(z) = z^2 - z + 1, && \Psi_6(z) = z^4 + z^3 - z - 1.
\end{aligned} \tag{1.7}$$

1.3.1 Cyclotomic polynomials of large height

Definition 1.6. *We denote by $A(n)$ and $S(n)$ the height and length of $\Phi_n(z)$, respectively. That is, for $\Phi_n(z) = \sum_{k=0}^{\phi(n)} a_n(k)$,*

$$A(n) = \max_{0 \le k \le \phi(n)} |a_n(k)|, \qquad \textit{and} \qquad S(n) = \sum_{k=0}^{\phi(n)} |a_n(k)|. \tag{1.8}$$

We similarly let $\bar{A}(n)$ and $\bar{S}(n)$ be the height and length of $\Psi_n(z)$.

We see that $A(n) = 1$ for the first six cyclotomic polynomials. The least n for which $A(n) > 1$ is $n = 105$. $A(n) = 2$, as

$$\begin{aligned}\Phi_{105}(z) = z^{48} + z^{47} + z^{46} - z^{43} - z^{42} - \mathbf{2}z^{41} - z^{40} - z^{39} + z^{36} \\ + z^{35} + z^{34} + z^{33} + z^{32} + z^{31} - z^{28} - z^{26} - z^{24} - z^{22} - z^{20} + z^{17} + z^{16} \\ + z^{15} + z^{14} + z^{13} + z^{12} - z^{9} - z^{8} - \mathbf{2}z^{7} - z^{6} - z^{5} + z^{2} + z^{1} + 1.\end{aligned}$$

Paul Erdős [13] proved that $A(n)$ is not bounded by n^c for any $c > 0$; however, his proof does not suggest, given c, exactly how great must n be for $A(n)$ to be greater than n^c? This was the original motivating question that spurred the author and Monagan to develop and implement algorithms to compute cyclotomic polynomials. We were interested in studying the growth of $\max_{m \leq n} A(m)$ with respect to n. $\Phi_n(z)$ of small order and index typically have small coefficients. These, unfortunately, are exactly the cyclotomic polynomials that are easy to compute. Koshiba [24] computed the first ten percent of the terms of $\Phi_n(z)$ for $n = 111546435$ and in doing so showed that $A(n) > n$, the first such example to our knowledge. Monagan [2] found $A(1181895) = 14102773$, and verified that this was the least n for which $A(n) > n$. Monagan, moreover, found examples of n satisfying $A(n) > n^2$ and $A(n) > n^4$. He found such examples by computing $\Phi_n(z)$ for n which were divisible by 1181895. By implementing new algorithms we have since shown that Monagan computed the least n for which $A(n) > n^2$ and $A(n) > n^4$ respectively. We, moreover, found the least n for which $A(n) > n^k$ for $k = 3, 5, 6,$ and 7.

Table 1.1: The least n for which $A(n) > n^c$

c	n	$A(n)$
1	1181895	14102773
2	43730115	862550638890874931
3	416690995	80103182105128365570406901971
4	1880394945	64540997036010911566826446181523888971563
5,6	17917712785	$\approx 8.103388 \cdot 10^{63}$
7	99660932085	$\approx 6.126721 \cdot 10^{87}$

1.3.2 Flat cyclotomic polynomials

$\Phi_n(z)$ is said to be *flat* if $A(n) = 1$. All cyclotomic polynomials of order 1 or 2 are flat; however, this not true in general for $\Phi_n(z)$ of higher order. The first non-flat cyclotomic polynomial, $\Phi_{105}(z)$, is in fact the first of order at least three, as $105 = 3 \cdot 5 \cdot 7$. Bachman [5], Kaplan [21], and more recently Elder [12] have found infinite families of flat $\Phi_n(z)$ of order three. Noe computed numerous flat $\Phi_n(z)$ of order 4 [30]. Kaplan [22] first constructed an infinite family of flat $\Phi_n(z)$ of order four. Elder [12] has since found a broader such family. Whether this family encompasses all flat cyclotomic polynomials of order four is unknown. We do not yet know, however, whether there exist *any* flat cyclotomic polynomials of order five or greater. For $\Phi_n(z)$ of order five with $n < 6.5 \cdot 10^8$, $A(n)$ is at least 4.

The author searched for flat cyclotomic polynomials of order five amongst candidate $\Phi_n(z)$ for n of the form $n = p_1p_2p_3p_4p_5$, a product of five odd primes satisfying $p_k \equiv \pm 1 \bmod \prod_{i=1}^{k-1} p_i$, for $1 < k \leq 5$. The smallest such n is

$$n = 746,443,728,915 = 3 \cdot 5 \cdot 31 \cdot 929 \cdot 1727939, \tag{1.9}$$

for which the degree of $\Phi_n(z)$ is $\phi(n) = 384,846,351,360$. Even storing the coefficients of $\Phi_n(z)$ to any precision, in memory, is not possible with most modern computers. The big prime algorithm, described in detail in Chapter 4, allows us to compute $A(n)$ in a manner which does not require us to store all the coefficients of $\Phi_n(z)$ at once. With this algorithm, we were able to find hundreds of examples of $\Phi_n(z)$ of order five and height 2. We were unable, however, to find $\Phi_n(z)$ of order 5 and height 1.

1.3.3 Fundamental properties of cyclotomic polynomials

We see in (1.7) that $\Phi_n(z)$ and $\Psi_n(z)$ are in $\mathbb{Z}[z]$ for $n \leq 6$. It is not immediate from their definitions whether this is true for all n. We show that it is.

Theorem 1.7. *$\Phi_n(z)$ is in $\mathbb{Z}[z]$.*

Before a proof of Theorem 1.7, we need to define the content of a polynomial.

Definition 1.8. *Let $f(z) = a_kz^k + \cdots + a_1z + a_0$ be a polynomial of degree k in $\mathbb{Z}[z]$. The content of f, denoted by $\mathrm{cont}(f)$, is defined as*

$$\mathrm{cont}(f) = \gcd(a_0, a_1, \ldots, a_k). \tag{1.10}$$

In other words, $\mathrm{cont}(f)$ is the greatest integer l for which f/l is over the integers. Gauss first proved the following result about content, which we will use in the proof of 1.7. We refer to [31] for a proof of Lemma 1.9.

Lemma 1.9 (Gauss's lemma)**.** *Let* $f, g \in \mathbb{Z}[z]$. *Then* $\mathrm{cont}(fg) = \mathrm{cont}(f)\mathrm{cont}(g)$.

We also use the following theorem.

Theorem 1.10. *Let* F *be a field. Then* $F[z]$ *forms a Euclidean domain with the valuation function* $v : F[z] \setminus \{0\} \to \mathbb{Z}_{\geq 0}$ *defined by* $v(f(z)) = \deg(f)$.

In other words, if F is a field and $f(z), g(z) \in F[z]$ are nonzero polynomials, then there exist $Q(z)$ and $R(z)$ in $F[z]$ such that $f(z) = Q(z)g(z) + R(z)$ and either $R(z) = 0$ or $\deg(R) < \deg(f)$. We refer to [25] for a proof.

Proof of Theorem 1.7. Towards a contradiction let n be the least natural number for which $\Phi_n(z)$ is not in $\mathbb{Z}[z]$. As $\Phi_1(z) = z - 1$, n exceeds 1; moreover, $f(z) = \prod_{d|n, d<n} \Phi_d(z)$ is a product of polynomials over $\mathbb{Z}$ and is hence over $\mathbb{Z}$ itself. By (1.4),

$$\Phi_n(z) = \frac{z^n - 1}{f(z)}, \tag{1.11}$$

$f(z)$ divides $z^n - 1$ exactly as polynomials over $\mathbb{C}$. Suppose then that $\Phi_n(z)$ is not over $\mathbb{Q}$ in addition to $\mathbb{Z}$. In which case, as $\mathbb{Q}[z]$ is a Euclidean domain by Theorem 1.10, there exist $Q(z)$, $R(z) \neq 0$ in $\mathbb{Q}[z]$ satisfying

$$z^n - 1 = Q(z)f(z) + R(z) \tag{1.12}$$

and $\deg(R) < \deg(f)$. However, as $f(z)\Phi_n(z) = z^n - 1$ as well, we have

$$f(z)(\Phi_n(z) - Q(z)) = R(z), \tag{1.13}$$

contradicting that $\deg(R) < \deg(f)$. Thus $\Phi_n(z) \in \mathbb{Q}[z]$, and we can write

$$\Phi_n(z) = \sum_{i=0}^{\phi(n)} \tfrac{a_i}{b_i} z^i, \tag{1.14}$$

where a_i and b_i are coprime integers. We can multiply out the denominators. Let M be the least common multiple of the integers b_i, for $0 \leq i \leq \phi(n)$. Then $M\Phi_n(z)$ is a polynomial

whose content is the greatest common divisor of a_i, $0 \leq i \leq \phi(n)$. As, by hypothesis, $\Phi_n(z)$ does not have all integer coefficients, M must be greater than 1. Thus, as a_i and b_i are coprime, M does not divide a_i, i.e. $\mathrm{cont}(M\Phi_n(z)) \neq M$. Moreover, as $f(z)$ is a product of cyclotomic polynomials, it is necessarily monic and $\mathrm{cont}(f) = 1$. By Gauss's lemma, as $(M\Phi_n(z))f(z) = Mz^n - M$,

$$\mathrm{cont}(M\Phi_n(z))\mathrm{cont}(f) = \mathrm{cont}(Mz^n - M) = M, \tag{1.15}$$

a contradiction. □

From the definition of $\Psi_n(z)$ we have that the $f(z)$ appearing in the proof of Theorem 1.7 is exactly $\Psi_n(z)$, giving the identity

$$\Psi_n(z) = \prod_{d|n, 0<d<n} \Phi_d(z), \tag{1.16}$$

hence the following corollary:

Corollary 1.11. *$\Psi_n(z)$ is in $\mathbb{Z}[z]$.*

One can check that the $\Phi_n(z)$ appearing in 1.7 are irreducible over $\mathbb{Q}$. This holds in general. We cite this result as a theorem, and refer to [15] for a proof.

Theorem 1.12 (Garling [15])**.** *$\Phi_n(z)$ is irreducible over $\mathbb{Q}$.*

1.3.4 The palindromic property of the cyclotomic coefficients

Lemma 1.13. *Let $n > 1$ and let $a_n(k)$ and $c_n(k)$ be the coefficients of the z^k zerm of $\Phi_n(z)$ and $\Psi_n(z)$ respectively, so that*

$$\Phi_n(z) = \sum_{k=0}^{\phi(n)} a_n(k)z^k, \qquad \textit{and} \qquad \Psi_n(z) = \sum_{k=0}^{n-\phi(n)} c_n(k)z^k. \tag{1.17}$$

Then $a_n(k) = a_n(\phi(n) - k)$ and $c_n(k) = -c_n(n - \phi(n) - k)$.

To prove Lemma 1.13, we use a complete characterization of $a_n(0) = \Phi_n(0)$.

Lemma 1.14.

$$\Phi_n(0) = \begin{cases} -1: & \textit{if } n = 1 \\ 1: & \textit{otherwise} \end{cases} \tag{1.18}$$

Proof. Our proof is by strong induction on n. Certainly as $\Phi_1(z) = z-1$, $\Phi_1(0) = 0-1 = 1$. Fix $n > 0$ and suppose, for $1 \leq d < n$, that (1.18) holds for $\Phi_d(0)$. Evaluating both sides of the identity $z^n - 1 = \prod_{d|n} \Phi_d(z)$ at $z = 0$, we have

$$\begin{aligned} -1 &= \prod_{d|n} \Phi_d(0) \\ &= \Phi_1(0)\Phi_n(0) \prod_{d|n, 1<d<n} \Phi_d(0) \\ &= -\Phi_n(0) \prod_{d|n, 1<d<n} \Phi_d(0). \end{aligned} \tag{1.19}$$

By our induction hypothesis, $\Phi_d(0) = 1$ for any $d|n$, $1 < d < n$. It follows that $\Phi_n(0) = 1$. □

Proof of Lemma 1.13. An nth primitive (non-primitive) root of unity ω is such if and only if ω^{-1} is as well. Thus

$$f(z) = z^{\phi(n)}\Phi_n(1/z) = \sum_{k=0}^{\phi(n)} a_n(\phi(n) - k)z^k \tag{1.20}$$

is a polynomial of degree $\phi(n)$ whose $\phi(n)$ roots are exactly the roots of $\Phi_n(z)$. Thus $f(z) = c\Phi_n(z)$ for some constant c. By a comparison of their leading coefficients, we see that $f(z) = (a_n(0)/a_n(\phi(n))) \cdot \Phi_n(z)$. By Lemma 1.14, $a_n(0) = 1$ for $n > 1$. Thus, as $\Phi_n(z)$ is monic, it follows that $f(z) = \Phi_n(z)$, and $a_n(k) = a_n(\phi(n) - k)$.

By an analogous argument,

$$g(z) = z^{n-\phi(n)}\Psi_n(1/z) = \sum_{k=0}^{\phi(n)} c_n(n-\phi(n)-k)z^k = (c_n(0)/c_n(n-\phi(n))) \cdot \Psi_n(z). \tag{1.21}$$

Given that

$$\Phi_n(z)\Psi_n(z) = \left(\sum_{k=0}^{\phi(n)} a_n(k)z^k \right) \cdot \left(\sum_{k=0}^{n-\phi(n)-k} c_n(k)z^k \right) = z^n - 1, \tag{1.22}$$

we thus have that $a_n(0)c_n(0) = -1$ and $c_n(0) = -1$. $\Psi_n(z)$ is monic, thus $c_n(n-\phi(n)) = 1$ and $g(z) = -\Psi_n(z)$. It follows that $c_n(k) = c_n(n - \phi(n) - k)$. □

We say that the coefficients of $\Phi_n(z)$ are *palindromic* and the coefficients of $\Psi_n(z)$ are *antipalindromic*. Note that Lemma 1.13 does not hold for $\Phi_1(z) = z - 1$, which is antipalindromic and $\Psi_n(z) = 1$, which is trivially palindromic. This is because $n = 1$ is the only case such that the constant coefficient of $\Phi_n(z)$ is -1 and that of $\Psi_n(z)$ is 1. We can generalize Lemma 1.13 to products of cyclotomic polynomials, which will prove useful later.

Lemma 1.15. *Let*

$$f(z) = \Phi_{n_1}(z)\Phi_{n_2}(z)\cdots\Phi_{n_k}(z) = \sum_{i=0}^{D} c(i)z^i \tag{1.23}$$

be a product of cyclotomic polynomials such that n_j is odd for $1 \leq j < k$. Then $c(i) = (-1)^D c(D-i)$ for $0 \leq i < D$. In other words, if D is odd $f(z)$ is antipalindromic, and if D is even, $f(z)$ is palindromic.

Proof. Since $\Phi_n(z)$ is monic, $f(z)$ is monic. As before, we observe that if ω is a root of f, then ω^{-1} is too. Set

$$g(z) = z^D f(z^{-1}) = \sum_{i=0}^{D} c(D-i)z^i. \tag{1.24}$$

By an argument similar to that in the proof of Lemma 1.13, $g(z)$ is a polynomial of degree D with leading coefficient $c(0)$ whose D roots are the roots of f. Thus $f(z)$ and $g(z)$ only differ by a constant factor. We need only resolve $c(0)$. To that end, we observe that $\phi(n)$ is even for odd $n > 1$, and $\phi(1) = 1$. Thus $r \equiv D \pmod 2$, where r is the cardinality of

$$\{j : 1 \leq j \leq k \text{ and } n_j = 1\}. \tag{1.25}$$

Note that the constant term of f, $c(0)$, is the product of the constant terms of the $\Phi_{n_j}(z)$ in (1.23). Since the constant term of $\Phi_1(z) = z - 1$ is -1, and by Lemma 1.14, the constant term of $\Phi_n(z)$ is 1 for $n > 1$, we have that $c(0) = (-1)^r = (-1)^D$. Hence if D is even, $c(0) = 1$, and $g(z)$ is monic and must equal $f(z)$. It follows from (1.24) that $c(i) = c(D-i)$. If D is odd, $c(0) = -1$ and so $f(z) = -g(z)$. In which case, $c(i) = -c(D-i)$. □

1.3.5 A naive algorithm for computing $\Phi_n(z)$

In this section we describe a basic algorithm for computing $\Phi_n(z)$. To that end, we prove the following elementary result, which will be useful in the proof of a few cyclotomic polynomial identities thereafter.

Lemma 1.16. *Let ω be an nth primitive root of unity and let $k > 0$ and $m = n/\gcd(k, n)$. Then ω^k is an mth primitive root of unity.*

Proof. If ω is an nth primitive root of unity, then $\omega^j = 1$ if and only if $n|j$. Let $d = \gcd(k, n)$, in which case $(\omega^k)^m = \omega^{k(n/d)} = (\omega^n)^{k/d} = 1$. Thus ω^k is an mth root of unity. It remains to be shown that ω^k is, in addition, primitive. Suppose then that $0 < j < m = n/d$ and $(\omega^k)^j = 1$. In which case $n|jk$, hence $\frac{n}{d}|j\frac{k}{d}$ and, as $\frac{n}{d}$ and $\frac{k}{d}$ are coprime, this in turn implies that n/d divides j, contradicting our choice of j. □

For the particular case where k is prime, we have the following corollary.

Corollary 1.17. *Let p, q be primes such that $p \nmid n$ and $q|n$. Let ω be an nth primitive root of unity. Then ω^p is an nth primitive root of unity, and ω^q is an (n/q)th primitive root of unity.*

The following two lemmas allow us to compute $\Phi_n(z)$ by recursion on the factors of its index n.

Lemma 1.18. *Let $q|n$ be prime. Then*

$$\Phi_{nq}(z) = \Phi_n(z^q), \text{ and} \tag{1.26}$$

$$\Psi_{nq}(z) = \Psi_n(z^q). \tag{1.27}$$

E.g. $\Phi_4(z) = \Phi_2(z^2) = z^2 + 1$.

Lemma 1.19. *Let p be a prime that does not divide n, then*

$$\Phi_{np}(z)\Phi_n(z) = \Phi_n(z^p), \text{ and} \tag{1.28}$$

$$\Psi_{np}(z) = \Psi_n(z^p)\Phi_n(z). \tag{1.29}$$

E.g. $\Phi_6(z)\Phi_2(z) = (z^2 - z + 1)(z + 1) = z^3 + 1 = \Phi_2(z^3)$.

Proof of Lemmas 1.18 and 1.19. As $\Phi_{nq}(z)\Psi_{nq}(z) = \Phi_n(z^q)\Psi_n(z^q) = z^{nq} - 1$, (1.27) follows from (1.26). Equation (1.29) similarly follows from (1.28). We will prove (1.26) and (1.28) by equating the complex roots of both sides of the respective equations.

Suppose $q|n$ is prime and ω is a root of $\Phi_{nq}(z)$. In which case, by Corollary 1.17, ω^q is a primtive nth root of unity, and thus ω is a root of $\Phi_n(z^q)$. As $q|n$, the degree of $\Phi_{nq}(z) = \phi(nq) = \phi(n)q$ is exactly the degree of $\Phi_n(z^q)$, and so the roots of $\Phi_{nq}(z)$, all of multiplicity 1, are exactly the roots of $\Phi_n(z^q)$. It follows that $\Phi_{nq}(z) = c\Phi_n(z^q)$ for some constant factor c. Since both $\Phi_n(z^q)$ and $\Phi_{nq}(z)$ are monic, they are necessarily equal.

Now suppose $p \nmid n$ is prime and ω is a root of $\Phi_{np}(z)\Phi_n(z)$. If ω is a root of $\Phi_{np}(z)$, then ω^p is an nth primitive root of unity by Corollary 1.17 and thus is also a root of $\Phi_n(z^p)$. If ω is a root of $\Phi_n(z)$ then again by corollary 1.17 it holds that ω^p also has multiplicative order n. In either case ω is a root of $\Phi_n(z)$. As $p \nmid n$, $\phi(np)+\phi(n) = \phi(n)(p-1)+\phi(n) = \phi(n)\cdot p$, and so the degrees of $\Phi_{np}(z)\Phi_n(z)$ and $\Phi_n(z^p)$ are equal. Thus the roots of $\Phi_{np}(z)\Phi_n(z)$, all distinct, are the roots of $\Phi_n(z^p)$ and the polynomials, both monic, must be equal. □

We oftentimes express (1.28) of Lemma 1.19 as

$$\Phi_{np}(z) = \frac{\Phi_n(z^p)}{\Phi_n(z)} = \frac{\Phi_n(z^p)\Psi_n(z)}{z^n - 1}. \tag{1.30}$$

Immediately we have that from Lemma 1.19 that for any prime p,

$$\Phi_p(z) = \frac{\Phi_1(z^p)}{\Phi_1(z)} = \frac{z^p - 1}{z - 1} = z^{p-1} + z^{p-2} + \cdots + z + 1. \tag{1.31}$$

By lemma 1.18, if q is a prime divisor of n, then the coefficients of $\Phi_{nq}(z)$ satisfy

$$a_{nq}(k) = \begin{cases} a_n(k/q) & \text{if } q|k, \\ 0 & \text{otherwise.} \end{cases} \tag{1.32}$$

In particular $A(nq) = A(n)$ and $S(nq) = S(n)$. Lemma 1.20 gives a similar result for $\Phi_n(z)$ of even index.

Lemma 1.20. *Let $n > 1$ be odd. Then $\Phi_{2n}(z) = \Phi_n(-z)$.*

E.g. $\Phi_{10}(z) = z^4 - z^3 + z^2 - z + 1 = (-z)^4 + (-z)^3 + (-z)^2 + (-z) + 1 = \Phi_5(-z)$.

Proof. Suppose ω is such that $-\omega$ is an nth root of unity, that is, ω is a root of $\Phi_n(-z)$. We will show ω is a root of $\Phi_{2n}(z)$. As n is odd, $\omega^n = -(-\omega)^n = -1$, and ω is not an nth root of unity. Clearly $\omega^{2n} = (-1)^2 = 1$ so ω is a $(2n)$th root of unity; it remains to show that ω is primitive.

If $(-\omega)^k = -1$ for some k, then $(-\omega)^{2k} = 1$, and $n|2k$. However, as n is odd, then if n divides $2k$ then n divides k and $(-\omega)^k = (-\omega^n)^{k/n} = 1$, a contradiction. Thus $(-\omega)^k \neq -1$ for any integer k. As $\omega^k = \pm(-\omega)^k$, it follows that if $(-\omega)^k \neq 1$, then $\omega^k \neq 1$, and thus ω is a $(2n)$th primitive root of unity.

As $\Phi_n(-z)$ and $\Phi_{2n}(z)$ have the same degree, the roots of $\Phi_{2n}(z)$ are exactly the roots of $\Phi_n(-z)$. Since n is odd and greater than 1, $\phi(n)$ is even and hence $\Phi_n(-z)$ is monic. Thus $\Phi_{2n}(z) = \Phi_n(-z)$. □

By Lemma 1.20, if n is odd, $A(2n) = A(n)$ and $S(2n) = S(n)$. Lemmas 1.18−1.20 outline a means of computing $\Phi_n(z)$ from the first cyclotomic polynomial $\Phi_1(z) = z - 1$. For instance, we could compute $\Phi_n(z)$ for $n = 150 = 2 \cdot 3 \cdot 5^2$ in the following manner:

$$\begin{aligned}
\Phi_3(z) &= \Phi_1(z^3)/\Phi_1(z) = (z^3 - 1)/(z - 1) = z^2 + z + 1, \text{ and} \\
\Phi_{15}(z) &= \Phi_3(z^5)/\Phi_3(z) = (z^{10} + z^5 + 1)/(z^2 + z + 1) \\
&= z^8 - z^7 + z^5 - z^4 + z^3 - z + 1, \text{ by Lemma 1.19.} \\
\Phi_{75}(z) &= \Phi_{15}(z^5) \text{ by Lemma 1.18,} \\
&= z^{40} - z^{35} + z^{25} - z^{20} + z^{15} - z^5 + 1. \\
\Phi_{150}(z) &= \Phi_{75}(-z) = z^{40} + z^{35} - z^{25} - z^{20} - z^{15} + z^5 + 1 \text{ by Lemma 1.20.}
\end{aligned}$$

Algorithm 1.1 describes this approach.

Algorithm 1.1 is well-known and until recently was used by most computer algebra systems. For example, in Maple 13 and in prior versions it is used by the `cyclotomic` command available in the `numtheory` package. It is clear from algorithm 1.1 that it is easy to compute $\Phi_n(z)$ given $\Phi_m(z)$, where m is the largest odd, squarefree divisor of n. As such, unless otherwise specified we only consider $\Phi_n(z)$ of squarefree, odd index throughout the remainder of this thesis. The brunt of the work in Algorithm 1.1 takes place in the polynomial divisions on line 3. Using classical, quadratic polynomial division and arbitrary-precision integers to perform polynomial division is much too slow to compute

Algorithm 1.1: A naive algorithm for computing $\Phi_n(z)$ by repeated polynomial division

Input: $n = 2^{e_0} p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, where $k \geq 0$; $2 < p_1 < \cdots < p_k$ are prime; $e_0 \geq 0$; and $e_i > 0$ for $1 \leq i \leq k$

Output: $\Phi_n(z)$

1 $m \longleftarrow 1$, $\Phi_m(z) \longleftarrow z - 1$

2 **for** $i = 1$ **to** k **do**

3 $\quad \Phi_{mp_i}(z) \longleftarrow \frac{\Phi_m(z^{p_i})}{\Phi_m(z)}, m \longleftarrow m \cdot p_i$ `// Lemma 1.19`

`//` m `is now the largest odd squarefree divisor of` n

4 **if** $2|n$ **then** $\Phi_{2m}(z) \longleftarrow \Phi_m(-z), m \longleftarrow 2m$ `// Lemma 1.20`

`//` m `is now the largest squarefree divisor of` n

5 $s \longleftarrow n/m$, $\Phi_n(z) \longleftarrow \Phi_m(z^s)$ `// Lemma 1.18`

return $\Phi_n(z)$

$\Phi_n(z)$ with n in the millions. One obvious way to perform this division step faster is by way of the discrete fast Fourier transform (FFT), which we describe in Section 2.1.

1.4 Implementing modular arithmetic

In many of our implementations of the algorithms described in this thesis we compute $\Phi_n(z)$ modulo a prime q. How we implement modular arithmetic is thus very important to the performance of our algorithms.

Reducing integers modulo q (i.e. the "`%`" binary operator in C code) is expensive compared to other integer arithmetic operations on a modern computer processor. Typically, for the purposes of our arithemetic, we store an integer modulo a prime q in the nonnegative range $[0, q)$. If $0 \leq u, v < q$, then $0 \leq u + v < 2q$ and $-q < u - v < q$. Thus to reduce $u + v$ modulo q, it suffices to take $u + v - q$ if $u + v$ exceeds q. Similarly, to reduce $u \geq v$ then $u - v \bmod q$ is exactly $u - v$, otherwise it is $u - v + q$. Thus both addition and subtraction in $\mathbb{Z}_q$ require at most one integer addition, one subtraction and one comparison. We prefer that $2q$ can fit in one machine word (i.e. $q < 2^{31}$ or $q < 2^{63}$ depending on the architecture) such that a sum $u + v$ can fit in a machine word before we reduce it modulo q.

Algorithm 1.2: Multiplication modulo a 42-bit prime

Input: $u = [u_{41}, u_{40}, \ldots, u_1, u_0]_2, v = [v_{41}, v_{40}, \ldots, v_1, v_0]_2$, two 42-bit integers modulo a 42-bit prime q (i.e. $q < 2^{42}$)

Output: $w = uv \bmod q$

// U_0 and V_0 are obtained via bitmask operations

1 $U_0 \longleftarrow [u_{20}, u_{19}, \ldots, u_0]_2$ // $U_0 = u \mod 2^{21}$

2 $V_0 \longleftarrow [v_{20}, v_{19}, \ldots, v_0]_2$ // $V_0 = v \mod 2^{21}$

// U_1 and V_1 are obtained by bitshift operations

3 $U_1 \longleftarrow [u_{41}, u_{40}, \ldots, u_{21}]_2$ // $U_1 = (u - U_0)/2^{21}$

4 $V_1 \longleftarrow [v_{41}, v_{40}, \ldots, v_{21}]_2$ // $V_1 = (v - V_0)/2^{21}$

5 $w \longleftarrow U_1 V_1 2^{21} \bmod q$, $w \longleftarrow w + U_1 V_0$, **if** $w > q$ **then** $w \longleftarrow w - q$

6 $w \longleftarrow w + U_0 V_1$, **if** $w > q$ **then** $w \longleftarrow w - q$

7 $w \longleftarrow w \cdot 2^{21} \bmod q$, $w \longleftarrow w + U_0 V_0$, **if** $w > q$ **then** $w \longleftarrow w - q$

We first implemented multiplication in $\mathbb{Z}_q$ for q a 42-bit prime, using 64-bit machine precision integers. Using only 64-bit integers, this is nontrivial as given $0 \leq u, v < q$, their product uv, before reduction modulo q, potentially requires more than 64-bits of storage. To circumvent this problem, we break u and v into sums of the form

$$u = U_1 \cdot 2^{21} + U_0, \qquad v = V_1 \cdot 2^{21} + V_0, \tag{1.33}$$

where $0 \leq U_0, U_1, V_0, V_1 < 2^{21}$, and then multiply the respective components of (1.33), reducing modulo q where necessary to avoid 64-bit overflow. Algorithm 7 describes this implementation of multiplication modulo a 42-bit prime. In our notation, we let "$[\cdots]_2$" to denote an integer in binary representation. That is, given $u_0, \ldots, u_k \in \{0, 1\}$,

$$[u_k, u_{k-1}, \ldots, u_0]_2 = \sum_{i=0}^{k} u_k 2^k. \tag{1.34}$$

It should be noted that, given a binary representation of u, division by a power of 2 is easy. If we wanted to compute a quotient Q and remainder R where $u = 2^s Q + R$, and $0 \leq R < 2^s$, then the binary representations of Q and R are merely

$$Q = [u_k, \ldots, u_s]_2 = \sum_{i=0}^{k-s} u_{i+s} 2^i \quad \text{and} \quad R = [u_{s-1}, \ldots, u_0]_2 = \sum_{i=0}^{s-1} u_i 2^i \tag{1.35}$$

Algorithm 1.3: Dividing a two-word integer by a one-word integer. (Granlund & Moller [26])

Input: $d < 2^{64}$, and $\bar{d} = \lfloor (2^{128} - 1)/d \rfloor - 2^{64}$; $u = U_1 2^{64} + U_0$, where $0 \leq U_0, U_1 < 2^{64}$ and $U_1 < d$

Output: The quotient Q and remainder R satisfying $u = dQ + R$ and $R < d$

// Q can be stored as a one-word integer

1 $Q \longleftarrow \lfloor \bar{d} U_1 / 2^{64} \rfloor + U_1$

// $R = u - Qd$ may be a two-word integer at this stage

2 $R \longleftarrow u - Qd$

// we iterate at most 3 times

3 **while** $R > d$ **do** $Q \leftarrow Q + 1, \quad R \leftarrow R - d$

return Q, R // Q, R are now both single-precision

respectively. Both Q and R can be computed by way of bit-mask operations (i.e. the "&" operator in C code), which is faster than a division operation. Algorithm 7 requires two division operations. Using assembly code, however, we can instead take two 64-bit integers, $0 \leq u, v < 2^{64}$, compute their product uv and store it as a 128-bit integer in two 64-bit machine words, and then divide that product by q and take the remainder $uv \bmod q$. This method works for primes q as large as $2^{64} - 1$.

Granlund and Montgomery [17] developed a division-free method of dividing by a fixed integer on a computer processor using multiplication operations. This algorithm is well-known and is implemented, for instance, in the GMP multi-precision arithmetic library. Granlund and Moller [26] improved this method for the specific case of dividing a two-word (128-bit) integer by a fixed, one-word integer. In our case, we are dividing by the fixed integer q. We explain here a simpler, previous variant of their new algorithm, also described in their paper. Suppose we want to divide an integer $u = U_1 2^{64} + U_0$, where $0 \leq U_0, U_1 < 2^{64}$, by a 64-bit fixed divisor d satisfying $2^{63} < d < 2^{64}$. Suppose, moreover, that $U_1 < d$, such that the quotient $Q = \lfloor u/d \rfloor < 2^{64}$, so that both the quotient and remainder are one-word integers. We precompute

$$\bar{d} = \left\lfloor \frac{2^{128} - 1}{d} \right\rfloor - 2^{64}. \tag{1.36}$$

$\bar{d}$ is called the *reciprocal* of d. We approximate $Q = \lfloor u/d \rfloor$ by

$$Q' = \left\lfloor \frac{\bar{d}U_1}{2^{64}} \right\rfloor + U_1. \tag{1.37}$$

It can be shown that Q' will be fairly close to our real quotient. Let k be the integer satisfying

$$\left\lfloor \frac{2^{128}-1}{d} \right\rfloor d = 2^{128} - k. \tag{1.38}$$

We must have that $1 \leq k \leq d < 2^{64}$. Moreover, by (1.36) we have that

$$k = 2^{128} - (2^{64} + \bar{d})d. \tag{1.39}$$

If we let $R' = u - Q'd$ be the "remainder" corresponding to our approximate quotient Q', then

$$\begin{aligned} R' &= U_0 + U_1 2^{64} - \left(\left\lfloor \frac{\bar{d}U_1}{2^{64}} \right\rfloor - U_1 \right) d, \\ &= U_0 + U_1 2^{64} - \left(\frac{\bar{d}U_1 - \lambda}{2^{64}} - U_1 \right) d, \qquad \text{where } \lambda = \bar{d}U_1 \bmod 2^{64}, \\ &= U_0 + \Big(U_1(2^{128} - \bar{d}d - 2^{64}d) - \lambda d \Big)/2^{64}, \\ &= U_0 + (U_1 k - \lambda d)/2^{64}. \end{aligned} \tag{1.40}$$

As $k < d$ and $\lambda, U_0, U_1 < 2^{64}$, we have that $0 \leq R' < 2^{64} + 2d < 4d$. Thus, if Q and R are the actual quotient and remainder satisfying $u = Qd + R$, and $0 \leq R < d$, then $R' - R < 4d$ and $Q - 3 \leq Q' < Q$. Thus, to correct Q' and R', we increment Q' by one and subtract R' by d until we have $R' < d$. Clearly we need at most 3 such corrections by this method (Algorithm 1.3, page 16). This method can be extended to divisors $d < 2^{63}$. Granlund and Moller [26] make further optimizations omitted here, which require fewer double-word integer operations and reduce the number of necessary corrections to at most 2.

1.5 The Chinese remainder algorithm

In many of our algorithms for computing $\Phi_n(z)$, we do not compute $\Phi_n(z)$ itself but rather images of $\Phi_n(z)$ modulo primes p_1, p_2, and so forth. The Chinese remainder algo-

rithm allows us to retrieve $\Phi_n(z)$, provided we have sufficiently many images.

Theorem 1.21 (The Chinese remainder theorem)**.** *Let $q_1, \ldots q_M \in \mathbb{Z}$ be pairwise coprime integers, and let $u_i \in \mathbb{Z}_{q_i}$ for $1 \leq i \leq M$. Then there exists a unique integer $U \in \mathbb{Z}$ such that $-\frac{Q}{2} \leq U < \frac{Q}{2}$, where $Q = q_1 \cdot q_2 \ldots q_M$, and*

$$U \equiv u_1 \pmod{q}_i \quad \textit{for } 1 \leq i \leq M. \tag{1.41}$$

We omit the proof of the Chinese remainder theorem. A proof can be found in [16], for instance.

Algorithm 1.4: The Chinese remainder algorithm

Input: $q_1, \ldots, q_M$, a set of coprime positive integers; $u_1, \ldots, u_M$, where u_i is an integer modulo q_i

Output: u in the range $-\frac{Q}{2} \leq u < \frac{Q}{2}$ satisfying $u \equiv u_i \pmod{q}_i$ for $1 \leq i \leq M$

$u \leftarrow u_1 \bmod q_1$

$Q_1 \leftarrow q_i$

for $k \leftarrow 2$ **to** M **do**

$Q_k \leftarrow Q_{k-1} \cdot q_k$

$U_k \leftarrow U_{k-1} + Q_{k-1}(u_k - u/Q_{k-1} \bmod q_k) \bmod Q_k$

return U_M

Given u_i and q_i for $1 \leq i \leq M$, the Chinese remainder algorithm allows us to find U satisfying the congruences (1.41). For $1 \leq k \leq n$, let $U_k \in \mathbb{Z}$ be the integer satisfying $U_k \equiv u_i \bmod q_i$ for $1 \leq i \leq k$ and $-\frac{Q_k}{2} \leq U_k < \frac{Q_k}{2}$ where $Q_k = q_1 \cdots q_k$. U_M and Q_M are exactly U and Q of Theorem 1.21, respectively. One can obtain Q_{k+1} from Q_k,

$$U_{k+1} = U_k + Q_k \cdot (Q_k^{-1} \cdot u_{k+1} - U_k \bmod u_i) \bmod Q_{k+1} : \tag{1.42}$$

where "$\bmod K$" here means in the symmetric range $[-\frac{K}{2}, \frac{K}{2})$.

Chapter 2

Computing $\Phi_n(z)$ using the fast Fourier transform

2.1 The discrete fast Fourier transform

Algorithm 1.1 describes a method of computing $\Phi_n(z)$ by way of a series of polynomial divisions. As such, fast polynomial arithmetic should prove useful in computing $\Phi_n(z)$. To that end, we detail the discrete fast Fourier transform.

Definition 2.1. *Let $\omega \in F$ be a primitive Nth root of unity and let $f(z) = \sum a_k z^k$ be a polynomial over F. The points $1, \omega, \omega^2, \ldots, \omega^{N-1}$ are called Fourier points. The N-tuple*

$$(f(1), f(\omega), f(\omega^2), \ldots, f(\omega^{N-1})), \tag{2.1}$$

is called the N-point discrete Fourier transform (or DFT) of f. We denote this transform by $\mathrm{DFT}(N, \omega, f(z)) \in F^N$. We alternately express the discrete Fourier transform as a polynomial:

$$f_\omega(z) = \mathrm{DFT}(N, \omega, f) = \sum_{k=0}^{N-1} f(\omega^k) z^k. \tag{2.2}$$

We call the polynomial $f_\omega \in F[z]$ the Fourier polynomial of f.

We alternate between expressing the DFT as an N-tuple or a polynomial where convenient throughout this chapter.

Using Horner's method, the evaluation of a polynomial f of degree less than N at a single point takes $\mathcal{O}(N)$ arithmetic operations in F; computing the DFT by this method would require $\mathcal{O}(N^2)$ operations. We can, however, compute the DFT for particular values of N in $\mathcal{O}(N \log N)$ operations by way of the fast Fourier transform, which we will describe here.

Suppose that $N = 2^s$ for some $s > 0$, and let f and ω be as in Definition 2.1 where f, moreover, has that $\deg(f) < N$. In which case, there exist polynomials $g(z), h(z) \in F[z]$ of degree less than 2^{s-1} for which

$$f(z) = g(z^2) + zh(z^2). \tag{2.3}$$

In particular, for $f(z) = a_0 + a_1 z + \cdots + a_{n-1}z^{n-1}$, we have that

$$g(z) = \sum_{i=0}^{2^{s-1}-1} a_{2i} z^i \text{ and } h(z) = \sum_{i=0}^{2^{s-1}-1} a_{2i+1} z^i, \tag{2.4}$$

satisfy (2.3). As N is even, ω^2 is a $(N/2)$th primitive root of unity by Corollary 1.17. Suppose then that we have the $(N/2)$-point discrete Fourier transforms of $g(z)$ and $h(z)$,

$$\begin{aligned} \mathrm{DFT}(N/2, \omega^2, g) &= (g(\omega^0), g(\omega^2), \ldots, g(\omega^{2\cdot(N/2-1)})) \text{ and} \\ \mathrm{DFT}(N/2, \omega^2, h) &= (h(\omega^0), h(\omega^2), \ldots, h(\omega^{2\cdot(N/2-1)})). \end{aligned} \tag{2.5}$$

We can obtain the N-point fourier transform of $f(z)$ (2.1) from the $(N/2)$-point transforms of g and h. For $j < N/2$,

$$f(\omega^j) = g(\omega^{2j}) + \omega^j \cdot h(\omega^{2j}). \tag{2.6}$$

To obtain the latter half of the DFT of f, we observe that $\omega^{N/2} = -1$ and hence $\omega^{N/2+j} = -\omega^j$. By (2.3), it follows that

$$f(-z) = g((-z)^2) - z \cdot h((-z)^2) = g(z^2) - z \cdot h(z^2), \tag{2.7}$$

in which case

$$f(\omega^{N/2+j}) = f(-\omega^j) = g(\omega^{2j}) - \omega^j \cdot h(\omega^{2j}). \tag{2.8}$$

If $N = 2^s$ is such that $s > 1$, then clearly $N/2$ is even, and we can express $g(z)$ and $h(z)$ as a sum analogous to f in (2.3). In which case we can obtain the $(N/2)$-point discrete

Procedure FFT(N,ω,f) : The fast Fourier transform

Input: F, a field; $N = 2^s$, a power of 2; ω, an Nth primitive root of unity in F;
$f(z) = \sum_{k=0}^{N-1} a_k z^k \in F[z]$, a polynomial of degree less than N

Output: $\mathrm{DFT}(N, \omega, f) \in F^N$, the discrete Fourier transform of $f(z)$

`// evaluate constant polynomial` $f(z) = a_0$

1 **if** $N = 1$ **then** $A_0 \leftarrow a_0$

2 **else**

3 $g(z) \longleftarrow \sum_{k=0}^{N/2-1} a_{2k} z^k$, $h(z) \longleftarrow \sum_{k=0}^{N/2-1} z_{2k+1} z^k$

4 $(B_0, B_1, \ldots, B_{N/2-1}) \longleftarrow$ FFT(N/2,ω^2,g),

5 $(C_0, C_1, \ldots, C_{N/2-1}) \longleftarrow$ FFT(N/2,ω^2,h)

6 $\lambda \longleftarrow 1$

7 **for** $j \longleftarrow 0$ *to* $N/2 - 1$ **do**

8 $t \longleftarrow \lambda C_j$

9 $A_j \longleftarrow B_j + t$, `// applying` (2.6)

10 $A_{N/2+j} \longleftarrow B_i - t$ `// applying` (2.8)

11 $\lambda \longleftarrow \lambda\omega$ `//` λ `is set to` ω^{j+1}

return $(A_0, A_1, \ldots A_{N-1})$

Fourier transforms of $g(z)$ and $h(z)$ from $(N/4)$-point transforms in a similar manner. For the base case $N = 1$, the discrete Fourier transform becomes trivial, as in such case we evaluate the constant polynomial $f = a_0$ to get $f(1) = a_0$. This recursive method of computing the DFT of $f(z)$ is the FFT algorithm, which we describe in procedure FFT. We measure the cost of the FFT in terms of the number of multiplications in F. It is easy to count the exact number of multiplications the FFT requires.

Let $T(N)$ be the number of multiplications in our field F to compute the N-point DFT by way of the FFT, as we describe in procedure FFT. From line 1 of procedure FFT, we have that $T(1) = 0$. For $N > 1$, we perform N multiplications in the for-loop beginning beginning on line 7, and one multiplication to compute ω^2, which we use in lines 4−5.

Thus $T(N) = 2T(N/2) + N + 1$ For $N = 2^s$, we have that

$$\begin{aligned} T(2^s) &= 2T(2^{s-1}) + N + 1, \\ &= 2\Big(2T(2^{s-2}) + N\Big) + N + 1 \qquad \text{provided } s > 1, \\ &= 4T(2^{s-2}) + 2N + 2. \end{aligned} \tag{2.9}$$

Continuing to recurse in this fashion until we have $T(N)$ entirely in terms of N and $T(1)$, we find that

$$T(2^s) = 2^s T(1) + sN + s = s(N+1). \tag{2.10}$$

That is, procedure FFT requires exactly $s(N+1) = (N+1)\log N$ multiplications to compute a N-point DFT. As integer multiplication is more expensive than addition or subtraction on a computer, we are less concerned with the total number of other arithmetic operations; however one can similarly show that the total number of arithmetic operations of procedure FFT is $\mathcal{O}(N \log N)$ as well.

We can use the FFT to take the N-point DFT of a polynomial f whose degree exceeds N, with some preconditioning. Every power of ω, an Nth primitive root of unity, is a root of $z^N - 1$. By Theorem 1.10, for any field F and any polynomial $f(z) \in F[z]$, there exists $Q, R \in F[z]$ satisfying

$$f(z) = Q(z)(z^N - 1) + R(z), \tag{2.11}$$

and either $\deg(R) < \deg(z^N - 1) = N$ or $R = 0$. In which case $f(\omega^k) = R(\omega^k)$ for any integer k. Thus to use the FFT to get the N-point DFT of f, we must first compute $R(z) = f \bmod (z^n - 1)$, and then run the FFT on input $R(z)$. Computing $R(z)$ is easy. For $d \geq N$,

$$z^d = z^{d-N}(z^N - 1) + z^{d-N}. \tag{2.12}$$

It follows that $z^d \equiv z^{d \bmod N} \pmod{z^N - 1}$. Thus for $f(z) = \sum_{i=0}^{D} a_i z^i$, we have that

$$f \bmod (z^N - 1) = \sum_{i=0}^{N-1} \left(\sum_{\substack{j \equiv i \pmod N \\ 0 \leq j \leq D}} a_i \right) z^i. \tag{2.13}$$

2.1.1 The inverse fast Fourier transform

Let $n = mp$ where $p \nmid m$ is prime and let N be a power of two as before. Suppose we have $\Phi_m(z)$ and we want to obtain the DFT of $\Phi_n(z)$. By Lemma 1.19, we have that

$$\Phi_n(\omega^k) = \frac{\Phi_m(\omega^{kp})}{\Phi_m(\omega^k)}. \tag{2.14}$$

We can compute the discrete Fourier transforms of $\Phi_n(z)$ and $\Phi_n(z^p)$ in $\mathcal{O}(N \log N)$ arithmetic operations via procedure FFT. We can subsequently compute the Fourier transform of $\Phi_n(z)$ by performing the division (2.14) for $0 \leq k < N$. One can then interpolate the N evaluation points $\mathrm{DFT}(N, \omega, \Phi_n(z))$ to obtain $\Phi_n(z)$, provided $N > \deg(\Phi(n)) = \phi(n)$.

Using a classical interpolation algorithm such as Newton or Lagrange interpolation takes $\mathcal{O}(N^2)$ arithmetic operations, defeating the purpose of FFT-based polynomial division. We require a sub-quadratic interpolation algorithm.

One can take the Fourier transform of a Fourier transform. This is how we obtain a polynomial from its Fourier transform. Recall that, given $f \in \mathbb{F}[z]$, f_ω is the polynomial defined as $f_\omega(z) = \sum_{i=0}^{N-1} f(\omega^i) z^i$. Then, for $f(z) = a_0 + a_1 z + \cdots + a_{N-1} z^{N-1}$,

$$f_\omega(z) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a_j \omega^{(ij \bmod N)} z^i \tag{2.15}$$

Evaluating $f_\omega(z)$ at $(\omega^{-1})^k$, where $0 \leq k < N$, we have

$$\begin{aligned} f_\omega(\omega^{-k}) &= \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a_j \omega^{i(j-k)} \\ &= \sum_{j=0}^{N-1} a_j \sum_{i=0}^{N-1} \omega^{i(j-k)}. \end{aligned} \tag{2.16}$$

If $k = j$, then the sum $\sum_{i=0}^{N-1} \omega^{i(j-k)}$ simplifies to N. Suppose then that $k \neq j$ and let $d = \gcd(j-k, N)$ and $m = N/d$. As $0 < j, k < N$, we have that $d < N$ hence $m > 1$. By Lemma 1.16, $\omega^{(j-k)}$ is an mth primitive root of unity, hence $\omega^{i(j-k)} = \omega^{(i \bmod m)(j-k)}$, and so

$$\sum_{i=0}^{N-1} \omega^{i(j-k)} = \sum_{i=0}^{d-1} \sum_{l=0}^{m-1} \omega^{l(j-k)} = \sum_{i=0}^{d-1} \omega^{(j-k)}. \tag{2.17}$$

Moreover, given $m > 1$, ω^{i-j} is a root of $(z^m - 1)/(z - 1) = z^{m-1} + z^{m-2} + \cdots + 1$. That is, $\sum_{l=0}^{m-1} \omega^{l(j-k)} = 0$, and the sum (2.17) simplifies to zero.

In summation, we have shown that $f_\omega(\omega^{-k}) = N \cdot a_k$, proving the following theorem.

Theorem 2.2 (Geddes, [16]). *Let f, N, ω be as in Definition 2.1. Suppose* $\deg(f) < N$. *Then* $(f_\omega)_{\omega^{-1}}(z)/N = f(z)$.

The transform of a polynomial $f_\omega(z) \in F[z]$,

$$\mathrm{IDFT}(N, \omega, f) = (\tfrac{1}{N} f_\omega(\omega^0), \tfrac{1}{N} f_\omega(\omega^{-1}), \ldots, \tfrac{1}{N} f_\omega(\omega^{-(N-1)})), \tag{2.18}$$

is accordingly called the inverse discrete Fourier transform (IDFT) of $f_\omega(z)$.

As $DFT(N, \omega, f) = DFT(N, \omega, f \bmod (z^n - 1))$, we can generalize Theorem 2.2.

Corollary 2.3. *Let f, N, ω be as in Definition 2.1. Then* $(f_\omega)_{\omega^{-1}}(z)/N = f(z) \bmod (z^N - 1)$.

2.2 Implementing the fast Fourier transform

The fast Fourier transform, as is presented in procedure FFT, does not wholly reflect our implementation of the algorithm. We could perform Algorithm 2.1 in the field $\mathbb{C}$ or $\mathbb{Q}(\omega)$; however, complex arithmetic is cumbersome to implement and does not lend itself to fast computation. We implement the FFT for polynomials over the field $\mathbb{Z}_q$, the integers modulo a prime q. As it is necessary that our field contains an Nth primitive root of unity for N a power of 2, we choose primes q of the form $q = 2^s r + 1$. We call such primes *Fourier primes*. Table 2.1 lists the primes we used in our computation and the Nth primitive roots of unity we used. Corollary 2.3 tells us that if we want to obtain a polynomial from its DFT, that we require that the size of its DFT exceed its degree. For primes $q < 2^{32}$ of the form, $q = r \cdot 2^s + 1$, s is at most 28, hence limiting us to cyclotomic polynomials of degree at most $2^{28} - 1$. Polynomials of larger degree require that we use 64-bit integer arithmetic.

We use a dense representation to store a polynomial $f \in \mathbb{Z}_q[z]$. That is, we store a polynomial $f(z) \in F[z]$ as an array of machine-precision integers

$$A = [A_0, A_1, \ldots, A_{N-1}], \tag{2.19}$$

Table 2.1: Primes and the primitive roots used in our FFT calculations

q	$=$	$r \cdot N + 1$	size of q*	Nth prim. root of unity in $\mathbb{Z}_q$
469762049	$=$	$7 \cdot 2^{26} + 1$	29 bits	2187
1811939329	$=$	$27 \cdot 2^{26} + 1$	31 bits	72705542
2013265921	$=$	$15 \cdot 2^{27} + 1$	31 bits	440564289
2748779069441	$=$	$5 \cdot 2^{39} + 1$	42 bits	243
4123168604161	$=$	$15 \cdot 2^{38} + 1$	42 bits	624392905782
2061584302081	$=$	$15 \cdot 2^{37} + 1$	41 bits	624392905781
206158430209	$=$	$3 \cdot 2^{36} + 1$	38 bits	10648
2027224563713	$=$	$59 \cdot 2^{35} + 1$	41 bits	1609812002788
1945555039024054273	$=$	$27 \cdot 2^{56} + 1$	61 bits	1613915479851665306
4179340454199820289	$=$	$29 \cdot 2^{57} + 1$	62 bits	68630377364883
6269010681299730433	$=$	$87 \cdot 2^{56} + 1$	63 bits	4467632415761384939

*We let the size of q be the number of bits necessary to store q as an unsigned integer, i.e. $\lfloor \log_2(q) \rfloor + 1$.

where, given a polynomial $f(z) = \sum_{k=0}^{N-1} a_k z^k$, A_k is initially set to the value a_k for $0 \leq k < N$. Our implementation of the FFT writes the N-point discrete Fourier transform of $f(z)$ to the array A, discarding $f(z)$ in the process; upon completion of the FFT, A_k is set to $f(\omega^k)$. To introduce some basic array notation, given an array A and integer m we let $A + m$ refer to the array starting at A_m. That is,

$$A + m = [A_m, A_{m+1}, \ldots]. \tag{2.20}$$

and $(A + m)_l = A_{m+l}$. Procedure FFT2 shows how our implementation of the fast Fourier transform organizes the problem using an array. We require additional space for $N/2$ integers to rearrange the values stored in A. We effectively let A be an array of size at least $3N/2$ and use $W = A + N$ as working space. After rearranging our values such that $g(z)$ and $h(z)$ of (2.4) are written to A and $A + 3N/4$ respectively, we recursively execute the FFT on g and h. This approach requires the storage of $3N/2$ elements of F, as opposed to $2N$ elements of F in procedure FFT. We have in the case that $N = 2$ that we do not have to rearrange any values in our array, as in this case f is of the form $f(z) = a_0 + a_1 z$, thus $g(z)$

and $h(z)$ of (2.3) are exactly the constant polynomials a_0 and a_1 respectively. Moreover, it is unnecessary to recurse in such case.

Procedure FFT2(N, M, q, α, A, Ω) : an implementation of the FFT

Input:

- $N = 2^s$ and $M = 2^t$, powers of 2 for which $M \geq N$.
- q a prime of the form $q = Mr + 1$.
- α, set to either true or false.
- $A = [A_0, A_1, \dots]$, an array of at least $3N/2$ integers modulo q. Given $f = \sum_{i=0}^{N-1} a_i z^i \in \mathbb{Z}_q[z]$, a polynomial of degree less than N, A_i is initialized to a_i.
- $\Omega = [\omega^0, \dots, \omega^{M/2}]$, a an array containing precomputed powers of ω, an Mth primitive root of unity in $\mathbb{Z}_q$.

Result: If α is true, $\mathrm{DFT}(N, \omega^{M/N}, f)$, is written to $[A_0, \dots, A_{N-1}]$. That is, A_k is set to $f((\omega^{kM/N})$ upon completion, for $0 \leq k < N$. If α is false, $\mathrm{DFT}(N, \omega^{-1}, f)$ is written to $[A_0, \dots, A_{N-1}]$.

1 **if** $N = 1$ **then return else**
2 **if** $N \geq 4$ **then**
3 $W \leftarrow A + N$ `// W is the array with 1st entry is` A_N
  `// the terms of` $g(z)$ `and` $h(z)$`, where` $f(z) = g(z^2) + zh(z^2)$`,`
  `are stored in` A `and` $A + N/2$`, respectively`
4 **for** $k \leftarrow 0$ **to** $N/2 - 1$ **do** $W_k \leftarrow A_{2k+1}, \quad A_k \leftarrow A_{2k}$
5 **for** $k \leftarrow 0$ **to** $N/2 - 1$ **do** $A_{3N/4+k} \leftarrow W_k$
6 FFT2($N/2$, M, q, α, A, Ω)
7 FFT2($N/2$, M, q, α, $A + 3N/4$, Ω)
8 **if** α *is true* **then** $(k, d) \leftarrow (0, M/N)$ **else** $(k, d) \leftarrow (M/2, -M/N)$
9 **for** $j \leftarrow 0$ **to** $N/2 - 1$ **do**
10 $t \leftarrow A_{3N/4+j} \cdot \Omega_k \bmod q$ `//` $\Omega_k = \omega^k$ `is a precomputed value`
11 $k \leftarrow k + d$
12 $c \leftarrow A_j$
13 $A_j \leftarrow c + t \bmod q, \quad A_{N/2+j} \leftarrow c - t \bmod q$
  return

We make an additional improvement to the performance of the FFT at the cost of memory. Instead of computing powers of ω every time we recurse, we can precompute powers of ω instead. This is particularly useful if we intend to perform many FFTs over the same field $\mathbb{Z}_q$. We fix a power of 2 $M = 2^t$, sufficiently large such that any DFT we may want to compute is of size $N \leq M$. Then, given Mth primitive root of unity ω, we precompute powers of ω and store them in an array $\Omega = [\Omega_0, \Omega_1, \dots]$ where ω^k is written to Ω_k. We pass M and Ω as additional parameters into procedure FFT2. The FFT only requires the first half of powers of ω. If we want a DFT of size $N = 2^s < M$, then certainly $N|M$ and by Lemma 1.16 $\zeta = \omega^{M/N}$ is an Nth primitive root of unity. The powers of ζ are exactly the values ω^k for which M/N divides k. We can, moreover, use these same precomputed powers for the inverse FFT as well. If $k < M/2$, then, as $\omega^{M/2} \equiv -1 \pmod{q}$,

$$(\omega^{-1})^k = \omega^{M-k} = -\omega^{M/2-k}. \tag{2.21}$$

We store the first $M/2 + 1$ powers of ω in Ω. By precomputing the powers of omega we reduce the number of multiplications. If $T(N)$ is the number of multiplications in our new procedure FFT2, we have that $T(N) = 2T(N/2) + N/2$ and $T(1) = 0$. Solving for this recurrence relation we have $T(N) = N\log(N)/2$, or half the multiplications necessary by procedure FFT.

2.3 Fast FFT-based polynomial division

We now have an outline of a fast algorithm to perform polynomial division. To compute the quotient $f = g/h$, where $f, g, h \in F[z]$, we can use the FFT to compute the DFTs of g and h, then from those iteratively produce the DFT of f through a sequence of divisions. This is how we go about computing $\Phi_{mp}(z) = \Phi_m(z^p)/\Phi_m(z)$ in algorithm 2.1. The author and Monagan first describe this approach in [2].

Algorithm 2.1 requires storage for approximately $5N/2 + M/2$ integers, where N is the least power of two greater than $\deg(\Phi_n(z)) = \phi(n)$ and M is a power of two at least N: N integers to store each of $\Phi_m(z)$ and $\Phi_m(z^p)$, stored in arrays A and B respectively; $M/2 + 1$ integers for our array Ω containing precomputed powers of ω; and the additional array of size $N/2$ used as work space in the FFT2 procedure. Ideally, we would like to choose M such that for the cyclotomic polynomials of highest degree that we would like to

Algorithm 2.1: Calculating $\Phi_n(z)$ by repeated FFT-based division

Input:

- $n = p_1 p_2 \dots p_k$, a product of k distinct primes.
- N, the least power of two greater than $\phi(n)$.
- M, a power of two at least N.
- q, a prime of the form $q = rM + 1$.
- $A = [A_0, A_1, \dots, A_{5N/2-1}]$, an array of $5N/2$ integers modulo q, whose values are initialized to zero.
- $\Omega = [\omega^0, \dots, \omega^{M/2}]$, a an array containing precomputed powers of ω, an Mth primitive root of unity in $\mathbb{Z}_q$.

Result: The coefficients of $\Phi_n(z)$ are written to $[A_0, \dots, A_{N-1}]$

1 $B \longleftarrow A + N$
2 **for** $i \leftarrow 0$ **to** p_1 **do** $A_i \leftarrow 1$ // A now stores $\Phi_{p_1}(z) = \sum_{i=0}^{p-1} z^i$
3 $m \leftarrow p_1, \quad N^* \leftarrow 1$
4 **for** $i \longleftarrow 2$ **to** k **do**
5 **while** $N^* \leq \phi(mp_i)$ **do** $N^* \longleftarrow 2N^*$
6 **for** $j \leftarrow 0$ **to** $N^* - 1$ **do** $B_i \longleftarrow 0$
// Write $\Phi_m(z^{p_i}) \bmod z^{N^*} - 1$ to B
7 **for** $j \leftarrow 0$ **to** $\phi(m)$ **do** $B_{jp_i \bmod N^*} \leftarrow A_j$
8 FFT2(N^*, M, q, true, A, Ω) // Write DFT of $\Phi_m(z)$ to A
9 FFT2(N^*, M, q, true, B, Ω) // Write DFT of $\Phi_m(z^{p_i})$ to B
10 **for** $j \leftarrow 0$ **to** $N^* - 1$ **do** $A_j \longleftarrow B_j/A_j$ // Write DFT of $\Phi_{mp_i}(z)$ to A
11 FFT2(N^*, M, q, false, A, Ω) // Interpolate $\Phi_{mp_i}(z)$
12 $m \longleftarrow mp_i$
return

compute, $N = M$ and, for such cases, the memory cost is $3N$. We can always reduce the memory cost, if necessary, by not precomputing powers of ω.

We require that $A_j = \Phi_m(\omega^k) \bmod q$ is nonzero for $0 \leq k < N$ in order to perform the division in line 10 of Algorithm 2.1. For $\Phi_n(z)$ where n is odd and squarefree, however, this certainly holds. To prove this, we use the following basic result.

Lemma 2.4.

$$\Phi_n(1) = \begin{cases} 0: & \textit{if } n = 1 \\ p: & \textit{if } n = p^\alpha, \textit{ where } \alpha > 0, \textit{ for some prime } p \\ 1: & \textit{otherwise} \end{cases} \tag{2.22}$$

Proof. Clearly $\Phi_1(1) = 0$. Suppose then that $n = p^\alpha$ is a prime power. By Lemma 1.18, $\Phi_n(1) = \Phi_{n/p}(1^p) = \Phi_{n/p}(1)$. By repeated application of Lemma 1.18, $\Phi_n(1) = \Phi_p(1)$. As $\Phi_p(z) = 1 + z + \cdots + z^{p-1}$, we have $\Phi_n(1) = \Phi_p(1) = p$ as desired.

Now suppose, towards a contradiction, that n is the least integer for which (2.22) does not hold. n must have at least two distinct prime factors, thus $\Phi_n(1) \neq 1$. In which case, we can write $n = mp^\alpha$ where $\alpha \geq 1$, $m > 1$ and p is a prime not dividing m. By Lemma 1.19 $\Phi_{mp}(z) = \Phi_m(z^p)/\Phi_m(z)$. As $m < n$, by our choice of n, m must satisfy (2.22). Thus, given $m > 1$, we have that $\Phi_m(1)$ is nonzero, and hence $\Phi_{mp}(1) = \Phi_m(1^p)/\Phi_m(1) = \Phi_m(1)/\Phi_m(1) = 1$. If $\alpha = 1$, then $\Phi_n(z) = 1$, a contradiction. Otherwise, using Lemma 1.18 again, we have that

$$\Phi_{mp^\alpha}(z) = \Phi_{mp^{\alpha-1}}(z^p) = \cdots = \Phi_{mp}(z^{p^{\alpha-1}}), \tag{2.23}$$

thus $\Phi_n(1) = \Phi_{mp^\alpha}(1) = \Phi_{mp}(1^{p^\alpha - 1}) = \Phi_{mp}(1) = 1$, again contradicting our choice of n. □

Lemma 2.5. *Let $n > 1$ and $N > 1$ be coprime. Moreover, let q be a prime of the form $q = rN + 1$, $q \nmid n$, and suppose that ω is an Nth primitive root of unity in $\mathbb{Z}_q$. Then $\Phi_n(\omega^k) \bmod q \neq 0$.*

Proof. Towards a contradiction, let $\lambda = \omega^k$ and suppose $\Phi_n(\lambda) \bmod q = 0$. Certainly, $\lambda^N = (\omega^N)^k \equiv 1 \bmod q$. Moreover, given $\Phi_n(z)\Psi_n(z) = z^n - 1$, we have that

$$\begin{aligned} \lambda^n - 1 &\equiv \Phi_n(\lambda)\Psi_n(\lambda) \pmod q, \\ &\equiv 0 \pmod q, \end{aligned} \tag{2.24}$$

and hence $\lambda^n \equiv 1 \pmod q$. As n, N are coprime, there exist integers s and t such that $ns + Nt = 1$. Thus

$$\lambda = \lambda^{ns+Nt} = (\lambda^n)^s(\lambda^N)^t \equiv 1 \pmod q. \tag{2.25}$$

Thus $\lambda \bmod q = 1$. By Lemma 2.4, we have that $\Phi_n(1) \bmod q = 0$ if and only if n is a power of q. As $q \nmid n$, this completes the proof. □

Thus, in the case where $N = 2^s$ and n is odd, Lemma 2.5 implies that $\Phi_n(\omega^k) \bmod q \neq 0$, provided our Fourier prime q is not a divisor of n. We note that Lemma 2.5 does hold for $n = 1$, as $\Phi_1(\omega^0) = \Phi_1(1) = 0$.

2.3.1 Reconstructing $\Phi_n(z)$ from multiple images

Our implementation of Algorithm 2.1 computes the image $\Phi_n(z) \bmod q$, from which we need to retrieve $\Phi_n(z)$. The natural candidate for the inverse image of $\Phi_n(z) \bmod q$ is the unique polynomial $f(z) = \sum_{i=0}^{\phi(n)} a_i z^i \in \mathbb{Z}[z]$ for which the coefficients a_i lie in the range $[-q/2, q/2)$ and $(\Phi_n(z) - f(z)) \bmod q = 0$. If $\Phi_n(z)$ has a coefficient exceeding $q/2$, then $f(z) \neq \Phi_n(z)$, in which case we may have to compute multiple images of $\Phi_n(z)$. We can use the Chinese remainder algorithm (alg. 4, page 18) to reconstruct $\Phi_n(z)$ from multiple images.

Given $f_j = \Phi_n(z) \bmod q_j = \sum_{i=0}^{\phi(n)} a_{i,j} z^i$ for $1 \leq j \leq K$, we perform the Chinese remainder algorithm on the coefficients of terms of a fixed degree. That is, for $0 \leq i \leq \phi(n)$, we find the unique integer $-\frac{Q}{2} < a_i \leq \frac{Q}{2}$, where $Q = q_1 q_2 \cdots q_K$, satisfying

$$a_i \equiv a_{i,j} \pmod{q_j} \text{ for } 1 \leq j \leq k. \tag{2.26}$$

The polynomial $f(z) = \sum_{i=0}^{\phi(n)} a_i z^i$ whose coefficients lie in $[-\frac{Q}{2}, \frac{Q}{2})$ and, moreover, satisfying $\Phi_n(z) \equiv f(z) \pmod{Q}$. If $A(n)$, the height of $\Phi_n(z)$, is less than $Q/2$, then $f = \Phi_n(z)$. We do not necessarily know if we have computed sufficiently many images of $\Phi_n(z)$. There are bounds for $A(n)$, the height of $\Phi_n(z)$ (see Section 5.2.1); however, oftencase said bounds are far greater than the height $A(n)$ itself. It is sometimes very impractical to compute sufficiently many images such that $Q/2$ exceeds a bound on $A(n)$.

In some cases we like to guarantee $f(z) = \Phi_n(z)$ with extremely high probability. In such cases we typically compute multiple images of $\Phi_n(z)$ and use Chinese remaindering to compute the inverse image $f(z) \in \mathbb{Z}[z]$ after computing each new image of $\Phi_n(z)$. We stop when the height of $f(z)$ is less than $Q/2^{20}$. We then, in addition, use the FFT to test that f is correct modulo an additional check prime q^*. By Lemma 1.19, if $n = mp$ with p prime, $\Phi_n(z)\Phi_m(z) - \Phi_m(z^p) = 0$. We let N be the least power of two greater than $\phi(m)p$, and compute

$$\mathrm{DFT}(N, \omega, f(z)), \mathrm{DFT}(N, \omega, \Phi_m(z)), \text{ and } \mathrm{DFT}(N, \omega, \Phi_m(z^p)), \tag{2.27}$$

in $\mathbb{Z}_{q^*}$, and check that

$$f(\omega^k) \cdot \Phi_m(\omega^k) - \Phi_m((\omega^k)^p) \bmod q^* = 0 \text{ for } 0 \leq k < N. \tag{2.28}$$

This additional check is less expensive than computing $\Phi_n(z) \bmod q^*$, as it avoids having to perform an additional inverse FFT.

2.4 A second means of computing the DFT of $\Phi_n(z)$

The N-point DFT of $\Phi_m(z^p)$,

$$(\Phi_m(\omega^0), \Phi_m(\omega^p), \ldots, \Phi_m(\omega^{p(N-1)})), \tag{2.29}$$

is merely a permutation of the DFT of $\Phi_m(z)$. In Algorithm 2.1, we perform three FFTs at every division step. To compute $\Phi_{mp}(z) = \Phi_m(z^p)/\Phi_m(z)$ we use two FFTs to compute the DFTs of $\Phi_m(z^p)$ and $\Phi_m(z)$, then an additional inverse FFT to interpolate $\Phi_{mp}(z)$. We can forego these FFTs by directly computing the DFT of $\Phi_{mp}(z)$ directly from that of $\Phi_m(z)$ by performing the division

$$\Phi_{mp}(\omega^i) = \frac{\Phi_m(\omega^{(ip \bmod N)})}{\Phi_m(\omega^i)}, \tag{2.30}$$

where "$\bmod N$" in this context means in the nonnegative range $[0, N)$, for $0 \leq i < N$. As such, we can compute the DFT of $\Phi_n(z)$, where $n = mp$, directly from that of $\Phi_m(z)$ in $\mathcal{O}(N)$ divisions modulo q. We call this method the *cyclotomic Fourier transform* or CFT. A disadvantage of this approach (procedure CFT) is that we must use a DFT of fixed size for the whole computation whereas in FFT2 we use only as large a discrete Fourier transform as is necessary to extract the quotient at each division step.

The discrete Fourier transform of $\Phi_1(z)$,

$$DFT(N, \omega, \Phi_1) = (0, \omega - 1, \omega^2 - 1, \ldots, \omega^{N-1} - 1), \tag{2.31}$$

can be obtained in linear time by iteratively computing successive powers of omega. We use Lemma 2.4 to compute $\Phi_n(1)$. As $N = 2^s$ is a power of two, we easily obtain $(l + p) \bmod N$ in line 8 of procedure CFT by taking the lower s bits of $l + p$, as discussed in Section 1.4.

Procedure CFT(n,N,q,ω,A) : Computing the Fourier transform of $\Phi_n(z)$

Input: $n = p_1 p_2 \dots p_k$, a product of $k \geq 1$ distinct primes; N, an integer relatively prime to n; q, a prime such that $N|\phi(q)$; ω, an Nth primitive root of unity modulo q; $A = [A_0, \dots, A_{2N-1}]$ an array of $2N$ elements of $\mathbb{Z}_q$

Output: CFT returns an array containing $\mathrm{DFT}(N, \omega, \Phi_n)$

$\lambda \leftarrow 1$

$B \leftarrow A + N$

`// write DFT of` $z-1$ `to` A

for $i \leftarrow 0$ **to** $N-1$ **do** $A_i \leftarrow \lambda - 1 \bmod q, \quad \lambda \leftarrow \lambda\omega \bmod q$

for $j \leftarrow 1$ **to** k **do**

  $(A, B) \leftarrow (B, A)$

  $l \leftarrow p_j \bmod N$

  for $i \leftarrow 1$ **to** *N-1* **do**

    $A_i \leftarrow B_l / B_i \bmod q, \quad l \leftarrow l + p \bmod N$

`// Get` $\Phi_n(1)$ `by Lemma 2.4`

if $k = 1$ **then** $A_0 \leftarrow p \bmod N$ **else** $A_0 \leftarrow 1$

return A

For n, a product of k distinct primes, procedure CFT can compute $DFT(N, \omega, \Phi_n)$ in $\mathcal{O}(kN)$ arithmetic operations in $\mathbb{Z}_q$, as opposed to $\mathcal{O}(N \log N) = \mathcal{O}(N \log \phi(n))$ by way of the FFT. For squarefree odd n, $\log_2(\phi(n))$ is at least k. In fact, for arbitrarily large, squarefree n, $k/\log_2(\phi(n))$ becomes arbitrarily small. That is, if we define $\theta(n)$ by

$$\theta(n) = \begin{cases} k & \text{if } \ n = p_1 p_2 \cdots p_k, \text{ a squarefree product of } k \text{ distinct primes} \\ 0 & \text{otherwise,} \end{cases} \tag{2.32}$$

then the following holds.

Lemma 2.6.

$$\lim_{n \to \infty} \frac{\theta(n)}{\log_2(\phi(n))} = 0. \tag{2.33}$$

Proof. Suppose $n > 1$ is squarefree and let $L > 1$ be the integer satisfying $L! \leq \phi(n) < (L+1)!$. Write $n = p_1 p_2 \dots p_k$, where $p_1 < p_2 < \cdots < p_k$ and p_i is prime for $1 \leq i \leq k$.

Clearly $p_j - 1$ must be at least j. As such, $\phi(n) = \prod_{j=1}^{k} p_j - 1 \geq k!$, and $\theta(n) = k < L+1$. Thus $\theta(n)/\log_2(\phi(n)) < (L+1)/\log_2(L!)$. As $L! > \lfloor L/2 \rfloor^{\lfloor L/2 \rfloor}$, we have

$$\begin{aligned}(L+1)/\log_2(L!) &< (L+1)/\log_2(\lfloor L/2 \rfloor^{\lfloor L/2 \rfloor}) \\ &= \frac{(L+1)}{\lfloor L/2 \rfloor} \frac{1}{\log_2(\lfloor L/2 \rfloor)} \\ &< \frac{4}{\log_2(\lfloor L/2 \rfloor)}.\end{aligned} \tag{2.34}$$

Taking the limit of $4/\log_2(\lfloor L/2 \rfloor)$ as L goes to infinity, we have

$$\lim_{L \to \infty} \frac{4}{\log_2(\lfloor L/2 \rfloor)} = 0, \tag{2.35}$$

completing the proof. □

Computing $\Phi_n(z)$ via the CFT, however, still requires $\mathcal{O}(N \log N)$ arithmetic operations, as we use the inverse FFT to interpolate $\Phi_n(z)$ from its DFT. As our implementation of the FFT (alg. 2.1) uses precomputed powers of ω to expedite the (inverse) FFT, we can use those powers of omega instead of recomputing them as we do on line 3 of procedure CFT. In the for loop beginning on line 4, we read from one array and write to another array. Which array our final result is in depends on the parity of k, the number of prime divisors of n. The CFT procedure returns the array containing the discrete Fourier transform of $\Phi_n(z)$ at the end of the computation.

2.4.1 A division-free CFT

The CFT procedure is not a particularly fast method of computing the discrete Fourier transform of $\Phi_n(z)$. This is because the procedure requires around kN division operations in $\mathbb{Z}_q$. We perform division modulo q by first inverting the denominator by way of the extended Euclidean algorithm and then multiplying the numerator by the inverted denominator. This is considerably slower than multiplication in $\mathbb{Z}_q$.

Our first improvement of the CFT was to reduce the number of divisions at the cost of additional storage for $2N$ integers modulo q. We use four arrays A, B, C, D, each of size N, in this approach. Instead of storing $\Phi_m(\omega^i)$ as one integer modulo q for $0 \leq i < N$, we

store integers A_j, C_j such that the fraction $A_j/C_j = \Phi_m(\omega^i)$. Suppose then that we wanted a fractional representation of $\Phi_{mp}(\omega^i)$ written to arrays B and D. As

$$\Phi_{mp}(\omega^i) = \frac{\Phi_m(\omega^{ip \bmod N})}{\Phi_m(\omega^i)} = \frac{A_{ip \bmod N} C_i}{A_i C_{ip \bmod N}}, \tag{2.36}$$

we merely write $A_{ip \bmod N} C_i$ to B_i and $A_i C_{ip \bmod N}$ to D_i. In this approach we have to perform N divisions at the very end of our computation after we have a fractional representation of the DFT of $\Phi_n(z)$. We call this method CFT2.

Procedure CFT2(n,N,q,ω,A) : Computing the Fourier transform of $\Phi_n(z)$

Input: $n = p_1 p_2 \dots p_k$, a product of $k \geq 1$ distinct primes; N, an integer relatively prime to n; q, a prime such that $N | \phi(q)$; ω, an Nth primitive root of unity modulo q; $A = [A_0, \dots, A_{4N-1}]$, an array of $4N$ integers modulo q.

Output: CFT2 returns an array containing $\mathrm{DFT}(N, \omega, \Phi_n)$

```
B ← A + N,   C ← A + 2N,   D ← A + 3N
λ ← 1
// store Φ_1(ω^i) as A_i/C_i
for i ← 0 to N − 1 do A_i ← λ − 1 mod q,   λ ← λ · ω,   C_i ← 1
for j ← 1 to k do
    (A, B, C, D) ← (B, A, D, C)
    l ← p_j mod N
    for i ← 1 to N-l do
        A_i ← B_l D_i,   C_i ← B_i D_l,   l ← l + p mod N
if k = 1 then A_0 ← p mod N else A_0 ← 1
for i ← 1 to N − 1 do A_i ← A_i/C_i // Division step
return A
```

Instead of delaying division until the end of the computation, we can instead move the division step to the start of the computation.

For notational convenience, we extend our definition of the discrete Fourier transform to rational functions. Given $f(z) \in \mathbb{Z}_q(z)$, we let $DFT(N, \omega, f)$ refer the set of values $f(\omega^k)$ for which $f(\omega^k)$ is defined, for $0 \leq k < N$.

Given the DFT of $\Phi_1(z)$, we compute the DFT of $1/\Phi_1(z)$. Note that $1/\Phi_1(\omega^k) = 1/(\omega^k - 1)$ for $0 < k < N$ is defined whereas $1/\Phi_1(\omega^0)$ is not. We invert $\Phi_1(\omega^i)$, for $i \leq 1 \leq N$ by way of the extended Euclidean algorithm. Using (2.30), we can compute the DFT of Φ_{mp} and $1/\Phi_{mp}$, excluding $\Phi_{mp}(1)$ and $1/\Phi_{mp}(1)$, from that of Φ_m and $1/\Phi_m$ respectively. In the last step of the computation it is unnecessary to compute the DFT of $1/\Phi_n(z)$. We use Lemma 2.4 to obtain $\Phi_n(1)$. This approach is particularly advantageous if we want to compute many cyclotomic polynomials, in which case we can precompute and store the values

$$1/\Phi_1(\omega^1), \ldots, 1/\Phi_1(\omega^{N-1}), \tag{2.37}$$

and use these values repeatedly, thus avoiding divisions altogether. Typically we precompute $\mathrm{DFT}(M, \omega, 1/\Phi_1)$ for ω, an Mth primitive root of unity modulo q where $M = 2^t$, a fixed power of two exceeding the degree of the cyclotomic polynomials we would like compute. If we want to compute $\Phi_n(z)$, where $2^{s-1} \leq \phi(n) < 2^s = N < M = 2^t$, we can easily obtain the 2^s-point DFT of $1/\Phi_1$, $\mathrm{DFT}(2^s, \omega^{2^{t-s}}, 1/\Phi_1)$ from $\mathrm{DFT}(M, \omega, 1/\Phi_1)$. In our timed implementation (see Section 5.1 for timings), we used $M = 2^{25}$. $\mathrm{DFT}(M, \omega, 1/\Phi_1)$ was written to disk. We read $\mathrm{DFT}(M, \omega, 1/\Phi_1)$ from disk every time we ran CFT3. Our choice of M affects performance. For $\Phi_n(z)$ with degree appreciably smaller than M, we find that the cost of $\mathrm{DFT}(M, \omega, 1/\Phi_1)$ from disk outweighs the cost of performing N divisions modulo q.

As always, there is a time-space tradeoff in effect. CFT2 requires storage for $4N$ integers whereas CFT3 requires $4N + M$.

2.4.2 A reduced-memory division-free CFT

In CFT1-3, we read from one array and write our new result to another array. We can, however, instead read and write to the same part of the same array. Procedure CFT4 shows how we may modify CFT3 to do this. Let $p \nmid N$ be prime. As p is invertible modulo N, it generates the multiplicative group

$$G = \{p^\alpha \bmod N : \alpha \in \mathbb{Z}\}. \tag{2.38}$$

G is a subgroup the multiplicative group $\mathbb{Z}_N^*$; moreover, this group naturally acts on $\mathbb{Z}_N$ by way of multiplication modulo N. One natural ordering to perform the division (2.30) for

Procedure CFT3(n,N,q,ω,A) : Computing the Fourier transform of $\Phi_n(z)$

Input: $n = p_1 p_2 \dots p_k$, a product of $k \geq 1$ distinct primes; N, an integer relatively prime to n; q, a prime such that $N|\phi(q)$; ω, an Nth primitive root of unity modulo q; $A = [A_0, \dots, A_{4N-1}]$, an array of $4N$ integers modulo q.

Output: CFT3 returns an array containing $\mathrm{DFT}(N, \omega, \Phi_n)$

$B \leftarrow A + N, \quad C \leftarrow A + 2N, \quad D \leftarrow A + 3N$

// $1/\Phi_1(\omega^i)$ obtained via precomputation

$(C_1, C_2, \dots, C_{N-1}) \leftarrow (1/\Phi_1(\omega), 1/\Phi_1(\omega^2), \dots, 1/\Phi_1(\omega^{N-1}))$

$\lambda \leftarrow 1$

for $i \leftarrow 0$ **to** $N-1$ **do** $A_i \leftarrow \lambda - 1$, $\lambda \leftarrow \lambda \cdot \omega$ // write DFT of Φ_1 to A

for $j \leftarrow 1$ **to** $k-1$ **do**

  $(A, B, C, D) \leftarrow (B, A, D, C), \quad l \leftarrow p_j \bmod N$

  for $i \leftarrow 1$ **to** *N-1* **do**

    $A_i \leftarrow B_l D_i \mod q, \quad C_i \leftarrow B_i D_l \mod q, \quad l \leftarrow l + p \bmod N$

$p \leftarrow p_k, \quad (A, B, C, D) \leftarrow (B, A, D, C)$

for $i \leftarrow 1$ **to** $N-1$ **do** $A_i \leftarrow B_l D_i, \quad l \leftarrow l + p \bmod N$

if $k = 1$ **then** $A_0 \leftarrow p_1 \bmod N$ **else** $A_0 \leftarrow 1$ // Lemma 2.4

return A

$0 \leq i < N$ is to group $i \in \mathbb{Z}_N$ belonging to one orbit under the action of G. We denote by $G \cdot i$ the orbit of i.

Suppose that we have the N-point DFTs of $\Phi_m(z)$ and $1/\Phi_m(z)$ in arrays A and C, and we aim to compute $\Phi_{mp}(\omega^k)$ and $1/\Phi_{mp}(\omega^k)$, for all $k \in G \cdot i$, and write the data back into the arrays A and C. We first save the values $\Phi_m(\omega^i)$ and $1/\Phi_m(\omega^i)$, stored in A_i and C_i respectively. Then we set A_i to $A_{ip \bmod N} C_i = \Phi_m(\omega^{ip})/\Phi_m(\omega^i)$, discarding the previous value of A_i. We then set C_i to $\Phi_m(\omega^i) C_{ip \bmod N}$. We proceed to update arrays A and C at indices ip, ip^2, ip^3, and so forth, until we reach ip^l for which $ip^{l+1} \bmod p = i$, at which point we need to use the previously saved values $\Phi_m(\omega^i)$ and $1/\Phi_m(\omega^i)$.

Over the computation of $\mathrm{DFT}(N, \omega, \Phi_{mp})$, our array A may simultaneously store values from the previous and the new DFT. In order to distinguish between the two, we could use an additional array of bits B where B_i is set to 1 if A_i stores $\Phi_{mp}(\omega^i)$ and 0 otherwise;

however, it is less cumbersome to store this information in the array A itself. We store our integers modulo q in the positive range $[0, q-1)$. Upon updating A_i to some new value $x \bmod q$, we set A_i to $-x$. Thus we know not to update values A_i for which A_i is negative (line 8 of CFT4)). The additional memory cost of this approach (CFT4) is one bit for every array value A_i, stored as a two's complement machine-precision integer.

Procedure CFT4(n,N,q,ω,A) : Computing the Fourier transform of $\Phi_n(z)$

Input: $n = p_1 p_2 \dots p_k$, a product of $k \geq 1$ distinct primes; N, an integer relatively prime to n; q, a prime such that $N|\phi(q)$; ω, an Nth primitive root of unity modulo q; $A = [A_0, \dots, A_{2N}]$ an array of $2N$ integers

Output: CFT4 returns an array containing $\mathrm{DFT}(N, \omega, \Phi_n)$

1 $C \leftarrow A + N$

`// 1/Φ1(ω^i) obtained via precomputation`

2 $(C_1, C_2, \dots, C_{N-1}) \leftarrow (1/\Phi_1(\omega), 1/\Phi_1(\omega^2), \dots 1/\Phi_1(\omega^{N-1})$

`// write DFT of z − 1 to A`

3 $\lambda \leftarrow 1$, **for** $i \leftarrow 0$ **to** $N-1$ **do** $A_i \leftarrow \lambda - 1$, $\lambda \leftarrow \lambda \cdot \omega$

4 **for** $j \leftarrow 1$ **to** k **do**

5 **for** $i \leftarrow 1$ **to** $N-1$ **do** $A_i \leftarrow |A_i|$

6 $i \leftarrow 1$, $p \leftarrow p_i$

7 **while** $i < N$ **do**

8 **while** $A_i < 0$ **do** $i \leftarrow i + 1$

9 $x \leftarrow A_i$, $y \leftarrow C_i$, $l \leftarrow i$

10 **while** $lp \neq i \bmod N$ **do**

11 $z \leftarrow A_l$, $m \leftarrow lp \bmod N$

12 $A_l \leftarrow -(C_l A_m \bmod q)$, $C_l \leftarrow -(zC_m \bmod q)$

13 $l \leftarrow m$

14 $A_l \leftarrow xC_l$, $C_l \leftarrow A_l y$, $i \leftarrow i + 1$

15 **if** $k = 1$ **then** $A_0 \leftarrow p_1 \bmod N$ **else** $A_0 \leftarrow 1$ `// Lemma 2.4`

return A

Chapter 3

Calculating $\Phi_n(z)$ as a truncated power series

The FFT-based approaches of chapter 2 proved most useful in computing $\Phi_n(z)$ and gave us many new results on cyclotomic polynomials of large height; however, it was surpassed by the sparse power series algorithm, whose implementation is roughly two orders of magnitude (i.e. 100 times) faster than any FFT-based approach. In this chapter we present the original sparse power series (SPS) method, and the three key improvements we made to the algorithm, ultimately resulting in a fast, recursive algorithm for computing $\Phi_n(z)$.

3.1 A useful identity of $\Phi_n(z)$

3.1.1 The Möbius Inversion Formula

Lemma 3.1 (See Cojocaru and Murty [11])**.**

$$\sum_{d|n} \mu(d) = \begin{cases} 1 & \text{if } \ n = 1 \\ 0 & \textit{otherwise}. \end{cases} \tag{3.1}$$

This is a well-known result. We give a simple proof.

Proof of Lemma 3.1. We prove by induction on the factors of n. The base case $\mu(1) = 1$ is trivial. Now suppose (3.1) holds for m, and consider the case $n = mp > 1$, for some prime

p. If p divides m, then clearly $m > 1$; moreover, the squarefree divisors of n are exactly the squarefree divisors of m, and thus

$$\sum_{d|n} \mu(d) = \sum_{d|m} \mu(d), \tag{3.2}$$

the latter sum being zero by hypothesis. If p does not divide m, then the squarefree divisors of n are all d of the form e or ep, where e is a squarefree divisor of m. As such,

$$\begin{aligned} \sum_{d|n} \mu(d) &= \sum_{d|m} \mu(d) + \sum_{d|m} \mu(dp), \\ &= \sum_{d|m} \mu(d) + \sum_{d|m} \mu(d)\mu(p) \qquad \text{as } \mu \text{ is multiplicative,} \\ &= (1 + \mu(p)) \sum_{d|m} \mu(d) = 0. \end{aligned} \tag{3.3}$$

□

Theorem 3.2 (The Möbius Inversion Formula [11])**.** *If* $F(n) = \prod_{d|n} f(d)$ *for* $n \geq 1$*, then* $f(n) = \prod_{d|n} F(n/d)^{\mu(d)}$ *for* $n \geq 1$.

Proof. Suppose that $F(n) = \prod_{d|n} f(d)$ for $n \geq 1$. Then

$$\prod_{d|n} F(n/d)^{\mu(d)} = \prod_{d|n} \prod_{e|\frac{n}{d}} f(e)^{\mu(d)}. \tag{3.4}$$

As e divides $\frac{n}{d}$ if and only if d divides $\frac{n}{e}$, (3.4) simplifies to

$$\prod_{e|n} \prod_{d|\frac{n}{e}} f(e)^{\mu(d)} = \prod_{e|n} f(e)^{\sum_{d|(n/e)} \mu(d)}. \tag{3.5}$$

Applying Lemma 3.1, the exponent in (3.5) $\sum_{d|(n/e)} \mu(d)$ is nonzero only if $e = n$, and the product simplifies to $f(n)$, completing the proof. □

Applying Theorem 3.2 to the equality $z^n - 1 = \prod_{d|n} \Phi_d(z)$ gives us an identity for $\Phi_n(z)$.

Corollary 3.3.

$$\Phi_n(z) = \prod_{d|n} (z^{n/d} - 1)^{\mu(d)}. \tag{3.6}$$

For $n > 1$, the number of squarefree divisors of n is even, hence

$$\Phi_n(z) = \prod_{d|n} (1 - z^{n/d})^{\mu(d)}. \tag{3.7}$$

As an example, for $105 = 3 \cdot 5 \cdot 7$,

$$\Phi_{105}(z) = \frac{(1 - z^{105})(1 - z^3)(1 - z^5)(1 - z^7)}{(1 - z^{15})(1 - z^{21})(1 - z^{35})(1 - z)}.$$

This identity is key for the sparse power series algorithm, which we present in the next section.

3.2 The sparse power series algorithm

In every variant of the SPS algorithm, we compute $\Phi_n(z)$ as the product (3.7); however, in subsequent identities it is oftencase more convenient to write $\Phi_n(z)$ as some product of terms of the form $(z^d - 1)^{\pm 1}$. As we refer to these terms often, we will call the $(1 - z^d)^{\pm 1}$ (alternatively, $(z^d - 1)^{\pm 1}$) comprising $\Phi_n(z)$ the *subterms* of $\Phi_n(z)$. Using that the power series expansion of $(1 - z^d)^{-1}$ is $1 + z^d + z^{2d} + \cdots$, we rewrite (3.7) as

$$\Phi_n(z) = \prod_{d|n, \mu(d)=1} (1 - z^{n/d}) \prod_{d|n, \mu(d)=-1} (1 + z^d + z^{2d} + \dots) \qquad (\text{for } n > 1) \tag{3.8}$$

As with the previous algorithm we use a dense representation for $\Phi_n(z)$: an array of integers $[a(0), a(1), \dots]$, where we store the coefficient of the term of degree i in $a(i)$. We call this array the *coefficient array*.

Multiplication by $1 - z^d$ on a dense polynomial can be done with a single pass over an array. Division by $1 - z^d$, however, implemented naively, could potentially be quadratic time. The sparseness of each term in (3.7) lends itself to fast power series arithmetic. Suppose we have a power series $B(z) = b(0) + b(1)z + b(2)z^2 + \cdots + b(D)z^D$, truncated to degree D, and we want to compute $C(z) = \sum_{i=0}^{D} c(i)z^i = B(z) \cdot (1 - z^d)^{-1} \bmod z^{D+1}$. In which case, as $(1 - z^d)^{-1} = 1 + z^d + z^{2d} + \dots$,

$$c(i) = \sum_{j \equiv i \bmod d, j<i} b(i)z^i = \begin{cases} b(i) & \text{if } i < d \\ b(i) + c(i - d) & \text{otherwise.} \end{cases} \tag{3.9}$$

Thus we can multiply $B(z)$ by either $(1 - z^d)$ and $(1 - z^d)^{-1}$ in $\mathcal{O}(D)$ addition and subtraction operations in $\mathbb{Z}$. More importantly, we can easily both divide and multiply a truncated power series by $(1 - z^d)$ in memory. That is, if we have the coefficients of $B(z)$ stored in an array, we can write the truncated power series of $B(z)(1-z^d)$ or $B(z)/(1-z^d)$ to the array without using additional storage. This is how we compute $\Phi_n(z)$ in procedure SPS. A squarefree product of k primes $n = p_1p_2\cdots p_k$ has 2^k divisors, thus $\Phi_n(z)$ has 2^k subterms. Multiplying or dividing by one subterm $1 - z^d$ requires $\mathcal{O}(\phi(n))$ operations (lines 5 and 7 in procedure SPS). As such, SPS requires some $\mathcal{O}(2^k \cdot \phi(n))$ addition and subtraction operations in $\mathbb{Z}$. This algorithm appears in [2].

We truncate to degree $\phi(n)/2$ in Algorithm SPS, thus retrieving only the lower half of the coefficients of $\Phi_n(z)$. This is because, using the palindromic property of the cyclotomic coefficients (Lemma 1.13), it is trivial to obtain the terms of higher degree.

Procedure SPS(n) : Computing $\Phi_n(z)$ as a quotient of sparse power series

The Sparse Power Series (SPS) Algorithm

Input: $n = p_1p_2\cdots p_k$, a product of k distinct primes

Output: SPS(n) returns the first half of the coefficients of $\Phi_n(z)$

```
1 D ⟵ φ(n)/2 + 1, a(0) ⟵ 1
2 for 1 ≤ i ≤ D do a(i) ⟵ 0
3 for d|n do
4     if μ(n/d) = 1 then                          // multiply by 1 − z^d
5         for i = D down to d by −1 do a(i) ⟵ a(i) − a(i − d)
6     else                                        // divide by 1 − z^d
7         for i = d to D do a(i) ⟵ a(i) + a(i − d)
return a(0), a(1), ..., a(φ(n)/2)
```

3.2.1 The sparse power series algorithm for $\Psi_n(z)$

As $\Phi_n(z)\Psi_n(z) = z^n - 1$, it follows from (3.7) that

$$\Psi_n(z) = -\prod_{d|n, d<n} (1 - z^d)^{-\mu(n/d)}, \tag{3.10}$$

which leads us to an analogous algorithm for $\Psi_n(z)$, as described by procedure $SPS-$

Psi. We truncate to degree $(n-\phi(n))/2$, half the degree of $\Psi_n(z)$, because the coefficients of $\Psi_n(z)$ are antipalindromic. We will use this procedure in a subsequent algorithm.

Procedure SPS-Psi(n) : Computing $\Psi_n(z)$ as a quotient of sparse power series

The Sparse Power Series (SPS) Algorithm for $\Psi_n(z)$

Input: $n = p_1 p_2 \cdots p_k$, a product of k distinct primes

Output: SPS-Psi returns he first half the coefficients of $\Psi_n(z)$

1 $D \longleftarrow \lfloor \frac{n-\phi(n)}{2} \rfloor$, $a(0) \longleftarrow -1$
2 **for** $1 \leq i \leq D$ **do** $a(i) \longleftarrow 0$
3 **for** $d|n$ such that $d < n$ **do**
4 **if** $\mu(\frac{n}{d}) = -1$ **then** `// multiply by` $1 - z^d$
5 **for** $i = D$ **down to** d **by** -1 **do** $a(i) \longleftarrow a(i) - a(i-d)$
6 **else** `// divide by` $1 - z^d$
7 **for** $i = d$ **to** D **do** $a(i) \longleftarrow a(i) + a(i-d)$

return $a(0), a(1), \ldots, a((n-\phi(n))/2)$

Given squarefree $n = p_1 p_2 \ldots p_k$, the SPS algorithm requires $\mathcal{O}(2^k(n-\phi(n)))$ additions and subtractions to compute $\Psi_n(z)$.

3.3 Improving the sparse power series method by further truncating degree

In this section we introduce the first of several improvements to the SPS algorithm. We will denote by SPSj the jth version of the sparse power series algorithm for $\Phi_n(z)$, where SPS1 refers to the original sparse power series algorithm.

3.3.1 A measure to compare SPS-based algorithms

If we have the prime factorization of squarefree $n = p_1 p_2 \cdots p_k$, then we can generate a divisor $d|n$ and the value $\mu(n/d)$ in $\mathcal{O}(k)$ operations. Thus the overhead of constructing and handling the 2^k divisors of n in the SPS algorithm is $\mathcal{O}(2^k \cdot k)$ operations in $\mathbb{Z}$. This cost is negligible. The brunt of the work of procedure SPS occurs on lines 5 and 7, where

we multiply by the subterm $(1 - z^d)$ and $(1 - z^d)^{-1}$, respectively. As stated prior, n a squarefree product of k primes, $\Phi_n(z)$ is a quotient comprised of some 2^k subterms $1 - z^d$. As such, SPS effectively computes 2^k distinct power series, each a quotient of subterms $1 - z^d$, each truncated to degree $\phi(n)/2$. Thus the cost of multiplying and dividing the subterms of $\Phi_n(z)$ is $\mathcal{O}(2^k \cdot \phi(n))$.

We have that the slowdown in computing $\Phi_{np}(z)$ compared to $\Phi_n(z)$ is twofold. By introducing a new prime factor p we double the number of subterms $(1 - z^d)$ that make our cyclotomic polynomial. In addition, the degree of $\Phi_{np}(z)$ is $p - 1$ times that of $\Phi_n(z)$. We observe additional slowdown with cyclotomic polynomials of increasing degree as we require larger (and slower) caches in the cache hierarchy to store the array of coefficients.

Procedure SPS2(n) : The first revision of the SPS algorithm

The Improved Sparse Power Series (SPS2) Algorithm

Input: $n = mp$, a squarefree, odd integer with greatest prime divisor p

Output: $a(0), \ldots, a(\frac{\phi(n)}{2})$, the first half of the coefficients of $\Phi_n(z)$

`// Compute first half of` $\Psi_m(z)$

$a(0), a(1), \ldots, a(\lfloor \frac{n-\phi(m)}{2} \rfloor) \longleftarrow$ SPS-Psi(m)

`// Construct other half of` $\Psi_m(z)$ `using Lemma 1.13`

$D \longleftarrow \max(m - \phi(m), \frac{\phi(n)}{2})$

for $i = \lceil \frac{m-\phi(m)}{2} \rceil$ **to** D **do** $a(i) \longleftarrow -a(m - \phi(m) - i)$

`// Multiply by` $\Phi_m(z^p)$

$D \longleftarrow \frac{\phi(n)}{2}$

$a(m - \phi(m) + 1), a(m - \phi(m) + 2), \ldots, a(D) \longleftarrow 0$

for $d|m$ such that $0 < d < m$ **do**

  if $\mu(\frac{n}{d}) = -1$ **then**

    for $i = D$ **down to** dp **by** -1 **do** $a(i) \longleftarrow a(i) - a(i - dp)$

  else

    for $i = dp$ **to** D **do** $a(i) \longleftarrow a(i) + a(i - dp)$

for $i = m$ **to** D **do** $a(i) \longleftarrow a(i) + a(i - m)$

return $a(0), a(1), \ldots a(D)$

Let $n = p_1 p_2 \cdots p_k$ be a product of k distinct odd primes. Let $d_1, d_2, \ldots, d_{2^k}$ be the divisors of n in the order by which an SPS algorithm SPS_j iterates through them all. The original SPS algorithm computes the truncated power series of

$$f_s(z) = \prod_{i=1}^{s} (1 - z^{d_i})^{\mu(n/d_i)} \tag{3.11}$$

for $0 \leq s \leq 2^k$, all truncated to degree $\phi(n)/2$. If, however, for some t, $f_t(z)$ is a polynomial of degree D_t, then we need only truncate f_s, where $s \leq t$, to degree at most D_t. Moreover, if f_t is a polynomial, then $f_t(z)$ is a product of cyclotomic polynomials satisfying Lemma 1.13, hence we need only truncate to degree $\lfloor D_t/2 \rfloor$ in order to obtain all the terms of f_t.

More generally, if we know $f_{t_1}, \ldots, f_{t_l}$ are polynomials of degree $D_{t_1}, D_{t_2}, \ldots, D_{t_l}$, then for $s \leq \min_{1 \leq i \leq l} t_i$, we need only truncate $f_s(z)$ to degree $\lfloor D/2 \rfloor$, where $D = \min_{1 \leq i \leq l} D_{t_i}$. We call the degree to which we truncate f_s the *degree bound*. If we truncate f_i to degree D_i, then multiplying f_{i-1} by $1 - z^{d_i}$ will take $\mathcal{O}(D_i)$ arithmetic operations. We consider

$$T_j(n) = \sum_{i=1}^{2^k} D_i \tag{3.12}$$

a suitable measure of the relative performance of SPSj. In actuality, when we multiply by $(1 - z^{d_i})$ or $(1 - z^{d_i})^{-1}$, we do addition or subtraction on the coefficients of terms of degree c, $d \leq c \leq D_i$. As such,

$$T'_j(n) = \sum_{i=1}^{2^k} \max(D_i + 1 - d_i, 0), \tag{3.13}$$

is the exact total number of addition and subtraction operations on the coefficients $a(i)$ that SPSj requires. For the purposes of our analysis, the sum (3.12) is a useful means of comparing different versions of the SPS algorithm, while, in addition, being less cumbersome than (3.13). We later compute (3.13) to give an exact comparison of the number of arithmetic operations on the coefficients $a(i)$.

Definition 3.4. *We call $T_j(n)$ (3.12) and $T'_j(n)$ (3.13) the* ***measure of work*** *and* ***exact measure of work****, respectively, to compute $\Phi_n(z)$ using algorithm SPSj.*

E.g. For the original sparse power series algorithm and given n, an odd, squarefree product of k primes, we have $T_1(n) = 2^k \cdot \phi(n)/2 = 2^{k-1}\phi(n)$.

We aim to improve the SPS algorithm by ordering the divisors of n in an intelligent manner which allows us to decrease the degree bound D_i, and hence our measure, wherever possible over the computation of $\Phi_n(z)$.

3.3.2 A first improvement

Let p be the largest prime divisor of n and suppose $m = n/p$ is greater than 1. In which case $\Phi_{mp}(z) = \Phi_m(z^p)/\Phi_m(z)$ by Lemma 1.19, which we reexpress as

$$\Phi_{mp}(z) = -\Psi_m(z) \cdot \Phi_m(z^p) \cdot \frac{1}{1-z^m}. \tag{3.14}$$

In light of (3.7) and (3.10), we can express (3.14) as quotients of subterms $(1 - z^d)$.

$$\begin{aligned} \Phi_n(z) &= \left(\prod_{d|m, d<m} (1-z^d)^{-\mu(\frac{m}{d})} \right) \cdot \left(\prod_{d|m} (1-z^{dp})^{\mu(\frac{m}{d})} \right) \cdot \frac{1}{(1-z^m)}, \\ &= \left(\prod_{d|m, d<m} (1-z^d)^{\mu(\frac{n}{d})} \right) \cdot \left(\prod_{d|m} (1-z^{dp})^{\mu(\frac{n}{dp})} \right) \cdot \frac{1}{(1-z^m)}. \end{aligned} \tag{3.15}$$

Thus to compute $\Phi_n(z)$, we can compute $\Psi_m(z)$, the leftmost product of (3.15) to degree $\lfloor \frac{m-\phi(m)}{2} \rfloor$, use the antipalindromic property of $\Psi_m(z)$ to reconstruct its remaining coefficients, and then multiply by the remaining subterms $(1 - z^d)^{\pm 1}$ as we would in the SPS algorithm. Procedure SPS2 describes the method.

We see for this variant of the SPS algorithm, that the measure of work reduces to

$$T_2(n) = \Big(2^{k-1} - 1\Big)\Big(\frac{m-\phi(m)}{2}\Big) + \Big(2^{k-1} + 1\Big)\Big(\frac{\phi(n)}{2}\Big), \tag{3.16}$$

$$\approx 2^{k-2}(\phi(n) + m - \phi(m)). \tag{3.17}$$

While, again, this improvement does not change the asymptotic running time of the algorithm, it saves us roughly a factor of two time in most tractable cases.

3.4 Calculating $\Phi_n(z)$ by way of a product of inverse cyclotomic polynomials

We are able to achieve yet better performance by an approach we will call the iterative SPS algorithm or SPS3. To that end we establish the next identity. Let $n = p_1 p_2 \cdots p_k$, a product of k distinct odd primes. For $1 \le i \le k$, let

$$m_i = p_1 p_2 \cdots p_{i-1} \qquad \text{and} \qquad e_i = p_{i+1} \cdots p_k. \tag{3.18}$$

We set $m_1 = e_k = 1$, and let $e_0 = n$. Note that $n = e_i p_i n_i$ for $1 \le i \le k$. In addition, $e_{i-1} = p_i e_i$ and $m_{i+1} = m_i p_i$. Then by repeated application of Lemma 1.19, we have

$$\begin{aligned}
\Phi_n(z) &= \frac{\Psi_{m_k}(z^{e_k})}{z^{n/p_k} - 1} \Phi_{m_k}(z^{e_{k-1}}) \\
&= \frac{\Psi_{m_k}(z^{e_k})}{(z^{n/p_k} - 1)} \frac{\Psi_{m_{k-1}}(z^{e_{k-1}})}{(z^{n/p_{k-1}} - 1)} \Phi_{m_{k-1}}(z^{e_{k-2}}) \\
&\cdots \\
&= \frac{\Psi_{m_k}(z^{e_k})}{(z^{n/p_k} - 1)} \cdots \frac{\Psi_{m_2}(z^{e_2})}{(z^{n/p_2} - 1)} \frac{\Psi_{m_1}(z^{e_1})}{(z^{n/p_1} - 1)} \Phi_1(z^{e_0}) \\
&= \left(\prod_{j=1}^{k} \frac{\Psi_{m_j}(z^{e_j})}{z^{n/p_j} - 1} \right) \cdot (z^n - 1).
\end{aligned} \tag{3.19}$$

As $\Psi_{m_1}(z^{e_1}) = \Psi_1(z^{e_1}) = 1$, we have

$$\Phi_n(z) = \left(\prod_{j=2}^{k} \Psi_{m_j}(z^{e_j}) \right) \cdot \left(\prod_{j=1}^{k} (z^{n/p_j} - 1)^{-1} \right) \cdot (z^n - 1) \tag{3.20}$$

For example, for $n = 105 = 3 \cdot 5 \cdot 7$,

$$\Phi_{105}(z) = \Psi_{15}(z)\Psi_3(z^7) \cdot (z^{15} - 1)^{-1}(z^{21} - 1)^{-1}(z^{35} - 1)^{-1} \cdot (z^{105} - 1) \tag{3.21}$$

As with procedure SPS2, we first calculate half the terms of $\Psi_{m_k}(z^{e_k}) = \Phi_{p_1 p_2 \ldots p_{k-1}}(z)$, those with degree at most $\lfloor \phi(m_k)/2 \rfloor$. However, unlike our previous method, we then instead iteratively compute the product

$$\Psi_{m_k}(z^{e_k}) \cdots \Psi_{m_2}(z^{e_2}), \tag{3.22}$$

Procedure SPS3(n) : The second revision of the SPS algorithm

The Iterative Sparse Power Series (SPS) Algorithm

Input:

•$n = p_1 p_2 \dots p_k$, a squarefree product of k primes

•$e_i = \prod_{j=i+1}^{k} p_j$ and $m_i = \prod_{j=1}^{i-1} p_j$, for $1 \le i \le k$

Output: $a(0), \dots, a(\frac{\phi(n)}{2})$, the first half of the coefficients of $\Phi_n(z)$

$a(0), a(1), a(2), \dots, a(\phi(n)/2) \longleftarrow 1, 0, 0, \dots, 0$

$D_f \longleftarrow 0, \quad D_g \longleftarrow m_k - \phi(m_k), \quad D \longleftarrow \min(D_g, \phi(n))$

for $j = k$ **down to** 2 **do**

  `// Multiply by` $\Psi_{m_j}(z^{e_j})$`; truncate to degree` $\lfloor D/2 \rfloor$

  for $d | m_j$ such that $d < m_j$ **do**

    if $\mu(\frac{n}{d}) = 1$ **then**

      for $i = \lfloor D/2 \rfloor$ **down to** de_j **by** -1 **do** $a(i) \longleftarrow a(i) - a(i - de_j)$

    else

      for $i = de_j$ **to** $\lfloor D/2 \rfloor$ **do** $a(i) \longleftarrow a(i) + a(i - de_j)$

  $D_f \longleftarrow D_g$

  if $j > 2$ **then** $D_g \longleftarrow D_g + (m_{j-1} - \phi(m_{j-1}))e_{j-1}, \quad D \longleftarrow \min(D_g, \phi(n))$

  else $D \longleftarrow \phi(n)$

  `// Use Lemma 1.15 to get higher-degree terms`

  for $i = \lfloor D_f/2 \rfloor + 1$ **to** $\lfloor D/2 \rfloor$ **do** $a(i) \longleftarrow (-1)^{D_f} a(D_f - i)$

`// Divide by` $(1 - z^{n/p_j})$`; truncate to degree` $\phi(n)/2$

for $j = 1$ **to** k **do**

  for $i = n/p_j$ **to** $\phi(n)/2$ **do** $a(i) \longleftarrow a(i) + a(i - n/p_j)$

return $a(0), a(1), \dots, a(\phi(n)/2)$

from the leftmost term right, i.e. we multiply the $\Psi_{m_i}(z^{e_i})$ in order of decreasing index.

We leverage Lemma 1.15 again when computing the prouduct (3.22). Suppose we have half the terms of $f(z) = \prod_{i=j+1}^{k} \Psi_{m_i}(z^{e_i})$, for some $j \ge 2$ and we want to compute $g(z) = f(z) \cdot \Psi_{m_j}(z^{e_j})$ (truncated to some degree), towards the aim of obtaining $\Phi_n(z)$. As

both $f(z)$ and $g(z)$ have the (anti)palindromic property of Lemma 1.15, when computing $g(z)$ we need to truncate to degree at most $\lfloor D/2 \rfloor$, where D is the lesser of

$$D_g \prod_{i=j-1}^{k} (m_i - \phi(m_i))e_i \quad \text{and} \quad \phi(n), \tag{3.23}$$

the former of which is the degree of $g(z)$, the latter being half the degree of $\Phi_n(z)$. Thus we apply Lemma 1.15 to generate the higher-degree terms of $f(z)$ up to degree D. Once we have the product (3.22) we then apply the palindromic property again to generate the coefficients up to degree $\phi(n)/2$, provided we do not have them already. We then divide by the subterms $(z^{n/p_j} - 1)$ for $1 \leq j \leq k$, truncating, again, to degree $\phi(n)/2$. We give pseudocode for the method in procedure SPS3. We assume m_j and e_j were precomputed in the procedure.

As we build the product $g(z)$ (3.22) we have to truncate to increasing degree as the computation progresses. However, given the manner we organize the product (3.22), every time we have to increase this degree bound we nearly halve the remaining subterms. We first compute $\Psi_{m_k}(z^{e_k})$, which comprises $2^{k-1} - 1$ of the 2^k subterms of $\Phi_n(z)$; we then multiply by $\Psi_{m_{k-1}}(z^{e_{k-1}})$, which has some $2^{k-2} - 1$ subterms; and so forth. Thus the problem is organized in a manner for which the biggest potential gains are awarded to the most subterms possible.

3.4.1 A note on the performance of the iterative SPS algorithm

We first try to answer: for what subterms of $\Phi_n(z)$ is the degree bound lower using the iterative SPS algorithm as opposed to SPS or SPS2? In the computation of $g(z)$ (3.22), the additional restriction that we bound the degree by $\phi(n)/2$ is not redundant. It follows from (3.19) that the degree of $g(z)$ is

$$\phi(n) + \left(\sum_{p|n} \frac{n}{p} \right) - n, \tag{3.24}$$

thus the degree of $g(z)$ is greater than that of $\Phi_n(z)$ provided $\sum_{p|n} 1/p > 1$. Thus for sufficiently composite n, we begin to lose the gains of the iterative SPS algorithm for some subterms of $\Phi_n(z)$. For $\Phi_n(z)$ that are presently feasible to compute, however, it is seldom

the case that the inequality $\sum_{d|n} 1/p > 1$ holds. The smallest odd, squarefree n for which it does is $n = 3,234,846,615$, the product of the first nine odd primes. Thus for $\Phi_n(z)$ of order 8 or less, we see gains with every subterm of $g(z)$ when compared to the SPS algorithm (less those appearing in $\Psi_{m_k}(z^{e_k})$ if we compare with $\Psi_m(z)$). For $\Phi_n(z)$ of higher order, we still see gains for subterms appearing in

$$\prod_{j=k-7}^{k} \Psi_{m_j}(z^{e_j}), \tag{3.25}$$

which comprises some $2^k - 2^{k-8} - 8$ of the 2^k subterms of $\Phi_n(z)$. Thus we have gains compared to the SPS algorithm with all but a negligible fraction of the subterms of $\Phi_n(z)$. Compared to SPS2, we have gains in roughly half of the subterms of $\Phi_n(z)$.

In the case of the iterative SPS (procedure SPS3), the measure of work to compute $\Phi_n(z)$ reduces to

$$T_3(n) = \sum_{j=2}^{k} \left(\left(2^{j-1} - 1\right) \cdot \left\lfloor \min\left(\sum\nolimits_{i=j}^{k} e^i (m_i - \phi(m_i)), \phi(n) \right) \right\rfloor \right) + (k+1)\phi(n)/2.$$

This does not, unfortunately, illuminate the improved performance as a result of the new approach. We give values of the exact measure of work $T_3'(n)$ for explicit n (Table 3.1, page 54) as well as numerous timings to convey how big an improvement the iterative SPS is over the previous versions (Section 5.1).

3.5 Calculating $\Phi_n(z)$ and $\Psi_n(z)$ recursively

The iterative SPS approach depended on the identity (3.20), which describes $\Phi_n(z)$ in terms of a product of inverse cyclotomic polynomials of decreasing order and index. We derive an analog for $\Psi_n(z)$. Let m_i and e_i be as defined in Section 3.4, and again let $n = p_1 p_2 \cdots p_k$ be a product of k distinct odd primes where $p_1 < p_2 < \cdots < p_k$.

Then by repeated application of Lemma 1.19:

$$\begin{aligned}\Psi_n(z) &= \Phi_{m_k}(z^{e_k})\Psi_{m_k}(z^{e_{k-1}}),\\ &= \Phi_{m_k}(z^{e_k})\Phi_{m_{k-1}}(z^{e_{k-1}})\Psi_{m_{k-1}}(z^{e_{k-2}}),\\ &= \Phi_{m_k}(z^{e_k})\Phi_{m_{k-1}}(z^{e_{k-1}})\Phi_{m_{k-1}})(z^{e_{k-2}})\Psi_{m_{k-2}}(z^{e_{k-2}}),\\ &\dots\\ &= \Phi_{m_k}(z^{e_k})\cdots\Phi_{m_1}(z^{e_1})\Psi_{m_1}(z^{e_1}).\end{aligned} \tag{3.26}$$

As $m_1 = 1$ and $\Psi_1(z) = 1$, we thus have that

$$\Psi_n(z) = \prod_{j=1}^{k} \Phi_{m_j}(z^{e_j}). \tag{3.27}$$

The identities (3.20) and (3.27) suggest a recursive method of computing $\Phi_n(z)$. Consider the example of $\Phi_n(z)$, for $n = 1155 = 3 \cdot 5 \cdot 7 \cdot 11$. To obtain the coefficients of $\Phi_{1105}(z)$, procedure $SPS3$ constructs the product

$$\Psi_{105}(z)\Psi_{15}(z^{11})\Psi_3(z^{77})(z^{385}-1)^{-1}(z^{231}-1)^{-1}(z^{165}-1)^{-1}(z^{105}-1)^{-1}(z^{1155}-1) \tag{3.28}$$

from left to right. However, in light of (3.27), we know this method computes $\Psi_{105}(z)$ in a wasteful manner. We can treat $\Psi_{105}(z)$ as a product of cyclotomic polynomials of smaller index:

$$\begin{aligned}\Psi_{105}(z) &= \Phi_{15}(z)\Phi_5(z^7)\Phi_1(z^{35}),\\ &= \Phi_{15}(z)\Phi_5(z^7)(1-z^{35})\end{aligned} \tag{3.29}$$

One could apply (3.20) yet again, now to $\Phi_{15}(z)$ and $\Phi_5(z^7)$, giving us

$$\begin{aligned}\Phi_{15}(z) &= \Psi_5(z)(z^5-1)^{-1}(z^3-1)^{-1}(z^{15}-1),\\ &= (z-1)\big[(z^5-1)^{-1}(z^3-1)^{-1}(z^{15}-1)\big],\end{aligned} \tag{3.30}$$

Upon computing $\Psi_{105}(z)$, we can break the next term of (3.28), $\Psi_{15}(z^{11})$ into smaller products in a similar fashion. We effectively compute $\Phi_n(z)$ by recursion into the factors of n. We call this approach the **recursive sparse power series** method, and we give a pseudocode implemetation in procedure SPS4 (page 52). This algorithm appeared in [3].

$SPS4$ effectively takes a product of cyclotomic polynomials $f(z)$ and multiplies it by either $\Phi_m(z^e)$ (or $\Psi_m(z^e)$) recursively as described above. If we are to multiply by $\Psi_m(z^e)$,

as $\Psi_m(z^e)$ is exactly a product of cyclotomic polynomials by (3.27), upon completion of our last recursive call, we are finished (line 7 of SPS4). If, however, we are to multiply by $\Phi_m(z^e)$, once we have completed our last recursive call, we need to divide and multiply by some additional subterms (lines 9 and 11), as is necessary by the identity (3.22). To avoid unnecessary recursion, we do not recurse in the SPS algorithm to multiply by $\Psi_1(z^e)$, as $\Psi_1(z) = 1$ (line 5).

Leveraging the palindromic property of cyclotomic coefficients in the recursive SPS method is not as immediate as in previous cases. We need to consider the state of the system (i.e. what degree we have truncated to) upon calling and returning from $SPS4$. The difference, with respect to leveraging palindromicity, between the iterative SPS and the recursive SPS, is that in the former we truncate to the least degree of two polynomials, whereas in the recursive sparse power series case, we may bound by the least degree of many polynomials. Moreover, we need to know what degree to bound to at each level of recursion. Procedure SPS4 has an additional parameter, D, which serves as a bound on the degree.

As before, let $f(z)$ be a product of cyclotomic polynomials. Let D_f be the degree of $f(z)$ and suppose, while we are in some intermediate step of the computation of $\Phi_n(z)$ or $\Psi_n(z)$, that we have the first $\lfloor D_f/2 \rfloor$ terms of $f(z)$, and we want next to compute the terms of

$$g(z) = f(z) \cdot \Phi_m(z^e) \qquad (\text{or } f(z) \cdot \Psi_m(z^e)), \tag{3.31}$$

up to degree $\lfloor D/2 \rfloor$, for some $D \in \mathbb{N}$. D, for our purposes, the degree of some product of cyclotomic polynomials we will eventually obtain later at some previous level of recursion. If we let D_g be the degree of $g(z)$, then when computing $g(z)$ from $f(z)$ we need only compute terms up to $\lfloor D^*/2 \rfloor$, where $D^* = \min(D, D_g)$ (line 2). Thus when we recurse in SPS4, if $D_g < D$ we lower the degree bound from D to D_g.

To guarantee that we can obtain higher-degree terms whenever necessary we impose the following rule: If SPS4 is given $f(z)$ and is to output $g(z)$, we require that $f(z)$ is truncated to degree $\lfloor \min(D_f, D)/2 \rfloor$ on input, and that $g(z)$ is truncated to degree $\lfloor \min(D_g, D)/2 \rfloor$ on output. Note, in the case that the degree bound on $g(z)$ is always at least the bound on $f(z)$.

To calculate the first half of the coefficients of $\Phi_n(z)$, one would merely initialize the

Procedure SPS4(m, e, λ, D_f, D, A) : Multiply by $\Phi_m(z^e)$ or $\Psi_m(z^e)$

A recursive algorithm to multiply a product of cyclotomic polynomials by $\Phi_m(z^e)$ or $\Psi_m(z^e)$

Input:

• m, a positive, squarefree odd integer; λ, a boolean; $D \in \mathbb{Z}$, a bound on the degree

• D_f, the degree of $f(z)$, a product of cyclotomic polynomials partially stored in array A. D_f is passed by value.

• An array of integers $a = [a(0), a(1), a(2), \dots]$, for which $a(i)$ is initialized to the coefficient of the term of degree i of f for $0 \le i \le D'$, where $D' = min(D_f, D)$. a is passed by reference.

Result: If λ is true, we compute $g(z) = f(z)\Phi_m(z^e)$, otherwise, we compute $g(z) = f(z)\Psi_m(z^e)$. In either case we truncate the result to degree $\lfloor D^*/2 \rfloor$, where D^* is the lesser of D and $D_g = \deg(g)$. We write the coefficients of g to array a, and return the degree of g, D_g.

1 **if** λ **then** $D_g \longleftarrow D_g + \phi(m)e$ **else** $D_g \longleftarrow D_g + (m - \phi(m))e$

2 $D^* \longleftarrow \min(D_g, D)$, $\quad e^* \longleftarrow e$, $\quad m^* \leftarrow m$

3 **while** $m^* > 1$ **do**

4 $\quad p \longleftarrow$ (largest prime divisor of m^*), $\quad m^* \longleftarrow m^*/p$

$\quad$ `// We do not recurse to multiply by` $\Psi_1(z) = 1$.

5 $\quad$ **if** $m^* > 1$ or λ is false **then** $D_f \longleftarrow$ SPS4(m^*, e^*, not λ, D_f, D^*, a)

6 $\quad e^* \longleftarrow e^*p$

7 **if** λ is false **then return** D_g `// We have multiplied by` $\Psi_m(z^e)$

`// Get higher degree terms as needed`

8 **for** $\lfloor D_f/2 \rfloor + 1$ **to** $\lfloor D^*/2 \rfloor$ **do** $a(i) \longleftarrow (-1)^{D_f} a(D_f - i)$

`// Divide by` $(1 - z^{me/p})$ `for` $p|m$

9 **for** each prime $p|m$ **do**

10 $\quad$ **for** $i = (me/p)$ **to** $\lfloor D^*/2 \rfloor$ **do** $a(i) \longleftarrow a(i) + a(i - me/p)$

11 **for** $i = \lfloor D^*/2 \rfloor$ **down to** d **do** $a(i) \longleftarrow a(i) - a(i - me)$ `//` $\times$ `by` $1 - z^{me}$

return D_g

values of array a as

$$(a(0), a(1), a(2), \ldots, a(\phi(n)/2) = (1, 0, 0, \ldots, 0)$$

and call SPS4(n, 1, true, 0, $\phi(n), a$). Similarly, to calculate the first half of $\Psi_n(z)$ we would initialize

$$(a(0), a(1), a(2), \ldots, a(\lfloor \tfrac{n-\phi(n)}{2} \rfloor)) = (-1, 0, 0, \ldots, 0)$$

and call SPS4(n, 1, false, 0, $n - \phi(n)$, a).

We found that the overhead of the recursive function calls is negligible for tractable cases (i.e. for $\Phi_n(z)$ of order less than 9). That is, if we run our implementation of the SPS4 algorithm on $\Phi_n(z)$ of order 9, removing the work done on the coefficient array appearing on lines $8-11$, then the algorithm takes less than one hundredth of a second.

3.5.1 Implementation details of the recursive SPS algorithm

In procedure SPS4 we oftencase need the prime divisors of the input m. It is obviously wasteful to factor m every time we recurse. To compute $\Phi_n(z)$ or $\Psi_n(z)$ for squarefree n, we first precompute the factorization of n and store it in a global array $P = [p_1, p_2, \ldots, p_k]$. Upon calling SPS4, every subsequent recursive call will multiply by some (inverse) cyclotomic polynomial of index $m|n$. Our implementation of the recursive sparse power series algorithm has an additional argument, $B = [b_1, b_2, \ldots, b_k]$, a series of bits, that, given P, gives us the factorization of m. We set b_i to 1 if p_i divides m, and zero otherwise. For all tractable cases, B can fit in two bytes.

Thus, in the while loop on line 3 in SPS4, we take a copy of B, call it B^*, and scan it for nonzero bits. Each time we find a nonzero bit we set that bit to zero, and pass the value of B^* to the recursive call to 5. We continue in this fashion until all the bits of B^* are set to zero. We get the prime divisors again in a similar fashion on line 9.

3.5.2 A comparison of the different SPS algorithms

It should be noted that we have yet to prove whether SPS4 is asymptotically faster than SPS3, or even the original SPS algorithm. It is a nontrivial problem to show how $T_4(n)$ behaves asympotically. Trivially, we have that, for n a product of k distinct primes,

$$\phi(n)2^{k-1} \leq T_4(n) \leq (k+1)\phi(n), \tag{3.32}$$

Table 3.1: The number of additions and subtraction operations on our coefficient array to compute $\Phi_n(z)$, for n a product of k distinct primes, using SPS1-4

		exact measure of work (see §3.3.1)			
n	k	$T_1'(n)$	$T_2'(n)$	$T_3'(n)$	$T_4'(n)$
2,145	4	5,562	2511	1011	762
40,755	5	232,930	112,339	37,507	25,347
1,181,895	6	14,155,762	7,021,951	1,881,203	1,200,119
43,730,115	7	1,063,763,368	537,122,477	116,580,144	70,804,945
1,880,394,945	8	91,381,479,222	46,599,257,971	8,492,255,607	4,924,120,957

but we do not know where exactly in this range $T_4(n)$ tends. This constitutes future work. We find, in practice, that the recursive SPS is slightly faster than the iterative SPS; however, this improvement is not nearly as substantial as was the iterative SPS over prior versions. The measure of work is always less (or equal) using SPS4 over SPS1-3. We include the exact measure of work (i.e. the number of additions and subtractions on our array of coefficients) using SPS1-4 to compute select $\Phi_n(z)$ in Table 3.1. We see that the gains of SPS4 over SPS3 are more substantial for $\Phi_n(z)$ of higher order, i.e. $\Phi_n(z)$ for which n has more distinct odd prime divisors.

Chapter 4

Reduced-memory methods for computing $A(n)$

Calculating cyclotomic polynomials of very large degree, using either an FFT-based or SPS algorithm, can bode problematic, as oftentimes $\Phi_n(z)$ will not fit in main memory. In such a case, there are a variety of approaches to calculate $\Phi_n(z)$.

One approach is to calculate $\Phi_n(z)$ modulo primes p_i sufficiently small that we can fit $\Phi_n(z) \bmod p_i$ in memory and write the images to hard disk. We then reconstruct the coefficients of $\Phi_n(z)$ sequentially from the images of $\Phi_n(z) \bmod p_i$. We have already seen this approach in chapter 2. This minimizes the amount of computation we have to do on the hard disk.

For yet larger cyclotomic polynomials, we may not even be able to store the lower-half of its coefficients modulo a prime in memory. In this chapter we describe various approaches we used for computing such cyclotomic polynomials. In Section 4.1, we explain the big prime algorithm, which was used to compute the heights of a particular family of sparse cyclotomic polynomials of low height. In Section 4.2 and 4.3, we describe variations of the sparse power series algorithm which was used to compute cyclotomic polynomials whose coefficients require tens or hundreds of gigabytes of storage.

4.1 The big prime algorithm

As a motivating example, consider $\Phi_n(z)$ for

$$n = 2,576,062,979,535 = 3 \cdot 5 \cdot 29 \cdot 2609 \cdot 2269829.$$

This is smallest $n = p_1p_2p_3p_4p_5$, a product of five distinct odd primes, such that

$$p_k \equiv -1 \mod \prod_{i=1}^{k-1} p_i \text{ for } k = 2, 3, 4, 5. \tag{4.1}$$

Nathan Kaplan [20] asked whether this cyclotomic polynomial is flat. To our knowledge, no one has yet found a flat cyclotomic polynomial of order 5. This was a natural candidate to test for flatness. Kaplan [21] proved for $n = p_1p_2p_3$ satifying (4.1) for $k = 2, 3$ that $A(n) = 1$. In addition, for every odd $n < 3 \cdot 10^8$ of the form $n = p_1p_2p_3p_4$ satisfying (4.1) for $2 \leq k \leq 4$, $\Phi_n(z)$ is flat.

4.1.1 Computing coefficients of $\Phi_n(z)$ recursively

For our purposes, it is not always necessary that we retrieve all the coefficients of $\Phi_n(z)$ at once, particularly if we just want $A(n)$, the height of $\Phi_n(z)$. Indeed, for a cyclotomic polynomial with degree in the tens of billions or beyond, there is very little we can feasibly do with $\Phi_n(z)$, so there may be no purpose to store it in memory for further computation.

Let $n = mp$ be a squarefree, odd integer with largest prime divisor p. We can compute $A(n)$ by inspecting some of the coefficients of $\Phi_n(z)$ sequentially such that we only have to store m coefficients of $\Phi_n(z)$ at any one time. This algorithm takes $\mathcal{O}(m^2) = \mathcal{O}(\frac{n^2}{p^2})$ integer operations, provided we have $\Phi_m(z)$ and $\Psi_m(z)$. Clearly, such an algorithm works best for n with a large prime divisor, hence we affectionally call it the **big prime** algorithm.

Recall that $\Psi_n(z) = (z^n - 1)/\Phi_n(z)$. By Lemma 1.19,

$$\begin{aligned} \Phi_n(z) = \Phi_{mp}(z) = \frac{\Phi_m(z^p)}{\Phi_m(z)} &= \Phi_m(z^p) \cdot \Psi_m(z) \cdot (z^m - 1)^{-1} \\ &= \Phi_m(z^p) \cdot \Psi_m(z) \cdot (-1 - z^m - z^{2m} - \cdots). \end{aligned} \tag{4.2}$$

Write $\Phi_m(z) = \sum_{i=0}^{\phi(m)} b_i z^i$ and $\Psi_m(z) = \sum_{j=0}^{m-\phi(m)} c_j z^j$ and $\Phi_n(z) = \sum_{s=0}^{\phi(n)} a_n(s) z^s$.

From equation (4.2), we can express the coefficients a_s in terms of the b_i and c_j :

$$a_n(k) = - \sum_{\substack{ip+j \equiv k \bmod m \\ ip+j \le k}} b_i c_j. \tag{4.3}$$

This leads to the recurrence

$$a_n(k) = a_n(k-m) - \sum_{ip+j=k} b_i c_j, \tag{4.4}$$

which is the idea for big prime method (Algorithm 4.1).

Algorithm 4.1: The big prime algorithm for computing $A(n)$

Input:

- $n = mp$, a squarefree odd integer with largest prime divisor p
- $a = [a(0), a(1), \dots]$, an array of m integers,
- $b_0, b_1, \dots, b_{\phi(m)/2}$, the first half of the coefficients of $\Phi_m(z) = \sum_{i=0}^{\phi(m)} b_i z^i$,
- $c_0, c_1, \dots, c_{m-\phi(m)}$, the coefficients of $\Psi_m(z) = \sum_{i=0}^{m-\phi(m)} c_i z^i$

Output: $H = A(n)$, the height of $\Phi_n(z)$

```
a(0), a(1), ..., a(m-1) ← 0, 0, ..., 0
H ← 0
for i ⟵ 0 to ⌊φ(n)/2p⌋ do
    k ⟵ ip mod N
    for j ⟵ 0 to m − φ(m) do
        a(k) ← a(k) − b_i c_j,
        if j < p and |a(k)| > H then H ⟵ |a(k)|
        k ← k + 1 mod m
return H
```

The algorithm iterates through pairs (i, j) such that $0 \le j \le m - \phi(m)$, and $ip \le \frac{\phi(n)}{2}$. Since there are only $\mathcal{O}(m \cdot \phi(m))$ such pairs, it follows that the big prime algorithm takes $\mathcal{O}(m \cdot \phi(m)) \in \mathcal{O}(m^2)$ arithmetic operations. We only require the first half of the coefficients of $\Phi_m(z)$, as $\frac{\phi(m)}{2} > \frac{\phi(m)(p-1)}{2p} = \frac{\phi(n)}{2p}$.

We store the value of $a_n(k)$ in $a(k \bmod m)$, and discard that value when computing $a_n(k+m)$. If $p > m - \phi(m)$, the degree of $\Psi_m(z)$, then Algorithm 4.1 does not consider every term of $\Phi_n(z)$ with degree less than $\phi(n)/2$. In particular, if there is no pair (i, j)

such that $0 \leq i \leq \phi(m)$, $0 \leq j \leq m - \phi(n)$, and $ip + j = k$, then the big prime algorithm will not consider the term of the degree k. It follows from (4.4) that for such $k \geq m$, $a_n(k) = a_n(k - m)$, and for such $k < m$, $a_n(k) = 0$. Thus we need not consider these terms to obtain the height $A(n)$.

It is easy to modify Algorithm 4.1 to generate all (or half) of the coefficients of $\Phi_n(z)$ (Algorithm 4.2), should we have reason to look at all the coefficients. For instance, we may want to look at all the coefficients if we wanted to compute the $S(n)$, the length of $\Phi_n(z)$, or an l-norm of $\Phi_n(z)$. In such case we would discard a cyclotomic coefficient upon looking at it. We may, however, want to compute and store $\Phi_n(z)$, provided we have the storage necessary. In such case the number of comparisons and arithmetic operations in $\mathbb{Z}$ increases from $\mathcal{O}(m \cdot \phi(m))$ to $\mathcal{O}(m \cdot \phi(m) + \phi(n))$. To compute the dense representation of $\Phi_n(z)$, Algorithm 4.2 requires space for $\mathcal{O}(\phi(n))$ integers; however, we could compute all the coefficients $a_n(k)$ using immediate storage for only m integers if we wrote coefficients to disk periodically every time we have computed another m terms.

Algorithm 4.2: The big prime algorithm for computing $\Phi_n(z)$

Input:

- $n = mp$, a squarefree odd integer with largest prime divisor p
- $b_0, b_1, \ldots, b_{\phi(m)/2}$, the first half of the coefficients of $\Phi_m(z) = \sum_{i=0}^{\phi(m)} b_i z^i$,
- $c_0, c_1, \ldots, c_{m-\phi(m)}$, the coefficients of $\Psi_m(z) = \sum_{i=0}^{m-\phi(m)} c_i z^i$

Output: $a_n(0), \ldots, a_n(\phi(n)/2)$, the first half of the coefficients of $\Phi_n(z)$

```
1 a_n(0), ..., a_n(φ(n)/2) ⟵ 0, 0, ..., 0
2 for i ⟵ 0 to ⌊φ(n)/2p⌋ do
3     k ⟵ ip
4     for j ⟵ 0 to m − φ(m) do
5         if k ≥ m then a_n(k) ⟵ a_n(k − m) − b_i c_j else a_n(k) ⟵ −b_i c_j
6         k ⟵ k + 1
  return a_n(0), ..., a_n(φ(n)/2)
```

4.1.2 A big prime algorithm for $\Psi_n(z)$

Provided we have $\Phi_m(z)$ and $\Psi_m(z)$, we can generate the terms of $\Psi_n(z)$ for $n = mp$, in $\mathcal{O}(m \cdot \phi(m))$ arithmetic operations in $\mathbb{Z}$. To do this we calculate $\Psi_n(z)$ as a product of

two polynomials. One can show that

$$\Psi_{mp}(z) = \Phi_m(z)\Psi_m(z^p) \tag{4.5}$$

by showing that both sides of (4.5) have the same roots. Thus if we again let $\Phi_m(z) = \sum_{i=0}^{\phi(m)} b_i z^i$ and $\Psi_m(z) = \sum_{j=0}^{m-\phi(m)} c_j z^j$, it is immediate that

$$\Psi_n(z) = \sum_{i+pj=k} b_i c_j z^k, \tag{4.6}$$

where the sum is taken over $0 \le i \le \phi(m)$ and $0 \le j \le m - \phi(m)$. If $p > \phi(m)$, the implementation is especially simple, as we have at most one solution to $i + jp = k$ for a given value k. We see, for $p \gg \phi(m)$, that Ψ_{mp} is sparse.

4.1.3 Implementation details and observations

As was the case with our motivating example $n = 2576062979535$, the big prime algorithm and its variants were developed to calculate $A(n)$ for large squarefree n with a large prime divisor. The n for which we first implemented this algorithm were of the form $n = p_1p_2p_3p_4p_5$, a product of five distinct primes, such that $p_k > \prod_{i=0}^{k-1} p_i$ for $k = 2, 3, 4, 5$.

For such cases, to calculate $A(n)$ it is often advantageous to use a sparse representation for $\Phi_m(z)$ and $\Psi_m(z)$. For example, for $n = mp$ where $n = 2576062979535$, $m = 1134915$ and $p = 2269829$, $\Phi_m(z)$ has degree 584192 but only 31679 terms, and $\Psi_m(z)$ has degree 550723 but only 2982 terms.

For yet larger examples of n, we cannot fit an array of m integers in main memory. For example, we wanted to see if $\Phi_n(z)$ had height 1 for

$$n = 2876941641794034669918155 = 5 \cdot 29 \cdot 2029 \cdot 2353639 \cdot 4154714171969.$$

This is the smallest $n = p_1p_2p_3p_4p_5$ with $p_1 = 5$ that satisfies the set of congruences in (4.1). To test if $A(n) = 1$ we first used the sparse power series algorithm to calculate $\Phi_{p_1p_2p_3}(z)$ and $\Psi_{p_1p_2p_3}(z)$, we then used the big prime algorithm algorithm to generate sparse representations of $\Psi_m(z)$ and $\Phi_m(z)$, where $m = p_1p_2p_3p_4$. We then calculated the terms of $\Phi_n(z)$ whose degress were in a range modulo m sufficiently small that we could fit in memory. We found that $\Phi_n(z)$ is not flat, as $|a_n(k)| = 2$ for

$$k = 266298073621 \cdot 4154714171969 + 109596 = 184398730073579852543491,$$

at which point we stopped the calculation.

4.2 A challenge problem

T.D. Noe [28] asked us to compute $A(n)$, for

$$n = 99660932085 = 3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 43 \cdot 53. \tag{4.7}$$

Noe wanted to know whether $A(n) > n^8$. The product of the least six prime divisors of n, 1181895, is the least integer s satisfying $A(s) > s$; the product of the least seven prime divisors, 43730115, the least integer satisfying $A(s) > s^2$. $n/43$ and $n/53$ are the first two integers satisfying $A(s) > s^4$.

If one knew that the largest coefficient of $\Phi_n(z) = \sum_{i=0}^{\phi(n)} a_n(i)z^i$ occurs at degree d, one could compute $\Phi_m(z) = \sum_{i=0}^{\phi(m)} a_m(i)z^i$ and $\Psi_m(z) = \sum_{i=0}^{m-\phi(m)} c_m(i)z^i$ where $m = n/53 = 1880394945$ and then, given

$$\Phi_n(z) = \Phi_m(z^{53})\Psi_m(z)(1 + z^m + z^{2m} + \dots), \tag{4.8}$$

we have that

$$a_n(d) = \sum_{\substack{53i+j \equiv d \bmod m \\ 53i+j \le d}} a_m(i)c_m(j). \tag{4.9}$$

Given that we can take $d \le \phi(n)/2$, we could compute $A(n)$ using $\mathcal{O}(\phi(n))$ arithmetic operations. More generally, if we know that the degree of the term with the largest coefficient of $\Phi_n(z)$ occurs in some range of length L, then we can compute $A(n)$ in $\phi(n)L$ arithmetic operations. We expected the largest coefficient of $\Phi_n(z)$ to occur at a term of degree close to $\phi(n)/2$. The largest coefficients of $\Phi_{1181895}(z)$, $\Phi_{43730115}(z)$, and $\Phi_m(z)$ all occur near their respective middle terms (see Table 4.1). We were unable, however, to find a band of terms, definitively containing the largest coefficient of $\Phi_n(z)$, sufficiently small to feasibly compute $A(n)$ by this method.

Our first approach, prior to the development of SPS2-4, was to compute the first half of the terms of $\Phi_n(z)$ modulo 32-bit primes using the SPS algorithm, and then to use Chinese remaindering to obtain $\Phi_n(z)$ from its images. One difficulty is storage. One image requires 76 GB of space. Thus our array of coefficients could not fit in memory, and we had to write results of intermediate computation to disk. Knowing the height of $\Psi_m(z)$ and $\Phi_m(z)$ and considering (4.8), we knew that $A(n) < 2^{320}$, and that we may require as many as ten images of $\Phi_n(z)$ requiring some 760 GB of storage total. As $\Phi_n(z)$ has order

Table 4.1: Where the maximal coefficient of $\Phi_n(z) = \sum a_n(i)z^i$ occurs

n	$\phi(n)/2$	$i \le \phi(n)/2$ for which $\lvert a_n(i)\rvert = A(n)$
1,181,895	241,920	222,192
43,730,115	8,709,120	8,709,112
1,880,394,945	365,783,040	365,783,040

9, the SPS algorithm takes some $2^9 = 512$ passes through our coefficient array to compute one image. The hard disk bodes a bottleneck in the computation; each image took roughly 2 weeks to compute. During the computation of the fifth image the hard disk crashed and we abandoned this approach.

4.2.1 A new approach

In our second attempt, we sought to minimize our use of the hard disk. We used a variant of the SPS2 algorithm. In addition, the problem was broken up in a manner that allowed us to distribute the computation over multiple desktop computers. We used three computers, two with 6 GB of RAM and one with 4 GB of RAM.

Half of the proper natural divisors of n are divisible by $p = 53$. If we compute the truncated product of all the subterms $(1 - z^d)^{\pm 1}$ of $\Phi_n(z)$ for which $53 \nmid d$, then for the remainder of the computation of $\Phi_n(z)$ we can partition our array of coefficients into 53 sections, grouping terms by their degree modulo 53. We first compute

$$\Psi_m(z) \bmod z^{(m-\phi(m))/2+1} = \sum_{k=0}^{(m-\phi(m))/2} c_m(k) z^k \tag{4.10}$$

using the recursive SPS algorithm. After which we compute the coefficients of $\Psi_m(z)(1 + z^m + z^{2m} + \dots) \bmod z^{\phi(n)/2+1}$ in 53 sections. For $0 \le i < 53$, we computed

$$F_i(z) = \sum_{0 \le 53j+i \le \phi(n)/2} c_m(j \bmod m), \tag{4.11}$$

where $j \bmod m$ above is taken in the range $0 \le j < m$ and $c_m(k)$ is set to zero for $k > m - \phi(m)$. The terms $c_m(k)$, for $k > (m - \phi(m))/2$, half the degree of $\Psi_m(z)$, were obtained from the lower-degree terms of $\Psi_m(z)$ using the antipalindromic property of

$\Psi_m(z)$. The coefficients of the F_i are merely those of $\Psi_m(z)(1-z^m)^{-1}$ rearranged, as

$$\sum_{i=0}^{52} z^i F_i(z^{53}) \equiv \Psi_m(z)(1+z^m+z^{2m}+\dots) \pmod{z^{\phi(n)/2+1}} \tag{4.12}$$

Thus, by (4.8),

$$\sum_{i=0}^{52} z^i F_i(z^{53})\Phi_m(z^{53}) \equiv \Phi_n(z) \pmod{z^{\phi(n)/2+1}}. \tag{4.13}$$

Thus, to compute the coefficients of $\Phi_n(z)$, it suffices to compute $F_i(z)\Phi_m(z)$ for $0 \le i < 53$. We multiply each of the F_i by the subterms of $\Phi_m(z)$ by way of the SPS algorithm, here truncating to degree $\lfloor \frac{\phi(n)}{2\cdot 53} \rfloor$. We computed images of $\Psi_m(z)$, $F_i(z)$, and $F_i(z)\Phi_m(z)$ modulo 32 or 64-bit primes and wrote them to disk. Chinese remaindering was only performed on the coefficients of $F_i(z)\Phi_m(z)$ at the end of the computation. The two computers with 6 GB of memory computed five images modulo 64-bit primes, and the other computer with 4 GB of memory computed 10 images modulo 32-bit primes.

The lower half of the terms of $\Psi_m(z)$ required 2.3 (4.6) GB of memory taken modulo a 32 (64-bit prime). One of the 53 sections of the lower-degree terms of $\Phi_n(z)$ required roughly 1.4 GB (2.8 GB) of memory taken modulo a 32 (64-bit) prime. To construct the $F_i(z)$, we would load $\Psi_m(z)$ into memory and compute the terms of $F_i(z)$ in order of increasing degree, periodically writing the terms of $F_i(z)$ to disk so as not to exhaust main memory. Once sufficiently many images of $F_i(z)$ were computed, we then computed the images of $F_i(z)\Phi_m(z)$, one image at a time. These images were again written to disk. We could not store all the images of $F_i(z)\Phi_m(z)$ entirely in memory, so we reconstructed the coefficients of $F_i(z)\Phi_m(z)$ in suitably small segments.

In order to distribute the computation over three computers, each computer we used computed images of $\Psi_m(z)$. Computing the images of $\Psi_m(z)$, however, comprises a very small part of the computation. With the recursive SPS algorithm the images of $\Psi_m(z)$ were computed in under an hour. The entirety of the computation took roughly two days on three computers.

We found that

$$A(99660932085) = 61267208717407836670896202324395260\backslash$$
$$124725254733381530786789617551493787739155364471853 70,$$

which is roughly $2^{291.6}$ or $n^{7.98}$. We used this approach again to compute $A(n)$ for

$$n = 100280245065 = 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31,$$

the product of the least 10 odd primes. Monagan computed $A(n)$ for $n = 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29$ using the FFT. He did not attempt to compute $A(100280245065)$ because of the high memory cost required by a FFT-based approach. $\Phi_{100280245065}(z)$ is a polynomial of degree exceeding 30 billion. We computed

$$A(100280245065) = 380127943284004471680549556026957 6. \tag{4.14}$$

4.3 A reduced-memory recursive SPS algorithm

The method we describe in the previous section is, in effect, a distributed version of the SPS2 algorithm. As in the SPS2 algorithm, we compute $\Phi_{mp}(z)$ as

$$\Phi_{mp}(z) = -\Psi_m(z) \cdot \frac{1}{1 - z^m} \cdot \Phi_m(z^p). \tag{4.15}$$

As in SPS2, when multiplying by the subterms of $\Phi_n(z)$ appearing in $\Psi_m(z)$, we truncated to half the degree of $\Psi_m(z)$. The key difference between our new approach and SPS2 is that we observed that when multiplying by the subterms of $\Phi_m(z^p)$, that we can break our intermediate truncated power series into p smaller truncated power series of nearly equal size. We break the terms of our polynomial into p sets and run our computation on each of these p sets separately.

For large cyclotomic polynomials which cannot be stored in main memory using a dense representation, it would be ideal to have an algorithm that minimizes disk I/O as in the case of the approach of the previous section, while at the same time requiring (nearly) the same number of coefficient operations as the recursive sparse power series algorithm. We describe such an approach.

Procedure SPS4b(m, e, p, λ, D_{IN}, D, k, l, a, b): a distributed version of SPS4

Input: m, e, D_{IN}, and D, nonnegative integers; p, a prime; λ, a boolean value; $k < l$, integers for which $0 < k \le l < p$ and $l \equiv D_{IN} - k \bmod p$; $a = [a(0), a(1), \dots]$, and $b = [b(0), b(1), \dots]$, two arrays of integers

Result: Let $H(z) = \sum_{i=0}^{p-1} z^i F_i(z^p)$ be a product of cyclotomic polynomials of odd index and let $D_{IN} = \deg(H)$. Let $H^*(z) = H(z)\Phi_n(z^{ep})$ if λ is true. Otherwise, let $H^*(z) = H(z)\Psi_n(z^{ep})$. Let $F_i(z)$ and $F_i^*(z)$, where $0 \le i < p$, be the polynomials for which $H(z) = \sum_{i=0}^{p-1} z^i F_i(z^p)$ and $H^*(z) = \sum_{i=0}^{p-1} z^i F_i^*(z^p)$. The first half of the coefficients of $F_k(z)$ and $F_l(z)$ are stored to arrays a and b on input and that of $F_k^*(z)$ and $F_l^*(z)$ are written to a and b on output.

if λ **then** $D_{OUT} \longleftarrow D_{OUT} + \phi(m)ep$ **else** $D_{OUT} \longleftarrow D_{OUT} + (m - \phi(m))ep$
$D^* \longleftarrow \min(D_{OUT}, D)$, $e^* \longleftarrow e$, $m^* \leftarrow m$
while $m^* > 1$ **do**
    $q \longleftarrow$ (largest prime divisor of m^*), $m^* \longleftarrow m^*/q$
    if $m^* > 1$ or λ is false **then**
        $D_{IN} \longleftarrow$ SPS4b(m^*, e^*, p, not λ, D_{IN}, D^*, k, l, a, b)
    $e^* \longleftarrow e^*q$
if λ is false **then return** D_{OUT} `// We have multiplied by` $\Psi_m(z^e)$
`// Get higher degree terms as needed`
$D^* \longleftarrow \lfloor \frac{D^*}{2p} \rfloor$, $i = \lfloor \frac{D_{IN}}{2p} \rfloor$, $j \leftarrow \lfloor \frac{D_{IN}-k}{p} \rfloor - i$
while $i \ge 0$ *and* $j \le D^*$ **do**
    $a(j) \longleftarrow (-1)^{D_{IN}} a(i)$, **if** $k \ne l$ **then** $a(j) \longleftarrow (-1)^{D_{IN}} a(i)$
    $i \longleftarrow i - 1$, $j \longleftarrow j + 1$
for each prime $q|m$ **do**
    for $i = (me/q)$ **to** D^* **do**
        $a(i) \longleftarrow a(i) + a(i - me/q)$, **if** $k \ne l$ **then** $b(i) \longleftarrow b(i) + b(i - me/q)$
for $i = D^*$ **down to** d **do**
    $a(i) \longleftarrow a(i) - a(i - me)$, **if** $k \ne l$ **then** $b(i) \longleftarrow b(i) - b(i - me)$
return D_{OUT}

We compute $\Phi_n(z)$ as the product

$$\Psi_m(z) \cdot \Phi_m(z^p) \cdot (z^m - 1)^{-1}, \tag{4.16}$$

in order from the leftmost term right. We first compute the lower half of the terms $\Psi_m(z)$ by way of the recursive SPS algorithm. Once we have the truncated power series of $\Psi_m(z)$, we break up the power series into p power series $F_0(z), F_1(z), \ldots, F_{p-1}(z)$ where, given $\Psi_m(z) = \sum c_m(i)z^i$, the $F_i(z)$ satisfy

$$F_i = \sum c_m(i + pj)z^j, \tag{4.17}$$

and thus

$$\Psi_m(z) = \sum_{j=0}^{p-1} z^j F_i(z^p). \tag{4.18}$$

To compute the coefficients of

$$\Psi_m(z) \cdot \Phi_m(z^p), \tag{4.19}$$

we can compute the product $F_i^*(z) = F_i(z)\Phi_m(z)$ for $0 \leq i < p$, to give us the coefficients of $\Psi_m(z)\Phi_m(z^p)$, much like what we did to compute $A(99660932085)$; however, in this fashion we can truncate the degrees of the F_i in a fashion similar to that seen in the recursive SPS algorithm. Let $0 \leq l_1 \leq l < p$ be such that $l \equiv m - \phi(m) - l_1 \pmod{p}$. Suppose for now that $l_1 \neq l$. We can show that the coefficients of $G(z) = z^{l_1}F_k(z^p) + z^l F_l(z^p)$ are antipalindromic. Note that the terms of $G(z)$ are merely a subset of the terms of $\Psi_m(z)$. A term of $z^{l_1}F_{l_1}(z^p)$ is of the form $c_m(k)z^k$, where $k = l_1 + pj$ for some j and $0 \leq k \leq m - \phi(m)$. As $m - \phi(m) - k \equiv l \pmod{p}$, we have that $c_m(m - \phi(m) - k)z^{m-\phi(m)-k}$ is a term of $z^l F_l(z^p)$ and, moreover, by the antipalindromic property of the coefficients of $\Psi_m(z)$, $c_m(k) = -c_m(m - \phi(m) - k)$.

Thus, if we write $F_k(z) = \sum f_k(j)z^j$ and $F_l = \sum f_l(j)z^j$, then

$$\begin{aligned} f_k(i) &= c_m(jp + l_1), \\ &= -c_m(m - \phi(m) - (jp + l_1)), \\ &= -c_m(m - \phi(m) - l_1 - jp)), \\ &= -c_m(\lfloor \tfrac{m-\phi(m)-l_1}{p} \rfloor p + l - jp), \\ &= -f_l(\lfloor \tfrac{m-\phi(m)-l_1}{p} \rfloor - j). \end{aligned} \tag{4.20}$$

Similarly, $f_l(j) = -f_k(\lfloor\frac{m-\phi(m)-l}{p}\rfloor - j)$.. As $k < l$, $m - \phi(m) - l < m - \phi(m) - k$. If there exists some k such that

$$m - \phi(m) - l < kp \leq m - \phi(m) - k,$$

then, as $m - \phi(m) - l \equiv k \pmod p$ and $m - \phi(m) - k \equiv l \pmod p$, we have

$$m - \phi(m) - l = (k-1)p + k \qquad \text{and} \qquad m - \phi(m) - k = kp + l,$$

and $l - k = (m - \phi(m) - k) - (m - \phi(m) - l) = p + (l - k)$, an obvious contradiction. Thus

$$\lfloor\tfrac{m-\phi(m)-k}{p}\rfloor = \lfloor\tfrac{m-\phi(m)-l}{p}\rfloor. \tag{4.21}$$

More generally, if $H(z) = \sum_{j=0}^{D} h(j)z^j$ is a product of cyclotomic polynomials of odd index where $\deg(h) = D \equiv m - \phi(m) \pmod p$, then, letting

$$F_i = \sum f_i(j)z^j = \sum h(i + jp)z^j, \tag{4.22}$$

we have

$$f_k(j) = (-1)^D - f_l(\lfloor\tfrac{m-\phi(m)-k}{p}\rfloor - j) \quad \text{and} \quad f_l(j) = (-1)^D - f_k(\lfloor\tfrac{m-\phi(m)-k}{p}\rfloor - j). \tag{4.23}$$

Similarly, if $k \equiv m - \phi(m) - k \pmod p$, then

$$f_k(j) = (-1)^D - f_k(\lfloor\tfrac{m-\phi(m)-k}{p}\rfloor - j). \tag{4.24}$$

Using this property we can, as we have with every SPS algorithm, obtain the higher-degree terms of $H(z)$ from its lower degree terms. To ensure we have the lower-half of the terms of $H(z)$, we compute the terms of each F_j up to degree $\lfloor\frac{D}{2p}\rfloor$.

Procedure SPS4b details this variant of the recursive SPS algorithm, which is a distributed analog to the recursive SPS4. The procedure handles the cases for which $k \neq l$ and $k = l$. We include source code in the appendix of an implementation of SPS4b. To compute the coefficients of $\Psi_m(z)\Phi_m(z^p)$ we would call $SPS4b(m, 1, p, m-\phi(m), \phi(n), k, l, F_k, F_l)$ for every pair $0 \leq k \leq l < p$ such that $l \equiv m - \phi(m) - k \pmod p$.

We note that this method is easily generalized to compute the coefficients of $\Psi_n(z) = \Phi_m(z)\Psi_m(z^p)$; moreover, in the case of $\Psi_n(z)$, we need not perform the additional step of multiplying by $(z^m - 1)^{-1}$.

Once we have polynomials $F_i^*(z) = \sum f_i^*(j)z^j$, for $0 \le i < p$, satisfying

$$\sum_{i=0}^{p} z^i F_i^*(z^p) = \Psi_m(z)\Phi_m(z^p), \tag{4.25}$$

it remains to multiply $\Psi_m(z)\Phi_m(z^p)$ by $(z^m-1)^{-1} = (-1 - z^m - z^{2m} - \dots)$ to obtain the coefficients of $\Phi_n(z)$. We can do this with an array of m integers. Note that the coefficient of the term of degree k of $\Psi_m(z)\Phi_m(z^p)$ is $f^*_{k \bmod p}(\lfloor k/p \rfloor)$. Thus for the coefficient of the term of degree k of $\Phi_n(z)$, $a_n(k)$, it holds that

$$\begin{aligned} a_n(k) &= - \sum_{\substack{0 \le j \le k \\ j \equiv k \pmod m}} f^*_{j \bmod p}\lfloor j/p \rfloor, \\ &= a_n(k-m) - f^*_{k \bmod p}\lfloor k/p \rfloor. \end{aligned} \tag{4.26}$$

Thus, we can obtain the terms of $\Phi_n(z)$ is a manner very similar to the big prime algorithm. We initialize an array of m integers to zero, then we iterative through the terms of $\Psi_m(z)\Phi_m(z^p)$ in order of increasing degree.

Algorithm 4.3 shows how we organize in memory the problem of computing $A(n)$ in the manner we described using an array of size $m+sp$, for some integer $s > 0$. As in chapter 2, given an array $a = [a(0), a(1), \dots]$ and integer $t > 0$, we let $a+t$ denote the array whose first integer is $a(t)$. That is, $(a+t) = [a(t), a(t+1), \dots]$ and $(a+t)(i) = a(t+i)$. This method has allowed us to compute $A(n)$ for $n > 10^{10}$. We used this algorithm to compute the values of $A(n)$ appearing in tables A.4 and A.5, for instance. We have implemented versions of Algorithm 4.3 that compute $\Phi_n(z)$ using 64, 128, and 192-bit integers. We also have a version which computes $\Phi_n(z) = \sum a_n(i)z^i$ modulo 64-bit primes, for this version we need to write the coefficients of $\Phi_n(z) \bmod q$ to disk as we compute them such that we can perform Chinese remaindering on the images of $\Phi_n(z)$. Algorithm 4.3 requires immediate storage (i.e. storage in main memory, as opposed to disk) for $m + sp$ integers, for some integer $s > 0$. In the for-loop beginning on line 6 of the algorithm, we require $\lfloor \frac{m-\phi(m)}{2p} \rfloor p$ immediate storage for coefficients of $\Psi_m(z)$ and an additional $\lfloor \frac{\phi(n)}{p} \rfloor + 2$ integers to store the coefficients of $F_k(z)$ and $F_l(z)$. However, as $\phi(n)/p < \phi(m)$, we find that

$$\lfloor \tfrac{m-\phi(m)}{2p} \rfloor p + \lfloor \tfrac{\phi(n)}{p} \rfloor + 2 \le (\tfrac{m-\phi(m)}{2} + p - 1) + \phi(m) + 1 < m + p,$$

which fits within our memory constraints.

It should be noted that the computation of the $F_i^*(z)$ satisfying (4.25) could be performed in parallel. That is, the for-loop of Algorithm 4.3 beginning on line 6 could be done in parallel. Running the algorithm in parallel would require additional immediate storage as we would need to store many of the $F_i^*(z)$ in main memory at once. Since the development of the algorithms of chapter 3, however, we find the computation of cyclotomic polynomials is a problem limited by space rather than by time; any cyclotomic polynomial that, using a dense representation, can be stored in main memory is already easy to compute. Moreover, this method is not easily distributed over multiple computers, as we did to compute $A(99660932085)$, as described in the previous section. This is because the last phase of Algorithm 4.3, multiplying by $(z^m - 1)^{-1}$, is sequential. We would require that a computer have access to all the polynomials $F_i^*(z)$.

Algorithm 4.4 shows the analogous algorithm for $\Psi_n(z)$. This algorithm again requires storage for $m+p$ integers: $\lfloor \frac{\phi(m)}{2p} \rfloor p$ integers for the coefficients of $\Phi_m(z)$, and $\lfloor \frac{n-\phi(n)}{p} \rfloor + 2$ integers to store the coefficients of $F_k(z)$ and $F_l(z)$, where here, the $F_i(z)$ satisfy

$$\sum_{i=0}^{p-1} z^i F_i(z^p) = \Phi_m(z). \tag{4.27}$$

Thus our memory requirement for this algorithm is

$$\begin{aligned} &\lfloor \tfrac{\phi(m)}{2p} \rfloor p + \lfloor \tfrac{n-\phi(n)}{p} \rfloor + 2 \leq (\phi(m)/2 + p - 1) + (m - \phi(n)/p + 1) \\ &= m + p + \phi(m)(\tfrac{1}{2} - p - 1p), \end{aligned} \tag{4.28}$$

which is at most $m + p - \phi(m)/6$, given $p \leq 3$. This algorithm, unlike its counterpart for $\Phi_n(z)$, does not require us to write to disk, as we do not have to multiply by another subterm $(z^m - 1)^{-1}$ at the end of the computation. In Algorithm 4.4 we merely compute the $F_i^*(z)$ satisfying

$$\sum z^i F_i^*(z^p) = \Phi_m(z)\Psi_m(z^p) = \Psi_{mp}(z). \tag{4.29}$$

We compute the $F_i^*(z)$ in pairs $F_k^*(z), F_l^*(z)$ where $l = (\phi(m) - k) \bmod p$. Then, as the coefficients of $F_k^*(z)$ and $F_l^*(z)$ are merely a subset of those of $\Psi_m(z)$, we look through the coefficients of $F_k^*(z)$ and $F_l^*(z)$ and then discard them, whereas in Algorithm 4.3 we would write the intermediate polynomials to disk. We used Algorithm 4.4 to compute the heights of $\Psi_n(z)$ of order 9, for squarefree odd n, $5 \cdot 10^9 < n < 18 \cdot 10^9$.

Algorithm 4.3: A memory-friendly SPS4-based algorithm for computing $A(n)$

Input:

- $n = mp$, a squarefree odd integer with largest prime divisor p
- $a = [a(0), a(1), \dots]$, an array of $m + sp$ integers, where $s > 0$, initialized to zero.

Output: $A(n)$, the height of $\Phi_n(z) = \sum a_n(i)z^i$

$a(0) \longleftarrow 1, \quad H \longleftarrow 0$

SPS4($m, 1$, false, $0, m - \phi(m), a$) // Compute $\Psi_m(z)$

$D \longleftarrow \lfloor \frac{m-\phi(m)}{2p} \rfloor, \quad D^* \longleftarrow \lfloor \frac{\phi(n)}{2p} \rfloor$

for $i \longleftarrow \frac{m-\phi(m)}{2}$ **to** $(D+1)p - 1$ **do** $a(i) \longleftarrow -a(m - \phi(m) - i)$

$b \longleftarrow a + (D+1)p, \quad c \longleftarrow b + (D^* + 1)$

for $k \longleftarrow 0$ **to** $p - 1$ **do**

  $l \longleftarrow m - \phi(m) - k \bmod p$

  if $k \le l$ **then**

    // Write $F_k(z), F_l(z)$ to arrays b, c

    for $i \longleftarrow 0$ **to** D **do** $b(i) \longleftarrow a(k + ip), \quad c(i) \longleftarrow a(l + ip)$

    // Compute $F_k^*(z)$ and $F_l^*(z)$

    SPS4b($m, 1, p$, true , $m - \phi(m), \phi(n), k, l, b, c$)

    Write the first D^* coefficients of $F_k^*(z) = \sum f_k^*(i)z^i$ and $F_l^*(z) = \sum f_l^*(i)z^i$, stored in arrays b and c respectively, to disk.

// Multiply by $(z^m - 1)^{-1}$

for $i \longleftarrow 0$ **to** $m - 1$ **do** $a(i) \longleftarrow 0$

$b \longleftarrow a + m$

for $i \longleftarrow 0$ **to** $\phi(n)/2$ **do**

  if $(sp)|i$ **then**

    Write $f_k^*(j)$, for $0 \le k < p$ and $i/p \le j < i/p + s$, from disk to array b.

  $a(i \bmod m) \longleftarrow a(i) - f^*_{i \bmod p}(\lfloor i/p \rfloor)$ // $a(i \bmod m)$ now stores $a_n(i)$

  if $|a(i \bmod m)| > H$ **then** $H \longleftarrow a(i \bmod m)$

return H

Algorithm 4.4: A memory-friendly SPS4-based algorithm for $\Psi_n(z)$

Input:

- $n = mp$, a squarefree odd integer with largest prime divisor p
- $a = [a(0), a(1), \dots]$, an array of at least $m + p - \phi(m)(\frac{1}{2} - \frac{p-1}{p})$ integers, initialized to zero.

Output: $\bar{A}(n)$, the height of $\Psi_n(z) = \sum c_n(i) z^i$

$a(0) \longleftarrow 1, \quad H \longleftarrow 0$

SPS4($m, 1, false$, true, $m - \phi(m), a$) // Compute $\Psi_m(z)$

$D \longleftarrow \lfloor \frac{\phi(m)}{2p} \rfloor, \quad D^* \longleftarrow \lfloor \frac{n-\phi(n)}{2p} \rfloor$

for $i \longleftarrow \frac{m-\phi(m)}{2}$ **to** $(D+1)p - 1$ **do** $a(i) \longleftarrow -a(m - \phi(m) - i)$

$b \longleftarrow a + (D+1)p, \quad c \longleftarrow b + (D^* + 1)$

for $k \longleftarrow 0$ **to** $p - 1$ **do**

  $l \longleftarrow \phi(m) - k \bmod p$

  if $k \leq l$ **then**

    // Write $F_k(z), F_l(z)$ to arrays b, c

    for $i \longleftarrow 0$ **to** D **do** $b(i) \longleftarrow a(k + ip), \quad c(i) \longleftarrow a(l + ip)$

    // Compute $F_k^*(z)$ and $F_l^*(z)$

    SPS4b($m, 1, p$, false , $m - \phi(m), \phi(n), k, l, b, c$)

    for $i \longleftarrow 0$ **to** D^* **do**

      if $|b(i)| > H$ **then** $H \longleftarrow |b(i)|$

      if $k \neq l$ *and* $|c(i)| > H$ **then** $H \longleftarrow |c(i)|$;

return H

Chapter 5

Timings and results

All of our cyclotomic polynomial data is available online at

`http://www.cecm.sfu.ca/~ada26/cyclotomic/`

5.1 Timings

The timings from this section were taken using a 2.67 GHz Intel Core i7 computer with 6 GB of RAM. All times are rounded to one hundredth of a second. The SPS algorithms and its variants all use 64-bit integers for the coefficients. Unless otherwise specified, we did timings without using an overflow check. All of our implementations are written in C and inline assembly.

Table 5.1 gives a comparison of our implementations of the different algorithms of chapter 2. We measure the time for each method to compute $\Phi_n(z)$ for different values of n. The test cases n were taken from Table A.1, which list $\Phi_n(z)$ of particularly large height. For our timings of each version of the cyclotomic Fourier transform (CFT), we measured the time it took to compute the discrete Fourier transform of $\Phi_n(z)$ by way of procedure CFT (or CFT2-4), plus the time to interpolate $\Phi_n(z)$ from its Fourier transform by way of the inverse FFT (procedure FFT2). In each timed implementation, we only computed one image $\Phi_n(z) \bmod q$, for q a 64-bit prime. As we explain in Section 2.4.1, our implementations of CFT3 and CFT4 we read $DFT(M, \omega, 1/\Phi_1(z))$, where $M = 2^{25}$, from disk, to avoid having to perform division in $\mathbb{Z}_q$. For smaller values of n in Table 5.1,

the time to load 2^{25} integers from disk outweighs the cost of divisions. We see, however, for larger n, that CFT3 is the best of the listed methods.

Table 5.2 compares the SPS methods to CFT3, our fastest method of chapter 2; Maple 11's `cyclotomic` command (whose timings are under the column entitled Maple11), which used classical polynomial arithmetic to compute $\Phi_n(z)$; and the big prime algorithm for computing $A(n)$ (Algorithm 4.1). For the big prime algorithm (column Bigprime64 in Table 5.2), we first compute $\Phi_m(z)$ and $\Psi_m(z)$, where $n = mp$ for some prime p, using the SPS algorithm, and then compute $A(n)$ by way of the big prime algorithm. In this implementation of the big prime algorithm we use a dense representation of Φ_m and Ψ_m. We use a dense representation for these timings because the $\Phi_n(z)$ used for these timings were all dense. We see that the big prime algorithm slows appreciably as n increases. This is because the algorithm is effectively quadratic time. It should be noted, however, that the algorithm was not intended for these cases. In every timed implementation in Table 5.2, we use 64-bit machine integers to store the coefficients of $\Phi_n(z)$.

Tables 5.3 and 5.4 compare the different versions of the sparse power series method to compute $\Phi_n(z)$ and $\Psi_n(z)$ respectively. We tested our code on the least squarefree n with k prime factors greater than 2^j for $3 \leq k \leq 8$ and $27 \leq j \leq 30$. We also included a few examples of $\Phi_n(z)$ of order 9. For all the cases chosen here, $A(n) < 2^{63}$. We also include, in Table 5.3 timings for an implementation of the SPS algorithm that checks if integer overflow occurs in the coefficient arithmetic. This overflow check was implemented using inline assembly.

Table 5.5 gives timings of the big prime algorithm (Algorithm 4.1) on its intended inputs: $\Phi_n(z)$ for which $n = mp$ has a large prime divisor p. All the n in Table 5.5 were candidates for a flat $\Phi_n(z)$ of order 5. We used the SPS algorithm to compute dense representations of $\Phi_m(z)$ and $\Psi_m(z)$ by way of the SPS algorithm, and then converted these polynomials to a sparse representation. As we knew these polynomials had small height, we stored the coeffcients as 8-bit integers.

Table 5.1: A comparison of Fourier-transform-related methods of computing $\Phi_n(z)$

	method				
n	alg. 2.1	CFT1	CFT2	CFT3	CFT4
171717	0.29	0.45	0.13	0.25	0.26
255255	0.31	0.52	0.13	0.24	0.25
279565	0.63	0.88	0.26	0.29	0.31
327845	0.63	0.88	0.29	0.29	0.35
707455	0.81	1.78	0.60	0.41	0.49
886445	1.70	3.60	1.18	0.69	0.97
983535	0.81	2.11	0.62	0.42	0.57
1181895	0.83	2.11	0.61	0.42	0.53
1752465	1.72	4.26	1.19	0.70	1.06
3949491	3.93	8.72	2.56	1.42	2.20
8070699	9.32	17.77	5.46	3.29	4.64
10163195	20.24	35.86	11.42	6.37	9.77
13441645	20.14	35.94	11.40	6.36	9.81
15069565	42.59	72.59	23.67	13.52	20.94
30489585	43.44	83.04	24.18	13.94	22.84
37495115	94.86	148.25	54.91	29.61	45.04
40324935	43.39	83.20	24.28	14.04	23.27
43730115	94.83	169.10	50.59	29.69	48.49

Table 5.2: A comparison of times to compute $\Phi_n(z)$

	method						
n	Maple11	CFT3	bigprime64	SPS	SPS2	SPS3	SPS4
171717	99.54	0.25	0.00	0.00	0.00	0.00	0.00
255255	315.17	0.24	0.08	0.00	0.00	0.00	0.00
279565	586.99	0.29	0.04	0.00	0.00	0.00	0.00
327845	1241.75	0.29	0.04	0.00	0.00	0.00	0.00
707455	-	0.41	0.11	0.00	0.00	0.00	0.00
886445	-	0.69	0.16	0.01	0.00	0.00	0.00
983535	-	0.42	0.44	0.01	0.00	0.00	0.00
1181895	-	0.42	0.66	0.01	0.00	0.00	0.00
1752465	-	0.70	0.67	0.02	0.01	0.00	0.00
3949491	-	1.42	4.72	0.06	0.03	0.01	0.01
8070699	-	3.29	12.33	0.16	0.08	0.02	0.02
10163195	-	6.37	43.17	0.26	0.13	0.04	0.03
13441645	-	6.36	43.58	0.35	0.18	0.05	0.05
15069565	-	13.52	49.73	0.40	0.21	0.06	0.05
30489585	-	13.94	390.77	1.11	0.58	0.16	0.13
37495115	-	29.61	201.61	1.07	0.57	0.15	0.14
40324935	-	14.04	393.81	1.48	0.76	0.19	0.16
43730115	-	29.69	579.30	1.72	0.95	0.23	0.19

Table 5.3: Time to calculate $\Phi_n(z)$ using different versions of the SPS algorithm

order(Φ_n)	n	SPS*	SPS	SPS2	SPS3	SPS4
3	134217737	1.08	0.89	0.62	0.45	0.47
	268435465	1.71	1.41	0.97	0.69	0.72
	536870913	2.78	2.71	1.56	0.95	1.07
	1073741829	5.56	4.72	3.25	1.92	2.20
4	134217733	2.13	1.58	0.87	0.55	0.61
	268435461	2.49	1.80	0.94	0.47	0.54
	536870915	5.97	4.48	2.39	1.42	1.48
	1073741837	16.59	11.09	6.63	3.87	4.32
5	134217755	3.89	2.11	1.17	0.44	0.46
	268435479	6.32	3.62	2.06	0.72	0.79
	536870973	14.29	7.66	4.39	1.48	1.67
	1073741835	22.53	16.07	7.96	2.32	2.69
6	134218455	5.30	2.53	1.50	0.30	0.33
	268435695	11.46	5.69	3.24	0.66	0.71
	536871237	24.86	13.65	7.01	1.54	1.71
	1073742117	48.16	25.09	13.58	3.18	3.37
7	134232945	9.96	4.65	2.35	0.54	0.50
	268453185	19.35	8.92	4.61	1.48	1.05
	536872245	41.67	16.33	10.53	1.89	1.94
	1073746605	96.13	45.80	25.72	4.23	4.27
8	140645505	16.58	8.10	4.39	1.60	1.13
	269023755	37.25	17.19	8.85	2.84	2.05
	536958345	79.90	35.40	18.90	6.00	3.79
	1074800265	160.89	74.00	38.97	10.19	6.65
9	3234846615	700.58	437.94	246.75	82.00	49.89
	4127218095	1042.55	566.40	316.39	96.69	59.29

* with overflow check

Table 5.4: Time to calculate $\Psi_n(z)$ using the SPS and recursive SPS algorithms

order(Φ_n)	n	SPS-Psi	SPS4
3	134217737	0.04	0.00
	268435465	0.30	0.00
	536870913	1.15	0.02
	1073741829	2.27	0.01
4	134217733	0.15	0.01
	268435461	1.54	0.13
	536870915	1.98	0.24
	1073741837	2.29	0.10
5	134217755	1.10	0.09
	268435479	2.72	0.09
	536870973	5.25	0.17
	1073741835	13.12	0.58
6	134218455	3.73	0.17
	268435695	7.42	0.35
	536871237	13.40	0.50
	1073742117	27.58	1.09
7	134232945	8.18	0.34
	268453185	17.32	0.82
	536872245	32.79	1.17
	1073746605	61.20	2.13
8	140645505	18.91	0.81
	269023755	35.96	1.40
	536958345	72.40	2.59
	1074800265	144.81	4.60

Table 5.5: Time to compute $A(n)$ using the big prime algorithm with 8-bit integers and sparse representations of $\Psi_m(z)$ and $\Phi_m(z)$

n	factorization of n	time to compute $A(n)$
746443728915	$3 \cdot 5 \cdot 31 \cdot 929 \cdot 1727939$	0.27
746444592885	$3 \cdot 5 \cdot 31 \cdot 929 \cdot 1727941$	0.29
1147113361785	$3 \cdot 5 \cdot 29 \cdot 1741 \cdot 1514671$	0.51
2576062979535	$3 \cdot 5 \cdot 29 \cdot 2609 \cdot 2269829$	0.77
7157926096635	$3 \cdot 5 \cdot 29 \cdot 4349 \cdot 3783629$	1.18
36654908721735	$3 \cdot 5 \cdot 29 \cdot 6959 \cdot 12108659$	1.90
44151142013985	$3 \cdot 5 \cdot 59 \cdot 5309 \cdot 9396929$	2.97
69589277763735	$3 \cdot 5 \cdot 29 \cdot 7829 \cdot 20433689$	2.13
98219673468435	$3 \cdot 5 \cdot 29 \cdot 6089 \cdot 37082009$	1.66
117714212390685	$3 \cdot 5 \cdot 59 \cdot 3539 \cdot 37584179$	1.97
313993273392885	$3 \cdot 5 \cdot 59 \cdot 7079 \cdot 50119319$	3.94
314037624291735	$3 \cdot 5 \cdot 59 \cdot 14159 \cdot 25061429$	7.88
457276728348885	$3 \cdot 5 \cdot 89 \cdot 8009 \cdot 42768059$	6.74
1349266102959585	$3 \cdot 5 \cdot 59 \cdot 8849 \cdot 172290029$	4.92
4990007947050435	$3 \cdot 5 \cdot 149 \cdot 22349 \cdot 99900029$	31.65
16628239064490285	$3 \cdot 5 \cdot 179 \cdot 10739 \cdot 576684299$	18.24
24901063029411285	$3 \cdot 5 \cdot 89 \cdot 37379 \cdot 499009649$	31.62
27949574611135785	$3 \cdot 5 \cdot 89 \cdot 26699 \cdot 784149629$	22.85

5.2 Heights of cyclotomic polynomials

We have amassed a library of data concerning the heights and lengths of cyclotomic and inverse cyclotomic polynomials. This library comprises $A(n)$ and $S(n)$ for squarefree, odd n belonging to the following, (with the number of such n in "[]" brackets):

- $n < 10^8$ with three prime factors [13,337,070].
- $n < 3 \cdot 10^8$ with four prime factors [18,561,168].
- $n < 6.6 \cdot 10^8$ with five prime factors [10,305,306].
- $n < 1.05 \cdot 10^9$ with six prime factors [2,056,713].
- $n < 2.65 \cdot 10^9$ with seven prime factors [346,584].
- $n < 6 \cdot 10^9$ with eight prime factors [19,086].
- $n < 1.8 \cdot 10^{10}$ with nine prime factors [341].
- n, a product of the least ten odd primes.

We have, in addition, computed the height and lengths of $\Psi_n(z)$ for odd, squarefree n amongst the following:

- $n < 10^8$ with three prime factors [13,337,070].
- $n < 10^8$ with four prime factors [5,744,524].
- $n < 5 \cdot 10^8$ with five prime factors [7,511,764].
- $n < 1.05^9$ with six prime factors [2,056,713].
- $n < 2.65 \cdot 10^9$ with seven prime factors [346,584].
- $n < 6 \cdot 10^9$ with eight prime factors [19,086].
- $n < 1.8 \cdot 10^{10}$ with nine prime factors [341].

This data is available online at:

`http://www.cecm.sfu.ca/~ada26/cyclotomic`.

5.2.1 Cyclotomic polynomials of very large height

Table A.1 shows the values $A(n)$ for which $A(n) > A(m)$ for all $1 \leq m < n$. This table is not exhaustive for $n > 5.4 \cdot 10^{10}$. That is, there may exist $n > 5.4 \cdot 10^{10}$ for which $A(n) > A(m)$ for all $m < n$. Monagan [2] used the FFT-based Algorithm 2.1 to compute the values $A(n)$ appearing in Table A.1 for $n \leq 43730115$ and for $n = 1880394945$ and $n = 2317696095$. I used the sparse power series algorithm to show that $A(1880394945)$ and $A(2317696095)$ were bigger than $A(m)$ for all lessers integers m. The SPS algorithm, moreover, was used to fill in the gap in the table for $43730115 < n < 1880394945$. Lastly, the disk-based methods of Section 4.3 were used to compute $A(n)$ for $n > 2317696095$ appearing in the table.

To show that $A(n)$ exceeds $A(m)$ for $m < n$, we needed bounds on $A(n)$. Bang [7] showed that for $n = pq$, a product of two primes, that $A(n) = 1$; and for $n = pqr$, a product of three primes, that $A(n) < p$. Bloom [9] later proved for $n = pqrs$, a product of four primes with $p < q < r < s$, that $A(n) < p(p-1)(pq-1)$. Bateman, Pomerance, and Vaughan [8] proved a generalized albeit slightly weaker result: for $n = p_1 p_2 \cdots p_k$, a product of k distinct primes with $p_1 < p_2 < \cdots < p_k$,

$$A(n) \leq A(p_1 p_2 \cdots p_{k-1}) \prod_{j=0}^{k-2} S(p_1 p_2 \cdots p_j) \tag{5.1}$$

Using $S(p_1 p_2 \cdots p_j) \leq A(p_1 p_2 \cdots p_j) \cdot p_1 p_2 \cdots p_j$ and Bang's results, they inductively obtain that

$$A(p_1 p_2 \cdots p_k) \leq \prod_{i=1}^{k-2} p_i^{2^{k-i-1}-1} \tag{5.2}$$

For example, $A(p_1 p_2 p_3 p_4 p_5 p_6) \leq p_1^{15} p_2^7 p_3^3 p_4^1$. We use this bound to narrow what orders of cyclotomic polynomials we need look at to search for values of $A(n)$ for which $A(n) > A(m)$ for all $m < n$. For instance, given $n = p_1 p_2 p_3 p_4 p_5$, a product of five primes where $p_1 < p_2 < p_3 < p_4 < p_5$, we have that $p_1 < n^{1/5}$, $p_1 p_2 < n^{2/5}$, and $p_1 p_2 p_3 < n^{3/5}$, which gives us the bound

$$A(n) \leq p_1^7 p_2^3 p_3 = (p_1)^5 \cdot (p_1 p_2)^2 \cdot (p_1 p_2 p_3) < (n^{1/5})^5 \cdot (n^{2/5})^2 \cdot n^{3/5} = n^{11/2}. \tag{5.3}$$

In general, for primes $2 < p_1 < p_2 < \cdots < p_k$,

$$A(p_1 p_2 \cdots p_k) \leq \prod_{i=1}^{k-2} n^{(2^{k-i-1}-1)/k} = n^{(2^{k-1})/k-1}. \tag{5.4}$$

This bound holds for all $k > 0$. This bound alone was not sufficient to allow us to produce Table A.1. For instance, by (5.4), for $\Phi_n(z)$ of order 7, $A(n) < n^{57/7}$. Thus, for instance, if n is an odd squarefree product of seven primes for which $n > A(99660932085)^{7/57} \approx 6.038 \cdot 10^{10}$, then $A(n)$ is potentially larger than $A(99660932085)$ if we strictly consider the bound (5.4). It is certainly intractible to compute all $\Phi_n(z)$ of order 7 for $6.038 \cdot 10^{10} < n < 99660932085 \approx 99.66 \cdot 10^{10}$. It is a nontrivial problem just to generate all the squarefree, odd products of seven primes n that appear in that range.

We can bound the heights of cyclotomic polynomials more accurately using that, given $n = mp$ for some prime $p \nmid m$, $\Phi_n(z) = \Phi_m(z^p)\Psi_m(z)/(z^m - 1)$. Certainly any term of $\Phi_m(z^p)$ multiplied by a term of $\Psi_m(z)$ will have a coefficient with absolute value at most $A(m)\bar{A}(m)$, where $\bar{A}(m)$ denotes the height of $\Psi_m(z)$. Counting how many such products of terms of $\Phi_m(z^p)$ and $\Psi_m(z)$ can have the same degree, we have that

$$|\Phi_m(z^p)\Psi_m(z)|_1 \leq A(m)\bar{A}(m) \cdot (\lfloor \tfrac{m-\phi(m)}{p} \rfloor + 1) \tag{5.5}$$

We can bound the coefficient of the term of degree l of $\Phi_n(z) = \Phi_m(z^p)\Psi_m(z)(1 + z^m + z^{2m} + \dots)$ by

$$|\Phi_m(z^p)\Psi_m(z)|_1(\lfloor l/m \rfloor + 1).$$

Since the largest coefficient of $\Phi_n(z)$ must occur at some term of degree $k \leq \phi(n)/2$, we have that

$$A(n) \leq A(m)\bar{A}(m) \cdot (\lfloor \tfrac{m-\phi(m)}{p} \rfloor + 1)(\lfloor \tfrac{\phi(n)}{2m} \rfloor + 1). \tag{5.6}$$

As $(\lfloor \frac{m-\phi(m)}{p} \rfloor + 1) < m$ and $(\lfloor \frac{\phi(n)}{2m} \rfloor + 1) < n/(2m) + 1 = p/2 + 1 < p$, we can relax (5.6) to

$$A(n) \leq A(m)\bar{A}(m)n \tag{5.7}$$

Similarly, given $\Psi_n(z) = \Psi_m(z^p)\Phi_m(z)$, we have

$$\bar{A}(n) \leq A(m)\bar{A}(m)(\lfloor \tfrac{\phi(m)}{p} \rfloor + 1) \tag{5.8}$$

Here our library of cyclotomic polynomial data becomes useful. Having computed (at the time of writing this), $A(m)$ and $\bar{A}(m)$ for all $m < 10^9$, a squarefree, odd product

of six primes, we find for such m, $A(m)\bar{A}(m) < 2^{96}$. Thus if $\Phi_n(z)$ is of order 7 and $n < 99660932085$, then if n has a prime divisor p for which $m = n/p < 10^9$, then $A(n) < 2^{96}n < 10^{133} < A(99660932085)$. In order to rule out the remaining $\Phi_n(z)$ of order 7, we use Maple to generate all n, a squarefree, odd products of 7 primes, for $n < 99660932085$. This can be done in a matter of minutes. There are 5,816 such n. For each of these n, we bound $A(n)$ using 5.7, for every $p|n$, applying the bound recursively on $m = n/p$ if necessary. For these remaining n, we computed that $A(n) < 2^{120}$, which is considerably less than $A(99660932085) > 2^{291}$. We similarly were able to show that $A(n)$, for $\Phi_n(z)$ of orders 8 and 9, for $n < 99660932085$, is less than $A(99660932085)$ as well.

Excluding n less than roughly 10000, those n for which we obtain the largest heights also typically yield the largest lengths. Table A.1 also gives $\log_n(A(n))$, which was of interest to us. Our results include the smallest n such that $A(n) > n$, $A(n) > n^2, A(n) > n^3$, and $A(n) > n^4$. Table A.2 (page 90) shows $A(n)$ for n, a product of the s smallest odd primes, for $1 \leq s \leq 9$. Table A.7 (page 94) shows $\Psi_n(z)$ of particularly large height. Many of the integers n appearing in Table A.1 appear in A.7 as well.

The heights of cyclotomic polynomials show a somewhat uneven distribution. For instance, we computed tens of examples of $\Phi_n(z)$ of order 8, $n < 5 \cdot 10^9$, for which $2^{133} < A(n) < 2^{136}$; however, there does not exist $\Phi_n(z)$ of order 8, $n < 5 \cdot 10^9$ for which $2^{103} < A(n) < 2^{133}$. Table A.3 lists the least n for which $2^{132} < A(n)$, and their prime factorizations. Note that each of the n listed share $3, 5, 11$, and 13 as prime divisors, and that many of the n share other prime divisors. Using Algorithm 4.3, we found similar families of $\Phi_n(z)$ of order 8 whose heights were greater than 2^{146} (Table A.4) and 2^{210} (Table A.5), as well as a family of $\Phi_n(z)$ of order 9 and height exceeding 2^{212} (Table A.6).

5.2.2 Flat cyclotomic polynomials

If p is a prime, then $\Phi_p(z) = 1 + z + \cdots + z^{p-1}$ is trivially flat. All cyclotomic polynomials of order 2 are also flat. This is easy to verify using the following identity (see Lenstra, [19]). Given primes p, q, let u, v be the integers such that $0 < u < p$, $0 < v < q$, and $uq + vp = pq + 1$. Then

$$\Phi_{pq}(z) = \sum_{i=0}^{u-1} \sum_{j=0}^{v-1} z^{iq+jp} - \sum_{i=u}^{p-1} \sum_{j=v}^{q-1} z^{iq-jp-pq}. \tag{5.9}$$

One can check that the degrees of the terms in each sum are distinct.

Cyclotomic polynomials of order three and greater are not, in general, flat.

Flat cyclotomic polynomials of order 3

There are $1,566,382$ natural numbers $n < 10^8$ of the form $n = pqr$, a product of three distinct odd primes, such that $A(n) = 1$. Bachman [6] proved that $A(pqr) = 1$ if $q \equiv -1 \pmod{p}$ and $r \equiv -1 \pmod{pq}$. Kaplan [21] proved a more general result, that $A(pqr) = 1$ when $r \equiv \pm 1 \pmod{pq}$. For $n = pqr < 10^8$, we find that $A(n) = 1$ if $q \equiv 1 \pmod{p}$ and $r \equiv \pm 2 \pmod{pq}$. The aforementioned families account for $414,832$ of these flat cyclotomic polynomials of order 3.

Flat cyclotomic polynomials of order 4

Noe [30] has calculated flat cyclotomic polynomials of order 4, for index $n < 5 \cdot 10^6$. We extend his result to $n < 3 \cdot 10^8$. There are $1,389$ such n for which $\Phi_n(z)$ is flat, and each of these $n = p_1p_2p_3p_4$ satisfies

$$p_2 \equiv -1 \pmod{p_1}, \quad p_3 \equiv \pm 1 \pmod{p_1p_2}, \quad p_4 \equiv \pm 1 \pmod{p_1p_2p_3}. \tag{5.10}$$

In addition, any cyclotomic polynomial $\Phi_n(z)$ of order four, with $n = p_1p_2p_3p_4 < 3 \cdot 10^8$ satisfying (5.10), is flat.

Are there flat cyclotomic polynomials of order 5 or greater?

For $n < 6.5 \cdot 10^8$, there is no cyclotomic polynomial $\Phi_n(z)$ of order 5 with height less than 4. Table A.8 shows the cyclotomic polynomials of order 5 and index n such that $\Phi_n(z)$ is flatter than any cyclotomic polynomial of order 5 and smaller index, for $n < 6.5 \cdot 10^8$. In an attempt to find a flat cyclotomic polynomial of order 5, we computed $A(n)$ for n, a product of 5 distinct primes such that for p dividing n, n/p satisfies the set of congruences (5.10). That is, we computed $A(n)$ for $n = p_1p_2p_3p_4p_5$ satisfying

$$\begin{aligned} p_2 \equiv -1 \pmod{p_1}, \quad p_3 \equiv -1 \pmod{p_1p_2}, \\ p_4 \equiv \pm 1 \pmod{p_1p_2p_3}, \quad p_5 \equiv \pm 1 \pmod{p_1p_2p_3p_4}. \end{aligned} \tag{5.11}$$

We only consider n satisfying (5.11) for which, given (p_1, p_2, p_3, p_4), p_5 is minimal for its congruence class modulo $p_1p_2p_3p_4$, because of the following theorem from Kaplan:

Theorem 5.1 (Kaplan, [22]). *Let $m > 0$ and let p, q be primes such that $m < p < q$ and $p \equiv q \pmod{m}$. Then $A(mp) = A(mq)$.*

We have calculated $A(n)$ for all such $n < 2^{63}$. There are 5349 such n. Of these, $A(n) = 2$ for 5212 cases and $A(n) = 3$ for the remaining ones. We list the smallest indices n of this form for which we have computed $A(n) = 2$ in Table A.9. The data for all 5349 cases can be found at our website.

5.3 Extrema of the kth cyclotomic polynomial coefficient

Let $\Phi_n(z) = \sum a_n(k) z^k$. Let $a(k) = \max_n |a_n(k)|$, and let $a^*(k) = \max_n a_n(k)$ and $a_*(k) = \min_n a_n(k)$ be the one-sided bounds. We also define

$$a^{**}(k) = \max_{\text{squarefree } n} a_n(k) \quad \text{and} \quad a_{**}(k) = \min_{\text{squarefree } n} a_n(k).$$

It is clear that $a^{**}(k) \le a^*(k)$ and $a_{**}(k) \ge a_*(k)$.

Bachman [4] showed that for a constant A_0, and for sufficiently large k,

$$\log a(k) = A_0 \frac{\sqrt{k}}{(\log k)^{1/4}} \left(1 + \mathcal{O}\left(\frac{\log\log k}{\sqrt{\log k}} \right) \right).$$

Gallot et al. [14] computed $a(k)$ for $k \le 30$. We calculated $a(k)$ for $k \le 172$ using a brute-force approach we detail below. Noe [29] calculated $a(k)$, for $k \le 1000$ using a brute-force approach for $k \le 128$, and a superior, fast method due to Grytczuk and Tropak [18] for larger k.

We verify his computation up to $k \le 172$, and for those k find the smallest index n for which we obtain $|a_n(k)| = a(k)$. It is immediate from the identity (3.8) that $a_n(k)$ depends on the divisors of n that are less than or equal to k. In particular, if p and q are distinct primes that are greater than k, then $a_n(k) = a_{npq}(k)$ and $a_{np}(k) = a_{nq}(k)$. Thus to calculate $a^{**}(k)$, we need only consider $a_n(k)$ for n of the form $n = m$ and $n = mq$, where m is a product of distinct primes less than or equal to k, and q is first prime greater than k.

We used this brute-force approach to calculate $a^{**}(k)$ and $a_{**}(k)$ for $0 \le k \le 172$. This entailed inspection of $\Phi_n(z)$ for every squarefree n that is a product of primes less than or equal to 173, the 40_{th} prime. There are $2^{40} > 10^{12}$ such n. We used a variant of the SPS

algorithm to obtain the first 211 terms of $\Phi_n(z)$; instead of truncating the power series of $\Phi_n(z)$ to degree $\phi(n)/2$, we truncate the power series to degree 210. For those applicable n with more than roughly 15 distinct prime factors, it is not reasonable to iterate through all the divisors $d|n$ in search of those for which $d \leq 210$. Rather, we iterate through the divisors with the least number of prime divisors first. Any product of four odd primes will exceed 210, thus we need only consider divisors that are products of three or less primes. Moreover, it is easy to verify that for $d = p_1p_2 \leq 210$, a product of two distinct odd primes with $p_1 < p_2$, that $p_1 \leq 11$. Similarly, for odd $d = p_1p_2p_3 \leq 210$, an odd product of 3 primes satisfying $p_1 < p_2 < p_3$, we have $p_1 = 3$.

Given odd n, we use Lemma 1.20 to obtain the truncated power series of $\Phi_{2n}(z)$. Since $\Phi_{2n}(z) = \Phi_n(-z)$ for odd $n > 1$, it follows that

$$a_{2n}(2k) = a_n(2k), \quad \text{and} \quad a_{2n}(2k+1) = -a_n(2k+1).$$

Given $a_n^{**}(d)$, for $0 \leq d \leq k$, one can obtain $a^*(k)$ by inspection. Suppose that $a^*(k) > a^{**}(k)$, then there exists some non-squarefree n for which $a_n(k) > a^{**}(k)$. Write $n = md$, where m is the squarefree part of n. By Lemma 1.18, $\Phi_n(z) = \Phi_m(z^d)$, and so if $d|k$, $a_n(k) = a_m(k/d)$, otherwise $a_n(k) = 0$. Thus $a^*(k) = \max_{d|k} a^{**}(k/d)$. Similarly, $a_*(k) = \min_{d|k} a_{**}(k/d)$. Typically we find that $a^*(k) = a^{**}(k)$. $k = 118$ is the least $k > 0$ such that $a(k) > k$, as $a^*(118) = 124$.

Another related problem is, given $b \in \mathbb{Z}$, find minimal k such that there exists n such that $a_n(k) = b$. We define

$$\alpha(b) = \min_{a_n(k)=b} k \quad (\text{for } b \in \mathbb{Z}) \quad \text{and}$$
$$\bar{\alpha}(b) = \min_{|a_n(k)|=b} k = \min(\alpha(b), \alpha(-b)) \quad (\text{for } b \geq 0),$$

where the minima are taken over all pairs (n, k) such that $n > 0, k \geq 0$.

In our computation of $a(k)$ we have simultaneously computed $\alpha(b)$, for $-927 \leq b \leq 927$, and the smallest n for which $a_n(\alpha(b)) = b$. Again by Lemma 1.18, we need only consider squarefree n to compute $\alpha(b)$. Suppose $b \neq 0$ and $a_{np^2}(k) = b$. Then, as $a_{np^2}(k) \neq 0$, p must divide k by Lemma 1.18, and $a_{np}(k/p) = b$. Thus $\alpha(b) \leq k/p < k$. Given that we know the maxima and minima of $a_n(k)$ for fixed $k \leq 172$, if the minimum k we have found for which $\exists n \ni a_n(k) = b$ is less than or equal to 172, then we know we have

the exact value of $\alpha(b)$. The same holds if the minimum such k equals 173; however, we cannot be certain that we have the smallest n for which $a_n(173) = b$.

Table A.10 shows $\bar{\alpha}(b)$ and least n such that $|a_n(\bar{\alpha}(b))| = b$ for select values of b. We extend results by Bosma [10], and by Grytczuk and Tropak [18]. Grytczuk and Tropak found results for $|b| \leq 10$. Bosma calculated results for $|b| \leq 50$. We have in fact calculated the smallest k such that $a_n(k) = b$ for $-927 \leq b \leq 927$. All of these results can be found at our website.

5.4 A look at the coefficients of $\Phi_n(z)$

We include here plots of $\Phi_n(z)$ for $n = 4849845$ (Figure 5.1), the product of the first 7 odd primes; $n = 40324935$ (Figure 5.2), a product of seven primes, for which $\Phi_n(z)$ has large height; $n = 1181895$ (Figure 5.3), the least n such that $A(n) > n$; and $n = 43730115$ (Figure 5.4), the least n for which $A(n) > n^2$. The plots were produced using Maple. We only plot a subset of the coefficients of $\Phi_n(z)$. We randomly chose a set of roughly 25,000 terms of $\Phi_n(z)$ of degree less than $\phi(n)/2$ and then used the palindromic property to extend the set to include higher-degree terms as well.

It appears that for some values of n, the coefficient plots appear "noisy," whereas, at least in the cases where for $\Phi_n(z)$ of particularly large height, the coefficients exhibit a general structure. We see, in the plot of the coefficients of $\Phi_{4849845}(z)$ that the coefficients form poorly-defined bands. In the the other plots these bands are much more well-defined. Each of the twelve bands in Figure 5.2 contains terms whose degree belongs to a particular congruence class modulo 12. Similarly, the plots of $\Phi_{1181895}(z)$ and $\Phi_{43730115}(z)$ have eight bands, each of which comprise terms of degree belonging to one congruence class modulo 8. This suggests, for instance, that the height of $(1 - z^8)\Phi_{43730115}(z)$ could be appreciably less than that of $\Phi_{43730115}(z)$. Indeed, we computed the height of $(1 - z^8)\Phi_{43730115}(z)$ to be $1,544,959,736,747$, which is more than a factor of 10^5 less than $A(43730115) = 862,550,638,890,874,931$. We note, moreover, that the plots of $\Phi_n(z)$ for $n = 1181895$ and $n = 43730115 = 1181895 \cdot 37$ appear somewhat similar.

Figure 5.1: The coefficients of $\Phi_n(z) = \sum_{k=0}^{\phi(n)} a_n(k) z^k$, for $n = 4849845$

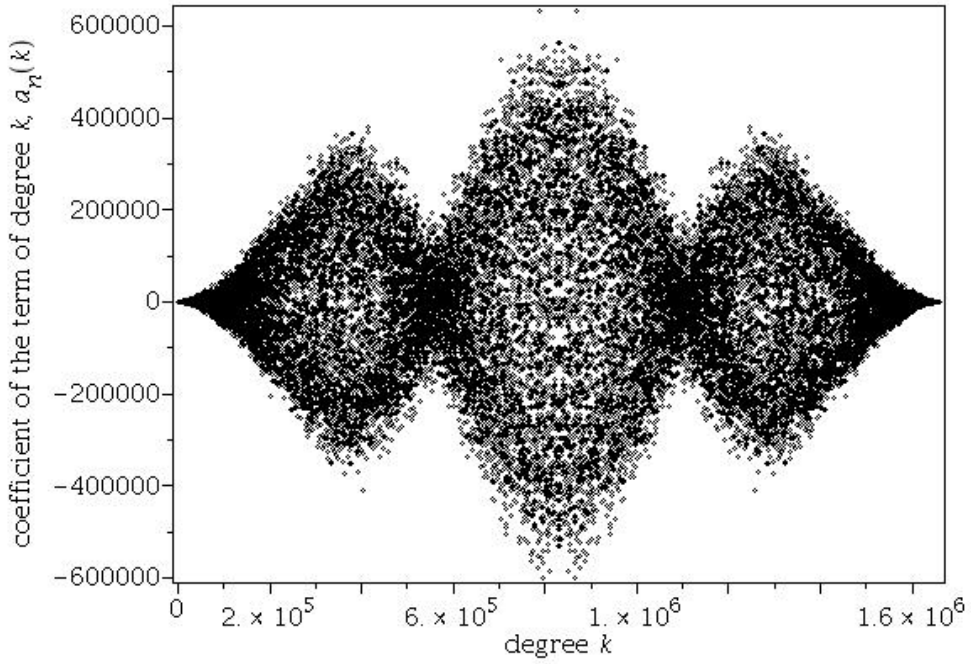


Figure 5.2: The coefficients of $\Phi_n(z) = \sum_{k=0}^{\phi(n)} a_n(k) z^k$, for $n = 40324935$

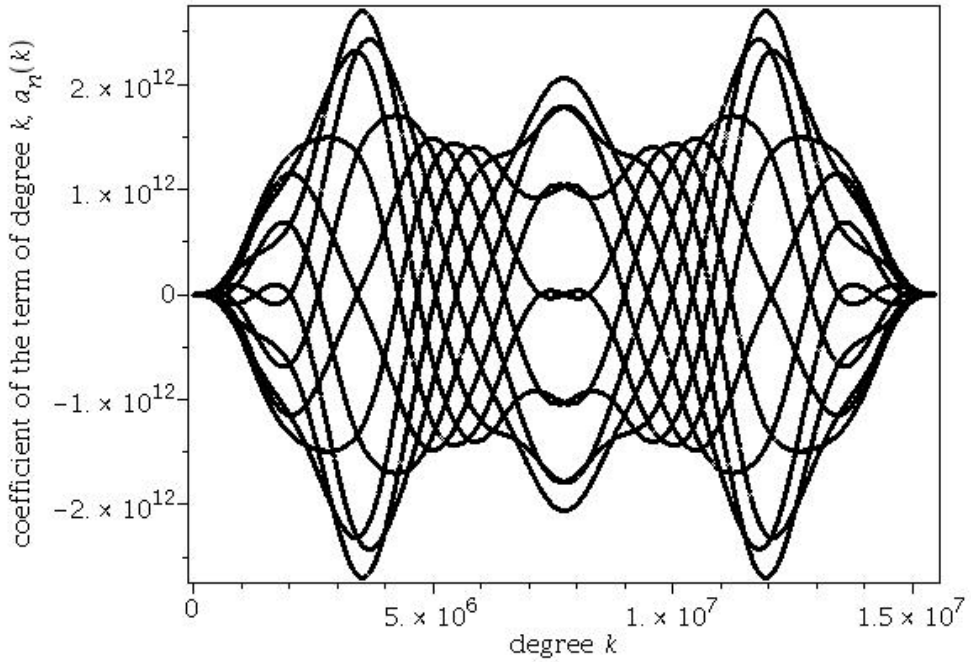


Figure 5.3: The coefficients of $\Phi_n(z) = \sum_{k=0}^{\phi(n)} a_n(k) z^k$, for $n = 1181895$

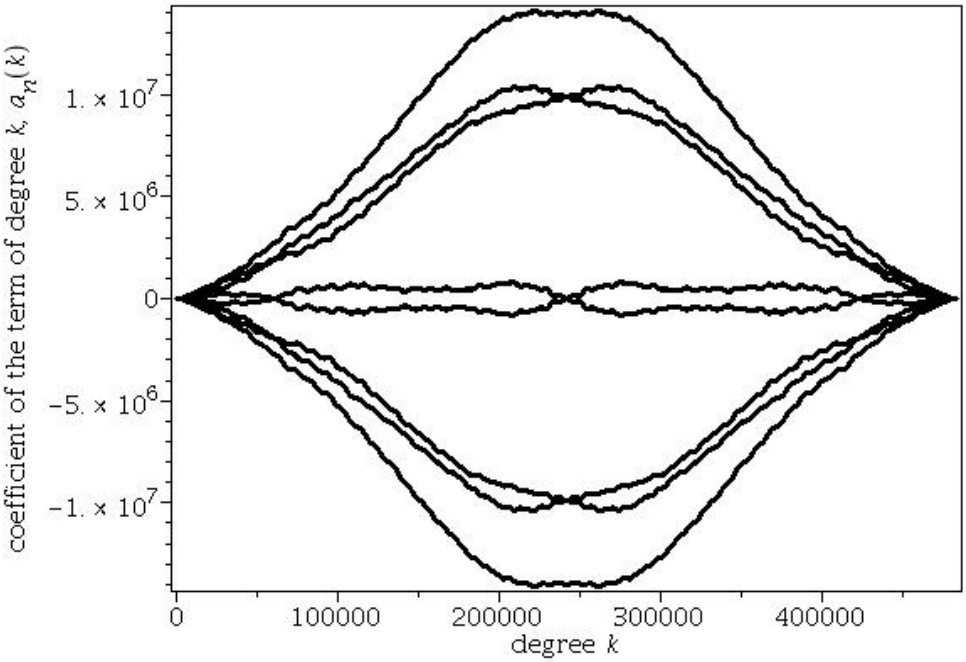
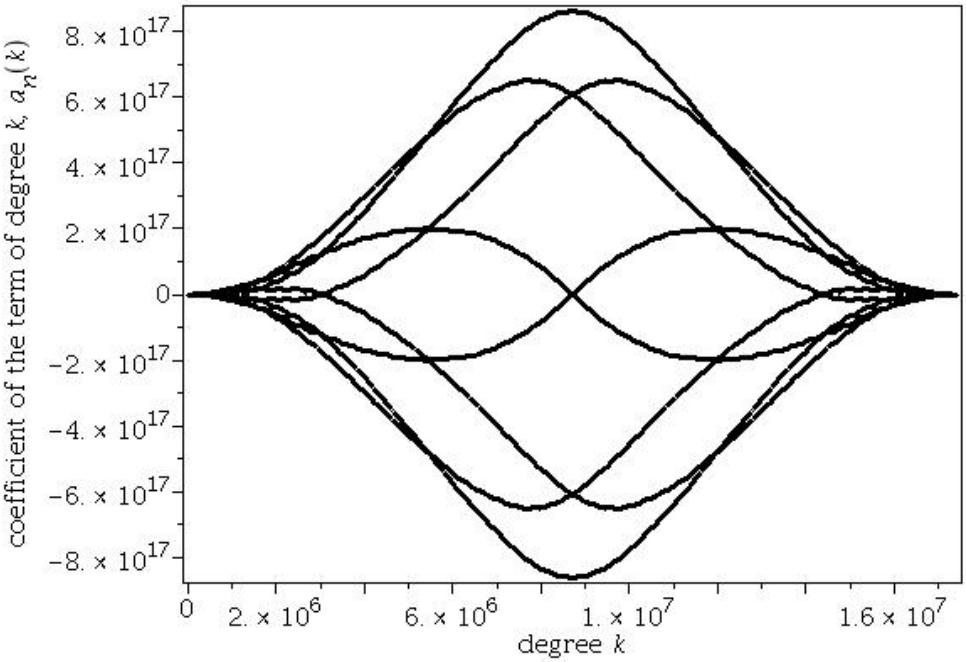


Figure 5.4: The coefficients of $\Phi_n(z) = \sum_{k=0}^{\phi(n)} a_n(k) z^k$, for $n = 43730115$



Appendix A

Data

Table A.1: n such that $A(n) > A(m)$ for $m < n$

n	$A(n)$	$\lceil \log_2 A(n) \rceil$	$\log_n A(n)$
1	1	0	-
105	2	1	0.14894
385	3	2	0.18454
1365	4	2	0.19204
1785	5	3	0.21496
2805	6	3	0.22569
3135	7	3	0.24172
6545	9	4	0.25007
10465	14	4	0.28513
11305	23	5	0.33596
17255	25	5	0.32994
20615	27	5	0.33178
26565	59	6	0.40026
40755	359	9	0.55423
106743	397	9	0.51683
171717	434	9	0.50384
255255	532	10	0.50415

Continued on Next Page...

Table A.1: n such that $A(n) > A(m)$ for $m < n$ – Continued

n	$A(n)$	$\lceil \log_2 A(n) \rceil$	$\log_n A(n)$
279565	1182	11	0.56415
327845	31010	14	0.81432
707455	35111	15	0.77704
886445	44125	15	0.78093
983535	59815	15	0.79709
1181895	14102773	24	1.17731
1752465	14703509	24	1.14795
3949491	56938657	26	1.17568
8070699	74989473	27	1.14016
10163195	1376877780831	41	1.73239
13441645	1475674234751	41	1.70710
15069565	1666495909761	41	1.70265
30489585	2201904353336	42	1.64919
37495115	2286541988726	42	1.63180
40324935	2699208408726	42	1.63449
43730115	862550638890874931	60	2.34738
169828113	31484567640915734941	65	2.36915
185626077	42337944402802720258	66	2.37364
416690995	80103182105128365570406901971	97	3.35316
437017385	86711753206816303264095919005	97	3.34912
712407185	11185937095152669880319825792 5	97	3.28132
1250072985	13756580004264445418853130688 6	97	3.20311
1311052155	19289231441599758355173100941 0	98	3.21195
1880394945	64540997036010911566826446181523888971563	136	4.40034
2317696095	67075962666923019823602030663153118803367	136	4.35946
7981921311	454 336118538773092209637015999240106863272841	149	4.50989
12436947159	633 62031348392041042436427665367419759880 4995	149	4.43815

Continued on Next Page...

Table A.1: n such that $A(n) > A(m)$ for $m < n$ – Continued

n	$A(n)$	$\lceil \log_2 A(n) \rceil$	$\log_n A(n)$
17917712785	8103387856491540894577 2816473092097967028572243597406763249828 27	213	6.23300
22084622735	9492813291464815330681 8482210296486781943218678482646529100926 51	213	6.18492
53753138355	18502043917986583739321 2415265919539992363783830789874059256100 51	214	5.98926
There may exist other values $5.4 \cdot 10^{10} < n < 99660932085$ not in this table for which $A(m) < A(n)$ for all $m < n$.			
66253868205	**18612363044507322761861 4172155463629537535125344944705583274334 58	214	5.93924
87840494385	**29548576895748088003903 0006730185431409340999694728017694779587 27	215	5.89111
99660932085	6126 7208717407836670896202324395260124725254 73 3381530786789617551493787739155364471853 70	292	7.98172

**We have yet to verify that there does not exist $m < n$ for which $A(m) \geq A(n)$

Table A.2: $A(n)$ for n a product of the k least odd primes

k	n	$A(n)$	$\log_n A(n)$
1	3	1	0
2	15	1	0
3	105	2	0.14894
4	1155	3	0.15579
5	15015	23	0.32604
6	255255	532	0.50415
7	4849845	*669606	0.87138
8	111546435	† 8161018310	1.23166
9	3234846615	† 2888582082500892851	1.94122
10	100280245065	380127943284004471680549555602695764	3.05238

*(Koshiba, 2002 [23]), † (Monagan, [2])

Table A.3: The least n for which $A(n) > 2^{133}$

n	factorization of n	$\log_2 A(n)$
$*\dagger 1880394945$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 43$	135.56734
$*\ddagger 2317696095$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 53$	135.62292
$*2580076785$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 59$	135.02048
$*2667537015$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 61$	135.22702
$\dagger\ddagger 2693538705$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 43 \cdot 53$	134.92366
$*2929917705$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 67$	134.84010
$\dagger 2998467615$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 43 \cdot 59$	135.21922
$\dagger 3100110585$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 43 \cdot 61$	134.76555
$\dagger 3405039495$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 43 \cdot 67$	134.92482
3436583865	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 37 \cdot 43 \cdot 53$	134.43396
$*3629599545$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 83$	134.51619
$\ddagger 3695785665$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 53 \cdot 59$	134.33890
$\ddagger 3821066535$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 53 \cdot 61$	134.51534
3825631095	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 37 \cdot 43 \cdot 59$	134.87970
3955313505	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 37 \cdot 43 \cdot 61$	134.34529
$\ddagger 4196909145$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 53 \cdot 67$	134.15068
$\dagger 4218183255$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 43 \cdot 83$	134.55022
4253640105	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 59 \cdot 61$	134.18318
4344360735	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 37 \cdot 43 \cdot 67$	134.57789
$*4416741615$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 101$	134.48033
4672030935	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 59 \cdot 67$	134.06365
$*4679122305$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 107$	134.18126
4715312745	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 37 \cdot 53 \cdot 59$	133.80302
$*4766582535$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 37 \cdot 109$	134.32245
4830404865	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 61 \cdot 67$	133.99686
4875153855	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 37 \cdot 53 \cdot 61$	134.43302
$\dagger 5132969985$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 43 \cdot 101$	133.93216
$\ddagger 5199156105$	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 19 \cdot 29 \cdot 53 \cdot 83$	133.81052
5245312215	$3 \cdot 5 \cdot 11 \cdot 13 \cdot 29 \cdot 37 \cdot 43 \cdot 53$	134.50069

*multiple of 43730115, †multiple of 50821485, ‡multiple of 62640435

Table A.4: The least n for which $A(n) > 2^{146}$

n	factorization of n	$\log_2 A(n)$
$*\dagger 7981921311$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 43 \cdot 47$	148.34860
$*\ddagger 9000889989$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 43 \cdot 53$	148.30726
$\dagger\ddagger 9838182081$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 47 \cdot 53$	148.04066
$*11378483571$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 43 \cdot 67$	147.56776
11433562959	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 43 \cdot 47 \cdot 53$	147.98987
$*12397452249$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 43 \cdot 73$	147.59878
$\dagger 12436947159$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 47 \cdot 67$	148.82845
$\dagger 13550703621$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 47 \cdot 73$	147.62967
$\ddagger 14024642541$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 53 \cdot 67$	147.21488
$*14095733379$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 43 \cdot 83$	147.40376
14453749401	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 43 \cdot 47 \cdot 67$	147.01078
$\ddagger 15280580679$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 53 \cdot 73$	147.05584
$\dagger 15406964391$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 47 \cdot 83$	147.46299
15748115019	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 43 \cdot 47 \cdot 73$	147.24087
16298908899	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 43 \cdot 53 \cdot 67$	147.58593
$*16473326961$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 43 \cdot 97$	147.04948
$\ddagger 17373810909$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 53 \cdot 83$	146.91472
$*17492295639$	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 37 \cdot 43 \cdot 103$	147.04717
17758512681	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 43 \cdot 53 \cdot 73$	148.58602
17815086471	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 47 \cdot 53 \cdot 67$	146.67615
17905391049	$3 \cdot 7 \cdot 13 \cdot 17 \cdot 23 \cdot 43 \cdot 47 \cdot 83$	147.04902

*multiple of 169828113, †multiple of 185626077, ‡multiple of 209323023

Table A.5: The least n of order 8 for which $A(n) > 2^{210}$

n	factorization of n	$\log_2 A(n)$
17917712785	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 43$	212.30000
22084622735	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 53$	212.52831
23161921405	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 43 \cdot 53$	211.82151
27918296665	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 67$	211.48422
29280164795	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 43 \cdot 67$	212.13758
30633508955	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 41 \cdot 43 \cdot 53$	212.12651
32746164745	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 31 \cdot 41 \cdot 43 \cdot 53$	211.15845
32918588605	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 79$	211.26572
34524373415	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 43 \cdot 79$	211.73429
36089505445	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 53 \cdot 67$	210.93612
37085498555	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 89$	211.25376
38725379245	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 41 \cdot 43 \cdot 67$	210.78141
38894547265	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 43 \cdot 89$	211.15920
41396095055	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 31 \cdot 41 \cdot 43 \cdot 67$	211.72594
42085790495	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 101$	211.07185
42553297465	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 53 \cdot 79$	210.71104
42919172485	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 103$	210.87071
44138755885	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 43 \cdot 101$	210.97029
45012790655	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 43 \cdot 103$	211.26476
45661268065	$5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 41 \cdot 43 \cdot 79$	210.63322

Table A.6: The least n of order 9 for which $A(n) > 2^{212}$

n	factorization of n	$\log_2 A(n)$
53753138355	$3 \cdot 5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 43$	213.49108
66253868205	$3 \cdot 5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 53$	213.49966
69485764215	$3 \cdot 5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 43 \cdot 53$	213.20507
83754889995	$3 \cdot 5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 41 \cdot 67$	212.90437
87840494385	$3 \cdot 5 \cdot 7 \cdot 17 \cdot 19 \cdot 29 \cdot 31 \cdot 43 \cdot 67$	214.16649

Table A.7: n such that $\bar{A}(n) = |\Psi_n(z)|_1 > \bar{A}(m) = |\Psi_m(z)|_1$ for all $m < n$

n	$\bar{A}(n)$	$\log_n \bar{A}(n)$
*1	1	-
561	2	0.109507
1155	3	0.155791
2145	4	0.180721
3795	5	0.195286
5005	7	0.228442
8645	9	0.242393
11305	21	0.326210
31395	26	0.314658
33495	38	0.349125
*40755	202	0.500057
*106743	341	0.503696
267995	530	0.501881
285285	836	0.535666
*327845	23247	0.791630
*983535	25685	0.735831
*1181895	9166109	1.146496
*3949491	43061336	1.157286
8273265	46017937	1.107731
*10163195	1085807040539	1.717668
*40324935	1187030460179	1.587581
*43730115	439343761754389367	2.309032
*169828113	24011100366340974489	2.354847
*416690995	532807494213153411635549112 73	3.332621
*1250072985	98339426540513481131872876038	3.187087
*1880394945	32415199783443679596598721580288327891505	4.368089

*n also appears in Table A.1

Table A.8: $\Phi_n(z)$ of order 5 that are flatter than all $\Phi_m(z)$ of order 5 for $m < n$

n	factorization of n	$A(n)$
15015	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	23
23205	$3 \cdot 5 \cdot 7 \cdot 13 \cdot 17$	21
31395	$3 \cdot 5 \cdot 7 \cdot 13 \cdot 23$	15
574665	$3 \cdot 5 \cdot 7 \cdot 13 \cdot 421$	14
774795	$3 \cdot 5 \cdot 7 \cdot 47 \cdot 157$	13
1331715	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 1153$	12
2666895	$3 \cdot 5 \cdot 7 \cdot 11 \cdot 2309$	9
3725085	$3 \cdot 5 \cdot 7 \cdot 13 \cdot 2729$	7
40765935	$3 \cdot 5 \cdot 7 \cdot 43 \cdot 9029$	6
48713385	$3 \cdot 5 \cdot 7 \cdot 47 \cdot 9871$	5
76762245	$3 \cdot 5 \cdot 7 \cdot 59 \cdot 12391$	4

Table A.9: $\Phi_n(z)$ of order 5 such that $A(n) = 2$ for select n

n	factorization of n
1147113361785	$3 \cdot 5 \cdot 29 \cdot 1741 \cdot 1514671$
2294224451565	$3 \cdot 5 \cdot 29 \cdot 1741 \cdot 3029339$
2576062979535	$3 \cdot 5 \cdot 29 \cdot 2609 \cdot 2269829$
7157926096635	$3 \cdot 5 \cdot 29 \cdot 4349 \cdot 3783629$
7157929880265	$3 \cdot 5 \cdot 29 \cdot 4349 \cdot 3783631$
14031384951165	$3 \cdot 5 \cdot 29 \cdot 6089 \cdot 5297431$
15456385821615	$3 \cdot 5 \cdot 29 \cdot 2609 \cdot 13618981$
36654908721735	$3 \cdot 5 \cdot 29 \cdot 6959 \cdot 12108659$
39282436838685	$3 \cdot 5 \cdot 59 \cdot 3541 \cdot 12535141$
44151142013985	$3 \cdot 5 \cdot 59 \cdot 5309 \cdot 9396929$
44151151410915	$3 \cdot 5 \cdot 59 \cdot 5309 \cdot 9396931$
46392857518515	$3 \cdot 5 \cdot 29 \cdot 7829 \cdot 13622461$
50859294230685	$3 \cdot 5 \cdot 89 \cdot 2671 \cdot 14263141$
55013978795385	$3 \cdot 5 \cdot 29 \cdot 6961 \cdot 18168211$
57276587912835	$3 \cdot 5 \cdot 29 \cdot 8699 \cdot 15136259$
64441098282135	$3 \cdot 5 \cdot 29 \cdot 13049 \cdot 11352629$
64441109634765	$3 \cdot 5 \cdot 29 \cdot 13049 \cdot 11352631$
69589277763735	$3 \cdot 5 \cdot 29 \cdot 7829 \cdot 20433689$
73341424450815	$3 \cdot 5 \cdot 29 \cdot 13921 \cdot 12111269$
85914891329415	$3 \cdot 5 \cdot 29 \cdot 8699 \cdot 22704391$
91637282399415	$3 \cdot 5 \cdot 29 \cdot 6959 \cdot 30271651$
96836813004615	$3 \cdot 5 \cdot 29 \cdot 11311 \cdot 19681139$
98219673468435	$3 \cdot 5 \cdot 29 \cdot 6089 \cdot 37082009$
98284212060735	$3 \cdot 5 \cdot 29 \cdot 6091 \cdot 37094191$
103395514616565	$3 \cdot 5 \cdot 29 \cdot 16529 \cdot 14380231$
117714212390685	$3 \cdot 5 \cdot 59 \cdot 3539 \cdot 37584179$
126365409450465	$3 \cdot 5 \cdot 29 \cdot 6091 \cdot 47692529$
128365940429115	$3 \cdot 5 \cdot 29 \cdot 6961 \cdot 42392489$
138626726606985	$3 \cdot 5 \cdot 29 \cdot 19139 \cdot 16650929$
193622281319715	$3 \cdot 5 \cdot 29 \cdot 22619 \cdot 19678531$

Table A.10: $\bar{\alpha}(b)$, the least k for which b occurs as $|a_n(k)|$; and the least n for which $|a_n\big(\bar{\alpha}(k)\big)| = b$, for select $b \leq 927$

b	$\bar{\alpha}(b)$	n
1	0	1
2	7	105
3	17	323323
4	23	1062347
5	30	37182145
10	52	30704573184285
20	70	152125131763605
30	82	307444891294245705
40	89	1352450076803386856295
50	95	3929160775540133527939545
100	112	2380678513899766904578570315 5
150	123	11992411764462614086353260819346129198105
200	132	16293842598153406076363508397702918810966 3335
250	140	39578916714398066291594920195861812535
300	143	23269755710293262865989902525327960747814644577 55
350	149	15701405093677855556359423959350086682055
400	153	54121313707641277576363900172106 95
450	156	51649375850708686273776484661179 45
500	158	59210574326445961491068454262921019714541080 35
550	160	61075474305231661065590295342068138445707728 55
600	163	50967866897743398269290966947741571435
650	165	66245494048399937113769656749604705225179624 55
700	167	41211036991280460777846257111155083155
750	168	64500267256926894396371059566857006022102554 45
800	170	52384830864766841980950592110874121661984 5
850	172	47802762363393533236767997900286819848886 5
900	173	*48064432442930401405202089668740173483676 5
927	173	*12691403741168443218970585192279277799437804512720731212914757 05

*There may exist smaller n for which $|a_n(k)| = b$.

Appendix B

Source code

B.1 A C implementation of SPS and SPS-Psi

```
#define Long long long int

/* SPS
Input:
   -n>0, a squarefree integer with at most 10 prime divisors
   -phi_n = phi(n)
   -omega, the number of prime divisors of n
   -P, an array containing the prime divisors of n
   -lambda, a boolean
   -A, an array of integers
Output:
   -if lambda is nonzero, procedure SPS writes the first half of the
      coefficients of Phi_n(z), the nth cyclotomic polynomial to array
      A, else SPS writes the first half of the coefficients of
      Psi_n(z), the nth inverse cyclotomic polynomial to A.
*/
Long SPS(Long n, Long phi_n, char omega, Long P[10], char l, Long *A){
   Long d,i,j,k,D; char isnumer;
   /* D is the degree bound */
   D = lambda ? phi_n/2 : (n-phi_n)/2;
   /* Omega is the number of d|n we must iterate through */
   Omega = (1<<omega)-1;
```

```
   for(i=0; i<Omega; i++){
       d = 1; k = i;
      if(l){ isnumer = 1; } else { isnumer = 0; }
          /* generate the divisor d|n*/
       for(j=0; j<omega; j++){
          if(k % 2) d *= P[j]; else isnumer = !isnumer; k /= 2; }
       /* multiply by 1-x^d */
       if (isnumer) for(k=deg; k>=d; k--) A[k] -= A[k-d];
       /* divide by 1-x^d */
       else for(k=d; k<=deg; k++) A[k] += A[k-d];
   }
}
```

B.2 A C implementation of SPS4

```
#define Long long long int

/* SPS4
Input:
   -m,e : squarefree integers
   -phi_m = phi(m)
   -lambda : a boolean
   -Df, degree of input polynomial f
   -D, the least degree of a polynomial we know will occur at a later
      stage of computation
   -A, an array containing the terms of f up to degree
      floor( min(Df,D)/2 )
   -P, a list of distinct primes, in increasing order, that contains
      all the prime divisors of m
   -mark, an integer whose ith bit is 1 if P[i] divides m
Output:
    -If lambda is nonzero, SPS4 computes the terms of g=f*Phi_m(z^e),
       where Phi_m is the mth cyclotomic polynomial, up to degree
       floor( min(Dg,D)/2 ), where Dg is the degree of g.  The
       computed coefficients of g are written to array A.
    -If lambda=0, SPS4 instead computes the terms of g=Psi_m(z^e),
       where Psi_m is the mth inverse cyclotomic polynomial.
    -SPS4 returns Dg, the degree of g  */
```

```
Long SPS4(Long m, Long phi_m, Long e, char lambda, Long Df, Long D,
 Long * A, Long P[10], int mark){

   Long i,k,d;
   Long Dg = Df+e*(lambda ? phi_m : m - phi_m);
   Long D1 = Dg < D ? Dg : D;
   Long e1 = e, m1 = m, mark1=mark;

   for(k=mark,i=0; k>0; k=k>>1,i++){
      if(k&1){
         m1/=P[i]; phi_m/=P[i]-1; mark1 -= (1<<i);
         if(m1>1 || !lambda){
            Df = SPS4(m1, phi_m, e1, !lambda, Df, D1, A, P, mark1); }
         e1 *= P[i];
   }}
   if (!lambda) return Dg;

   /* Generate higher-degree terms */
   Long halfDf = Df>>1;  D1=D1>>1;
   k=halfDf+1; i=Df-halfDf-1;
   if(Df & 1){ while(i>=0 && k<=D1){ A[k] = -A[i]; i--; k++; }}
   else { while(i>=0 && k<=D1){ A[k] = A[i]; i--; k++;}}

   /* multiply by 1/(1-z^(me/p)) for p|m */
   for(k=0, m=m*e; mark>0; k++,mark=mark>>1){
      if(mark & 1){ d=m/P[k];
         for(i=d; i<=D1; i++){ A[i] = A[i]+A[i-d]; }
   }}

   /* multiply by 1-z^(me) */
   for(i=D1; i>=m; i--){ A[i] = A[i] - A[i-m]; }
   return Dg;
}
```

Bibliography

[1] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. available online at http://www.intel.com/Assets/PDF/manual/248966.pdf.

[2] Andrew Arnold and Michael Monagan. Calculating cyclotomic polynomials. To appear in Mathematics of Computation. Available at `http://www.cecm.sfu.ca/~ada26/cyclotomic/PDFs/CalcCycloPolysApr2010.pdf`.

[3] Andrew Arnold and Michael Monagan. A high-performance algorithm for calculating cyclotomic polynomials. In *PASCO '10: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pages 112–120, New York, NY, USA, 2010. ACM.

[4] G. Bachman. On the coefficients of cyclotomic polynomials. *Mem. Amer. Math. Soc.*, 106(510):vi+80, 1993.

[5] G. Bachman. Flat cyclotomic polynomials of order three. *Bull. London Math. Soc.*, 38(1):53–60, 2006.

[6] G. Bachman. Flat cyclotomic polynomials of order three. *Bull. London Math. Soc.*, 38(1):53–60, 2006.

[7] A. S. Bang. Om ligningen $\phi_n(x) = 0$. *Nyt Tidsskrift for Mathematik*, (6):6–12, 1895.

[8] P.T. Bateman, C. Pomerance, and R.C. Vaughan. On the size of the coefficients of the cyclotomic polynomial. In *Topics in classical number theory, Vol. I, II (Budapest, 1981)*, volume 34 of *Colloq. Math. Soc. János Bolyai*, pages 171–202. North-Holland, Amsterdam, 1984.

[9] D.M. Bloom. On the coefficients of the cyclotomic polynomials. *Amer. Math. Monthly*, 75:372–377, 1968.

[10] W. Bosma. Computation of cyclotomic polynomials with Magma. In *Computational algebra and number theory (Sydney, 1992)*, volume 325 of *Math. Appl.*, pages 213–225. Kluwer Acad. Publ., Dordrecht, 1995.

[11] Alina Cojocaru, J W Bruce, and Ram Murty. *An Introduction to Sieve Methods and Their Applications; electronic version*. Cambridge Univ. Press, Cambridge, 2005.

[12] S. Elder. Flat cyclotomic polynomials : A new approach.

[13] P. Erdős and R.C. Vaughan. On the coefficients of the cyclotomic polynomial. *Bull. Amer. Math. Soc.*, 52:179–184, 1946.

[14] Y. Gallot, P. Moree, and H. Hommerson. Value distribution of cyclotomic polynomial coefficients. Available at `http://arxiv.org/abs/0803.2483`.

[15] D. J. H. Garling. *A course in Galois theory*. Cambridge University Press, Cambridge, 1986.

[16] K.O. Geddes, S.R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Boston, 1992.

[17] Torbjorn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *In Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 61–72, 1994.

[18] A. Grytczuk and B. Tropak. A numerical method for the determination of the cyclotomic polynomial coefficients. In *Computational number theory (Debrecen, 1989)*, pages 15–19. de Gruyter, Berlin, 1991.

[19] Jr H.W. Lenstra. Vanishing sums of roots of unity. In *Proceedings, Bicentennial Congress Wiskundig Genootschap (Vrije Univ., Amsterdam, 1978), Part II*, volume 101 of *Math. Centre Tracts*, pages 249–268, Amsterdam, 1979. Math. Centrum.

[20] N. Kaplan. Personal correspondence.

[21] N. Kaplan. Flat cyclotomic polynomials of order three. *J. Number Theory*, 127(1):118–126, 2007.

[22] Nathan Kaplan. Flat cyclotomic polynomials of order four and higher. *Integers*, 10:A30, 357–363, 2010.

[23] Y. Koshiba. On the calculations of the coefficients of the cyclotomic polynomials. *Rep. Fac. Sci. Kagoshima Univ.*, (31):31–44, 1998.

[24] Y. Koshiba. On the calculations of the coefficients of the cyclotomic polynomials. II. *Rep. Fac. Sci. Kagoshima Univ.*, (33):55–59, 2000.

[25] Serge Lang. *Algebra*, volume 211 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, third edition, 2002.

[26] Niels Moller and Torbjorn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 99(PrePrints), 2010.

[27] Pieter Moree. Inverse cyclotomic polynomials. *J. Number Theory*, 129(3):667–680, 2009.

[28] T.D. Noe. Personal correspondence.

[29] T.D. Noe. Maximum possible magnitude of the x^n coefficient of a cyclotomic polynomial. Sequence A138474 in N. J. A. Sloane (Ed.), The On-Line Encyclopedia of Integer Sequences (2008), published electronically at `http://www.research.att.com/~njas/sequences/A138474`.

[30] T.D. Noe. Numbers n such that phi(n,x) is a flat cyclotomic polynomial of order four. Sequence A117318 in N. J. A. Sloane (Ed.), The On-Line Encyclopedia of Integer Sequences (2008), `http://www.research.att.com/~njas/sequences/A117318`.

[31] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, Cambridge, second edition, 2003.