Issues in the Implementation of Multiplication and Factoring Algorithms in Galois Fields

by

Alec Sobeck

Bachelor of Computer Science, UNB, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Masters of Computer Science

In the Graduate Academic Unit of your GAU

Supervisor(s):	Rod Cooper, Faculty of Computer Science, UNB
	Ali Ghorbani, Faculty of Computer Science, UNB
Examining Board:	Michael Fleming, Faculty of Computer Science, UNB, Chair
	Rongxing Lu, Faculty of Computer Science, UNB
	Brent Petersen, Electrical and Computer Engineering, UNB

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May 2020

Abstract

Cryptography in finite fields often requires factoring polynomials. As a result, there is value to understanding the different approaches to polynomial factoring algorithms and their performance. First, several of the approaches to univariate polynomial multiplication are explored. These polynomial factoring algorithms are then used to solve the problem of univariate polynomial factorization in a Galois field. An emphasis is placed on the theoretical running time and memory requirements of these algorithms, as well as the actual running time (in seconds) of some sample problems. This is done to give an idea of the real computing costs of running these algorithms. After covering univariate polynomial multiplication and factoring, the more complex problem of factoring with a field extension is explored. One approach to this problem is to modify the univariate polynomial factoring algorithms (such as Cantor-Zassenhaus) to account for the more complicated finite field structure. Finally, a novel application of the Cantor-Zassenhaus algorithm is developed.

Contents

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Factoring problems are at the core of many cryptography problems. The best known of these problems tend to involve integer factorization, such as the RSA algorithm. However, another useful set of factoring algorithms involve the factoring of polynomials in Galois fields.

The core building blocks of many modern polynomial factoring algorithms are the ability to compute a greatest common divisor (gcd), polynomial multiplication, and computing polynomial remainders. The choice of algorithms for these problems has a large effect on the overall running time of the factoring algorithms. This thesis will explore several important algorithms such as the Euclidean algorithm for solving the gcd, Karatsuba polynomial multiplication, and polynomial long division. Using these building blocks, several factoring algorithms are implemented and compared, including: the brute force method, Berlekamp's algorithm for polynomial factorization, Cantor-Zassenhaus factorization, and Shoup's polynomial factorization algorithm. These algorithms use

different techniques to extract factors and have different expected running times.

1.2 Contributions

The algorithms covered in this thesis are fundamental building blocks of cryptography algorithms. Vendor languages such as Maple, Magma and Mathematica often provide implementations of these algorithms but may be difficult to incorporate into other systems. Vendor languages may also be closed source, making it difficult to know the exact performance of the algorithms and the details of their construction.

This thesis explains fundamental algorithms in finite field cryptography, including: the greatest common divisor, polynomial multiplication, polynomial division, powmods and polynomial factoring. Summaries of the theoretical time and space complexity are included for all algorithms. Next, these algorithms are all implemented in Rust and tested on randomly generated data. The results of some of these tests are summarized in the Algorithm Performance section. Using these fundamental algorithms, the problem of factoring with a finite field extension is explored in detail. This problem of factoring with a field extension is poorly covered in the literature. One of the major contributions of this thesis is an approach to solving this problem and a corresponding analysis of the theoretical time complexity of the algorithm presented.

Finally, an approach to factoring univariate polynomials in finite fields is presented that makes use of only the Cantor-Zassenhaus equal degree factorization algorithm. This approach to factoring avoids the need for a distinct degree factorization algorithm.

Rust was chosen for its strengths as a secure programming language. Most of the programming languages that offer the best performance are unsafe languages, such as C and C++. Rust maintains a similar level of performance to these un-

safe languages while implementing various features to improve program safety and make it easier to write secure programs. These language features include Borrows, Moves and Ownership [1], allowing the compiler to verify memory safety and prevent multithreading errors at compile time. In many other languages, these errors are only detectable at runtime, making them much harder to find and fix.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2, the Background, describes relevant algorithms used in the thesis and summarizes the relevant literature. Chapter 3, Algorithm Performance, covers implementations of algorithms in the background section and how their real running times compare to the theoretical running times. Chapter 4, Factoring Irreducibles over Irreducibles, explores one possible approach to factoring with field extensions, a problem that is poorly covered in the literature. Chapter 5, Cantor-Zassenhaus Arbitrary Polynomial Factoring Algorithm, covers a novel approach to factoring with Cantor-Zassenhaus that does not require a distinct degree factoring algorithm. Chapter 6, Conclusions and Future Work, summarizes the results in the other sections and some related problems that could be explored.

Chapter 2

Background

This chapter covers core algorithms and applications of finite field theory. Each subsection covers a different algorithm, including pseudocode and theoretical analysis, based on the literature. Sources for each algorithm are provided in the appropriate section alongside a description of their content, instead of an independent chapter focused on a literature review.

2.1 Finite Fields

A finite field is a mathematical structure known as a field with the added constraint that only a fixed number of elements exist for that field at all times. Finite fields may also be refered to as Galois fields, in honour of the first mathematician to define them, Évariste Galois. A field structure has several properties that hold for all elements in the field at all times [2]. They are as follows:

1. Field elements can be added together to produce a third field element c i.e. $a + b = c$

2. Field elements can be multiplied together to produce a third field element

c i.e. $a \times b = c$

3. Field elements are commutative under addition i.e. $a + b = b + a$
4. Field elements are commutative under multiplication i.e. $a \times b = b \times a$
5. Field elements are associative under addition i.e. $a + (b + c) = (a + b) + c$
6. Field elements are associative under multiplication i.e. $a \times (b \times c) = (a \times b) \times c$
7. Field elements have an additive identity such that: $a + 0 = a$
8. Field elements have a multiplicative identity such that: $a \times 1 = a$
9. Field elements satisify the distributive property of multiplication over addtion i.e. $a \times b + a \times c = a \times (b + c)$
10. Field elements, except for 0, have multiplicative inverses denoted a^{-1}, such that $a \times a^{-1} = 1$

2.1.1 Finite Fields mod p

An easy way to define a finite field is using a prime number. Suppose we call this prime p. The elements of such a field are the numbers 0, 1, 2, ..., $(p - 1)$ for that prime p and this field as a whole can be referred to as GF(p). Even though there are a finite number of elements in a field constructed this way, the above field properties all hold. The obvious problem with this construction is that the addition or multiplication of two field elements can be larger than the prime. Finite fields address this problem using the idea of equivalence. Certain numbers are equivalent to each other, even though they are not necessarily the exact same number. It is the case that a and b are equivalent, denoted as:

$a \equiv b \pmod{p}$

if $a - b = kp$, for some integer k. Stated another way, a and b are equivalent if the remainder of a divided by p and the remainder of b divided by p are the same. The numbers 0, 1, ..., $(p-1)$ are consequently special and are called the residue classes. At any time, any number can be reduced to its residue class by taking its remainder on division by the prime p. In this way, even computations whose result is greater than p or less than zero can be converted to a value between 0 and $(p-1)$ at any time.

Example

For this example, let $p = 29$ be the prime. The residue classes of this finite field are therefore exactly the numbers: 0, 1, 2, ... , 27, 28. Any of these numbers may be added, multiplied, and all have inverses. Here are some examples of basic arithmetic:

- $5 + 1 \equiv 6 \pmod{29}$.
- $20 + 15 \equiv 35 \equiv 6 \pmod{29}$. Note that 35 and 6 are equivalent because 35 divided by 29 gives a remainder of 6.
- $10 \times 10 \equiv 100 \equiv 13 \pmod{29}$. Note that 100 and 13 are equivalent because 100 divided by 29 has remainder 13.
- The inverse of 10 is 3 as $10 \times 3 \equiv 30 \equiv 1 \pmod{29}$. Using the notation above, $10^{-1} \equiv 3 \pmod{29}$

2.1.2 Finite Fields mod p^n

Some of the algorithms in this thesis rely on a more complicated finite field structure. This more complicated field is defined by both a prime p and an

irreducible[1] polynomial $I(x)$ of degree n. The elements of such a finite field are polynomials of degree less than I, with coefficients in the range [0, $p-1$]. There are exactly p^n such polynomials, so this type of finite field is described as GF(p^n). Polynomials with integer coefficients in the range [0, $p-1$] are also, more generally, elements of $F_p[x]$ which is a useful shorthand. An example of an irreducible polynomial is $x^5 + 12x^4 + 18x^3 + 4x^2 + 15x + 15 \pmod{29}$. There is no possible way to factor this into a product of simpler polynomials over $F_p[x]$. These more complex GF(p^n) fields function in essentially the same way as a GF(p) field. The above properties, such as addition and multiplication, still hold except the operands are now polynomials instead of integers. Equivalence between polynomials is also similar. The *mod* used for GF(p) fields is often changed to *modd* to signify that the field is defined with both a prime and a polynomial. More precisely, equivalence is defined as:

$$a(x) \equiv b(x) \text{ (modd } p \text{ a prime, } I(x) \text{ an irreducible)}$$

if $a(x) - b(x) = k(x)I(x)$, for $a(x), b(x), k(x), I(x) \in F_p[x]$. This is effectively the same idea as for GF(p), except all values present are now polynomials in $F_p[x]$ instead of being integers.

Example

Let the field be defined by:

- $p = 29$
- $I(x) = x^5 + 12x^4 + 18x^3 + 4x^2 + 15x + 15$

Two possible field elements are the following. Each has coefficients between 0 and (p − 1) and degree less than $I(x)$:

[1] An irreducible polynomial is one that cannot be factored into a product of lower degree polynomials. Equivalently, there are no polynomials of lower degree that divide it and give a remainder of exactly 0.

- $a(x) = 3x^4 + 9x^3 + x^2 + 15x + 2$
- $b(x) = 4x^4 + 12x^3 + 8x^2 + 18x + 1$

A few possible computations using $a(x)$ and $b(x)$ are addition, multiplication, and inverses:

- $a(x) + b(x) \equiv (3x^4 + 9x^3 + x^2 + 15x + 2) + (4x^4 + 12x^3 + 8x^2 + 18x + 1) \equiv 7x^4 + 21x^3 + 9x^2 + 33x + 3 \equiv 7x^4 + 21x^3 + 9x^2 + 4x + 3 \pmod{p,\ I(x)}$. In the case of addition, all that is required is that the coefficients be reduced modulo p because the degree of the answer never increases.
- $a(x) \times b(x) \equiv (3x^4 + 9x^3 + x^2 + 15x + 2)(4x^4 + 12x^3 + 8x^2 + 18x + 1) \equiv 12x^8 + 72x^7 + 136x^6 + 198x^5 + 361x^4 + 171^3 + 287x^2 + 51x + 2 \equiv 5x^4 + 21x^3 + 24x^2 + 15x + 10 \pmod{p,\ I(x)}$. Multiplication is more complicated. First, the polynomials are multiplied together, then the coefficients are reduced modulo p, followed by division by the irreducible $I(x)$ to get the remainder. This remainder is the simplest possible representation in this Finite Field. Some of the approaches to performing this multiplication are covered later on.
- $a(x)^{-1} = 16x^4 + 17x^3 + 25x^2 + 23x + 19 \pmod{p,\ I(x)}$. This can be verified by multiplying $a(x) \times a(x)^{-1} \equiv (3x^4 + 9x^3 + x^2 + 15x + 2)(16x^4 + 17x^3 + 25x^2 + 23x + 19) \equiv 1 \pmod{p,\ I(x)}$ The Extended Euclidean Algorithm will be covered later, which can be used to solve for inverses.

2.2 Polynomial Remainders

One of the principal operations required to work in a finite field $\mathrm{GF}(p^n)$ is the computation of polynomial remainders modulo p and $I(x)$. Two possible methods of computing polynomial remainders are using long division, as well

as a precomputed table of powers. The precomputed table of powers has the potential to be faster on certain problems due to having a slightly different running time.

2.2.1 Long Division

The schoolbook method of long division is named such because it is one of the primary methods of division taught to students learning to divide. The process is relatively straightforward. On every iteration, the algorithm determines what multiple of the divisor is necessary to eliminate the highest remaining power of x in the dividend. This gives the next highest power of the quotient. When whatever is left has degree less than the divisor, the algorithm halts and produces whatever is left as the remainder. The original *dividend* is equal to the *divisor* times *quotient* plus *remainder*. Let $a(x)$ and $b(x)$ be the polynomials involved in the long division $a(x)/b(x)$. A quotient $q(x)$ and remainder $r(x)$ are produced such that $a(x) = b(x)q(x) + r(x)$. As a result, long division can be used to directly compute polynomial remainders.

Example

Consider the case of dividing $x^4 + 3x^3 + 5x + 4$ by $x^2 + 6$. The computation involves three steps in this case:

1. x^2 divides into x^4 exactly x^2 times. Write (x^2) above and subtract $(x^2)(x^2 + 6) = x^4 + 6x^2$:

$$
\begin{array}{rl}
 & \qquad\qquad\quad x^2 \qquad\qquad\quad \\
\hline
x^2+6\big) & \quad x^4+3x^3 \qquad\quad +5x \;\; +4 \\
 & \quad -x^4 \qquad\quad -6x^2 \\
\cline{2-2}
 & \qquad\quad 3x^3-6x^2 \;\; +5x
\end{array}
$$

2. x^2 divides into $3x^3$ exactly $3x$ times. Write $(3x)$ above and subtract $(3x)(x^2+6) = 3x^3+18x$:

$$
\begin{array}{rl}
 & \qquad\qquad\quad x^2 \;\; +3x \qquad\quad \\
\hline
x^2+6\big) & \quad x^4+3x^3 \qquad\quad +5x \;\; +4 \\
 & \quad -x^4 \qquad\quad -6x^2 \\
\cline{2-2}
 & \qquad\quad 3x^3-6x^2 \;\; +5x \\
 & \qquad -3x^3 \qquad\quad -18x \\
\cline{3-3}
 & \qquad\qquad -6x^2-13x \;\; +4
\end{array}
$$

3. x^2 divides into $-6x^2$ exactly -6 times. Write (-6) above and subtract $(-6)(x^2+6) = -6x^2-36$:

$$
\begin{array}{rl}
 & \qquad\qquad\quad x^2 \;\; +3x \;\; -6 \\
\hline
x^2+6\big) & \quad x^4+3x^3 \qquad\quad +5x \;\; +4 \\
 & \quad -x^4 \qquad\quad -6x^2 \\
\cline{2-2}
 & \qquad\quad 3x^3-6x^2 \;\; +5x \\
 & \qquad -3x^3 \qquad\quad -18x \\
\cline{3-3}
 & \qquad\qquad -6x^2-13x \;\; +4 \\
 & \qquad\qquad\quad 6x^2 \qquad\quad +36 \\
\cline{4-4}
 & \qquad\qquad\qquad -13x+40
\end{array}
$$

This gives polynomials $q(x) = x^2+3x-6$ and $r(x) = -13x+40$ such that $a(x) = b(x) \times q(x) + r(x)$

Algorithm Analysis

This algorithm has a simple, well defined set of rules and translates well to computer code. Knuth covers polynomial division in detail in his book Seminumerical Algorithms [3]. Suppose this algorithm is operating on two polynomials $a(x)$ and $b(x)$ with $\deg(a(x)) = m$, $\deg(b(x)) = n$, $m \geq n$ (otherwise the division is trivial). Also assume that the operation is $a(x)$ / $b(x)$. The time complexity is then $\mathcal{O}(n(m-n))$. Note that due to constants being dropped, this Big-O value is slightly misleading. One iteration is still required when $m = n$, even though $n(m-n)$ would be zero. Each of the subtraction steps affects n terms (every coefficient in $b(x)$ is subtracted from $a(x)$). At most, $(m-n+1)$ subtraction steps are required because the degree of the remaining polynomial being operated on decreases by one on each step, until it is smaller than $b(x)$. The space requirements are two additional polynomials of degree $\max(m,n)$, used to store the quotient and remainder. Therefore, the space complexity is $\mathcal{O}(\max(m,n))$.

2.2.2 Precomputed Powers

Another approach to this problem is to precompute a table of values, then do some simple additions and multiplications to obtain a remainder. Suppose we want to compute the product of two polynomials of degree n modulo an irreducible polynomial $I(x)$ with prime p. The product, before being simplified, has maximum degree $2n$. We may compute, in advance, a table of values as follows:

Table 2.1: Precomputed Powers Array Layout

Index	Value
0	$x^0 \bmod I(x)$
1	$x^1 \bmod I(x)$
2	$x^2 \bmod I(x)$
...	...
2n	$x^{2n} \bmod I(x)$

The result of the above computations is a table with the powers of x, up to x^{2n} (or another predetermined stopping point), each of which is simplified as a polynomial of degree less than $I(x)$ in advance. This table may be computed with a small number of long divisions in advance. The following remainder algorithm operates, from this point onward, without the need for long division:

Listing 2.1: Precomputed Powers Pseudocode

```
fn remainder(poly a, poly I, int p, vec<poly> powers) -> poly{
  if deg(a) < deg(I) {
    return a
  }

  let answer = zero polynomial of degree (deg(I) - 1)
  for i in 0 to deg(a) {
    answer += powers[i] * a.coefficient[i]
  }

  reduce coefficients of answer modulo p

  return answer
}
```

First, the algorithm checks if the polynomial $a(x)$ is already a residue class of the field, in which case the polynomial is already in its simplest form. Next, a zeroed out polynomial is created, with enough space to hold any field element. Each iteration of the loop takes the i^{th} power of x from the table of precomputed powers and multiplies it by the i^{th} integer coefficient of $a(x)$. These powers of x are already reduced to the simplest possible field elements, so multiplication by a constant and adding does not grow beyond the degree of the irreducible

$I(x)$ at any point. The last step is to reduce all the coefficients modulo p.

Algorithm Analysis

Let $\deg(a(x)) = m$ and $\deg(I(x)) = n$. Then, this function requires n operations to initialize a polynomial, n operations to reduce the coefficients modulo the prime and $\deg(a(x)) = m$ loop iterations. The interior of the loop is multiplication of a polynomial of degree n by a constant then adding that value to the answer. This requires $2n$ operations. The overall running time of this function is therefore $\mathcal{O}(2nm + 2n) \in \mathcal{O}(nm)$ for any non-trivial polynomial $a(x)$ with $m > 0$. The space requirements for executing the function are $\mathcal{O}(n)$, a polynomial to store the answer. In addition, some memory is required to store the precomputed table. Let r be the maximum precomputed power of x, then this function requires $\mathcal{O}(nr)$ space to store the table. In the case of polynomial multiplication, $r = 2n$, making the additional space requirement $\mathcal{O}(n^2)$.

2.3 Greatest Common Divisors

The computation of the Greatest Common Divisor (GCD) is another fundamental algorithm covered in detail in Knuth's book Seminumerical Algorithms [3] and other books such as Introduction to Algorithms [4]. This algorithm can be applied to both integer values and polynomials in very similar ways.

2.3.1 Integer GCD

If an integer x divides another integer y with remainder of zero, then "x divides y" or "$x|y$". For two integers, call them a and b, the greatest common divisor of a and b, denoted $\gcd(a, b)$, is the largest integer that divides both a and b. Conveniently, the gcd operation is associative, commutative, and distributive

over the integers. In addition, it is always the case that $\gcd(a, 0) = a$ and $\gcd(a, 1) = 1$ [5].

Example

Let $a = 1638$ and $b = 952$ be integers. One way to compute their gcd is using their prime factorizations, which are as follows:

- $a = 2 \times 3 \times 3 \times 7 \times 13$
- $b = 2 \times 2 \times 2 \times 7 \times 17$

Note that both a and b are divisible by exactly 2 and 7, with no other factors in common. From this, it follows that the largest number that can possibly divide both is the product of $2 \times 7 = 14$. Hence, $\gcd(1638, 952) = 14$.

2.3.2 Algorithm to Compute GCD

While it is possible to compute the GCD using the prime factorizations of the two integers, as in the above example, this is very inefficient[2]. A better approach is to use the Euclidean algorithm. The Euclidean algorithm is based on an interesting fact about Euclidean division. Let a and b be two integers, with $a \geq b$. By definition, dividing a by b would result in a quotient and a remainder such that $a = q_0 \times b + r_0$. Clearly, if the remainder is 0 then $\gcd(a, b) = b$ (this follows from the definition of divisibility). The definition of divisibility tells us that if $r_0 = 0$ then $b|a$. Further, $b|b$ and no integer larger than b exists that can also divide b. As a result, b must be the largest possible divisor of both a and b. We can perform this division over and over again, using the divisor and remainder from the previous step, eventually stopping at a remainder of zero. The final non-zero remainder r_n is the greatest common divisor of a and b:

[2]Integer Factorization becomes an intractable problem once the integers are thousands of bits in size; this is the standard size for any cryptographic keys involved in a factoring problem

$a = q_0 \times b + r_0$

$b = q_1 \times r_0 + r_1$

$r_0 = q_2 \times r_1 + r_2$

$r_1 = q_3 \times r_2 + r_3$

...

$r_n = q_{n+2} \times r_{n-1} + 0$

This repeated division causes each successive remainder to decrease, until reaching a remainder of zero. This algorithm relies on the fact that $\gcd(a, b) = \gcd(b, r_0) = \gcd(r_i, r_{i+1})$ all the way until the remainder is zero. This allows for the extraction of the GCD without having to factor either a or b.

Listing 2.2: Recursive GCD Pseudocode

```
fn gcd_recursive(int a, int b) -> int {
  if b = 0 {
    return a
  }
  return gcd_recursive(b, a mod b)
}
```

Listing 2.3: Iterative GCD pseudocode

```
fn gcd_iterative(int a, int b) -> int {
    while b != 0 {
        let r = a mod b
        a, b = b, r
    }
    return a
}
```

Algorithm Analysis

When operating on integers a and b with $n = \max(a, b)$, this algorithm requires $\mathcal{O}(\log n)$ divisions. Effectively, the size of the remaining problem is reduced by 1 bit per iteration. If it is assumed that division on integers has constant cost $\mathcal{O}(1)$, then the algorithm has overall running time $\mathcal{O}(\log n)$. The Euclidean

algorithm applied to integers requires $\mathcal{O}(1)$ additional space to store a couple of integers.

2.3.3 Polynomial GCD

The Euclidean algorithm can be defined in the same way when operating on polynomials. The time complexity and space complexity change to reflect the fact that polynomial operands are used instead of integers.

Example

Let $p = 29$ be the prime used and let $a(x), b(x) \in F_p[x]$:

- $a(x) = x^3 + 18x^2 + 13x + 26 \pmod{29}$
- $b(x) = 10x^3 + 25x^2 + 21x + 2 \pmod{29}$.

Neither of the above polynomials are irreducible and are therefore able to be factored into a product of smaller polynomials. Applying a factoring algorithm produces the following factorizations:

- $a(x) = (x + 10)(x + 9)(x + 28) \pmod{29}$
- $b(x) = (10x + 3)(x + 9)(x + 28) \pmod{29}$

The polynomial factors that divide both $a(x)$ and $b(x)$ are clearly $(x + 9)(x + 28) = (x^2+8x+20)$. This is also the highest degree polynomial that divides both $a(x)$ and $b(x)$ giving $\gcd(a(x), b(x)) = x^2 + 8x + 20$. Note that when computing the gcd of two polynomials in a finite field, the answer is only accurate up to a constant.

2.3.4 Algorithm to Compute GCD

Let $a(x)$ and $b(x)$ be polynomials with $\deg(a(x)) \geq \deg(b(x))$. The Euclidean algorithm applied to polynomials relies on the same ideas as the integer version. First, dividing $a(x)$ by $b(x)$ yields a quotient and remainder such that $a(x) = q_0(x) \times b(x) + r_0(x)$. In addition, it is still the case that $\gcd(a(x), b(x)) = \gcd(b(x), r_0(x)) = \gcd(r_i(x), r_{i+1}(x))$. When operating on polynomials, the degree of the remaining values are reduced by 1 per iteration.

Listing 2.4: Polynomial GCD Pseudocode

```
fn poly_gcd(poly a, poly b, int prime) -> poly {
    while a != 0 {
        let r = remainder(b / a)
        b, a = a, r
    }

    return b
}
```

Algorithm Analysis

Let the two operands be $a(x)$ and $b(x)$ with $m = \deg(a(x))$ and $n = \deg(b(x))$, $m \geq n$. The Euclidean algorithm on polynomials performs $\mathcal{O}(m)$ divisions. The dividends have decreasing degrees $m, (m-1), ..., 1$ under worst case running conditions. Using the schoolbook method of long division, each long division is expected to cost $\mathcal{O}(n(m-n))$. This suggests a total time complexity of $\mathcal{O}(mn(m-n))$ operations for the Euclidean algorithm[3]. On the common problem of $\deg(a(x)) = \deg(b(x))$, this can be simplified to $\mathcal{O}(n^2)$ [5]. The algorithm again requires space to store two polynomials, giving $\mathcal{O}(m)$ space complexity.

[3]Recall that long division when $m = n$ requires one iteration, not zero, due to constants being dropped from Big-O values.

2.4 Inverses

Suppose we have some number, call it x. The inverse of x, denoted x^{-1}, is another number such that $x \times x^{-1} = 1$. In other words, a number multiplied by its inverse gives 1. This idea is applicable to many different types of numbers.

Rational Number Example

Finding inverses in the rational numbers $\mathbb{Q}$ is almost trivial. Suppose $x \in \mathbb{Q}$. By definition $x = a/b$ for some integers a and b, with $b \neq 0$. Provided that x is not 0, its inverse is always the reciprocal $x^{-1} = b/a$. This follows from the following cancellation:

$$x \times x^{-1} = \frac{a}{b} \times \frac{b}{a} = \frac{ab}{ab} = 1 \tag{2.1}$$

For example, let $x = 8/5$. The inverse x^{-1} is the reciprocal $5/8$, which gives:

$$\frac{8}{5} \times \frac{5}{8} = \frac{40}{40} = 1 \tag{2.2}$$

Finite Field Example

Inverses also exist in all finite fields by definition. Consider the simple case of a finite field defined by just the prime 29, GF(29). Some examples of numbers and their inverses are:

- $a = 10$, $a^{-1} = 3$
- $b = 8$, $b^{-1} = 11$
- $c = 25$, $c^{-1} = 7$

These inverses can be quickly verified by multiplying the two values then taking the remainder on division by 29, which should be 1. A more complicated finite

field can be defined by the prime 101 and the irreducible $x^3 + 75x^2 + 55x + 53$. An example of a field element and its inverse are:

- $a(x) = 12x^2 + 4x + 41$
- $a^{-1}(x) = 68x^2 + 63x + 68$

This can again be verified by quickly multiplying $a(x) \times a^{-1}(x)$ and checking the value is equivalent to 1 when divided by the irreducible.

2.4.1 Algorithm for Inverses

Inverses are closely related to the greatest common divisor problem, as they can both be solved with variations of the Euclidean algorithm. Knuth [3] also covers this application of the Euclidean algorithm, called the Extended Euclidean algorithm. This version of the Euclidean algorithm computes not only the GCD but also a solution to the equation $ax + by = gcd$, where a and b are the two inputs to the algorithm. There are multiple approaches to implementing the Extended Euclidean algorithm, but a common approach is to store an intermediate solution to the equation $ax + by = gcd$ on each step. This partial solution is improved every iteration of the algorithm, until the final answer can be found. The following pseudocode is based on Knuth's version of the algorithm. Other versions of this algorithm with the same asymptotic running times exist, such as those presented in Gathen [5].

Listing 2.5: Extended Euclidean Algorithm Pseudocode

```
// Returns a tuple of information (x, y, gcd) for ax + by = gcd
fn extended_euclidean(int a, int b) -> (int, int, int) {
  let u = (1, 0, a)
  let v = (0, 1, b)
  let t = (0, 0, 0)

  while v[3] != 0 {
    let q = floor(u[3] / v[3])

    t = u - vq
    u = v
    v = t
  }

  return u
}
```

The Extended Euclidean algorithm can find inverses with a clever choice of parameters. Suppose we want to find a^{-1} (mod p) using the Extended Euclidean algorithm, for some integer a and prime p. Calling extended_euclidean(a, p) produces a tuple of information (x, y, gcd) such that $ax + py = gcd$. Based on the choice of parameters for the Extended Euclidean algorithm, it is the case that $a^{-1} = x$ because:

- $ax + py \equiv gcd$ (mod p)
- $ax \equiv gcd$ (mod p)

Since p is a prime, having divisors only 1 and itself, it also follows that $\gcd(a, p) = 1$ for all $0 < a < p$ (the non-zero residue classes of the field). Therefore:

- $ax \equiv 1$ (mod p)

And by definition, $x = a^{-1}$ (mod p), the desired inverse.

Algorithm Complexity

The asymptotic running time of the Extended Euclidean algorithm is similar to the standard Euclidean algorithm. When operating on integers, $\mathcal{O}(\log n)$

iterations are required. It would be reasonable to assign each iteration a fixed cost $\mathcal{O}(1)$ for an overall running time of $\mathcal{O}(\log n)$ on integers. This algorithm similarly requires a constant amount of additional space, making the space complexity $\mathcal{O}(1)$ for integers.

For the polynomial version, let the two operands $a(x)$ and $b(x)$ be polynomials with $m = \deg(a(x))$ and $n = \deg(b(x))$, $m \geq n$. The polynomial version again executes $\mathcal{O}(m)$ iterations. Each iteration requires: a division, a multiplication and a subtraction. Let D be the cost of a division, and M be the cost of a multiplication. The cost per iteration is, in general, $\mathcal{O}(D + M)$, giving an overall running time of $\mathcal{O}(m(D+M))$. An exact running time can be obtained by substituting the cost of division and multiplication into the algorithm. The space requirements are several polynomials of degree m, making the space complexity $\mathcal{O}(m)$ for the polynomial version of the algorithm.

2.5 Powers in Finite Fields (Powmod)

Computing powers is a key operation for many algorithms that use finite fields. This operation is also sometimes referred to as the Powmod function or fast exponentiation over a finite field. This problem involves efficiently computing $a^b \pmod p$ in a finite field, where a is a field element, b is an integer, and p is a prime. There are several approaches to this problem, with varying performance. The most obvious approach, and incidentally one of the worst, is to naively perform b multiplications and then mod by p at the very end. This approach requires b multiplications and will result in a very, very large intermediate value. This approach looks like this when represented in pseudocode:

Listing 2.6: Bad Powmod Pseudocode

```
fn powmod(int a, int b, int prime) -> int {
  let answer = 1
  for i in 1 to b {
    answer *= a
  }
  answer = answer mod prime
  return answer
}
```

There are two improvements to this approach that make the computation of powers into a much faster operation. The first of these improvements is to mod the intermediate result after every multiplication. This means that the intermediate result will be of size of most $2p$. The second improvement is to apply the method of repeated squaring to greatly reduce the number of multiplications required [6].

Squaring is a relatively efficient operation, requiring only a single multiplication of a number by itself. Suppose we want to compute the simple power, 3^2. This can be evaluated directly as 3×3 using a single multiplication. Next, consider evaluating the more complicated 3^4. The first way to evaluate 3^4 is the obvious way using three multiplications: $3^4 = 3 \times 3 \times 3 \times 3$. However, note that $3^4 = (3 \times 3)^2 = ((3^2)^2)$. Therefore, we can evaluate 3^4 by squaring twice, requiring only two multiplications to achieve the same result. This extends to higher powers such as $3^8 = (((3^2)^2)^2)$ and $3^{16} = ((((3^2)^2)^2)^2)$. Evaluating 3^8 this way requires only three multiplications, instead of seven. Evaluating 3^{16} requires only four multiplications instead of 15. The savings increase further as the power gets larger.

Expression	Naive Approach	Written as Squares
3^2	3×3	3^2
3^4	$3 \times 3 \times 3 \times 3$	$((3^2)^2)$
3^8	$3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$	$(((3^2)^2)^2)$
3^{16}	....	$(((3^2)^2)^2)^2$

This approach based on squaring can be converted almost directly into a divide

and conquer algorithm that cuts the problem size in half every single iteration. To compute 3^{16}, we compute 3^8 and multiply that value by itself. Computing 3^8 can be done by computing 3^4 and multiplying 3^4 by itself. 3^4 is 3^2 multiplied by itself. 3^2 is 3^1 times itself. Finally, we know that 3 is equal to itself, giving us the information we need to go back and solve for all the previous values. Like many other divide and conquer algorithms, this algorithm is naturally recursive. The improved modular power algorithm expressed as a recursive function looks like this:

Listing 2.7: Good Powmod Pseudocode

```
fn powmod(int a, int b, int prime) -> int {
    if b = 0 {
        return 1
    }

    let half = powmod(a, floor(b / 2), prime)
    if b is odd {
        return (a * half * half) mod prime
    } else b is even {
        return (half * half) mod prime
    }
}
```

The correction for odd exponents is very easy and straightforward. As an example, consider $3^9 = 3^4 \times 3^4 \times 3$. We can still divide the problem in half to solve it, we just have to multiply by the value one extra time after computing the subproblem. This also works for numbers that are not adjacent to a power of two, such as $3^{13} = 3^6 \times 3^6 \times 3$. The result can still be corrected by multiplying by the value one extra time.

Algorithm Analysis

These adjustments to the algorithm cause the number of multiplications required to drop dramatically. An example of this dramatic decrease can be seen even for powers as a low as 100:

Table 2.2: Multiplies Required for Powers

n	Naive Multiplies	Repeated Square Multiplies
2	1	1
4	3	2
8	7	3
16	15	4
32	31	5
64	63	6
128	127	7
256	255	8
512	511	9
1,024	1,023	10

Based on the fact that every iteration of this algorithm cuts the problem in half, and does constant work, the problem has a recurrence of $T(b) = T(b/2) + 1$. This recurrence solves to $\Theta(\log b)$ for the time complexity. Every recursion step requires a fixed amount of space as well, so the space complexity is also $\Theta(\log b)$. This algorithm requires the same number of steps when operating on polynomials, although the cost of multiplication and division are higher.

2.6 Polynomial Addition

The addition of two polynomials is a coefficient by coefficient addition of the two operands. Let $a(x)$ and $b(x)$ be the two polynomials being added together, then the result of the addition has degree $\max(\deg(a(x)), \deg(b(x)))$.

Example

Consider the following two polynomials:

- $a(x) = 14x^4 + 6x^2 + 19x + 25$
- $b(x) = x^3 + 26x^2 + 28x + 1$

Their addition, given by adding together each coefficient with the same power of x, is:

- $14x^4 + x^3 + 32x^2 + 47x + 26$

If this operation is in a finite field, the only difference is the coefficients need to be reduced modulo the prime. Suppose $p = 29$ is the prime for the finite field. Then, the answer produced from $a(x) + b(x)$ is the following after reducing all the coefficients:

- $14x^4 + x^3 + 3x^2 + 18x + 26$

Algorithm Analysis

This algorithm expressed in pseudocode is the following:

Listing 2.8: Polynomial Addition Pseudocode

```
fn poly_add(poly a, poly b, int prime) -> poly {
  let answer = zero polynomial of degree max(deg(a), deg(b))

  for i in 0 to deg(a) {
    answer[i] = a[i]
  }
  for i in 0 to deg(b) {
    answer[i] += b[i]
  }

  mod coefficients in the answer poly by prime

  return answer
}
```

Let $a(x), b(x) \in F_p[x]$ be the two polynomials used in this algorithm, with $n = \max(\deg(a(x)), \deg(b(x)))$. This algorithm executes at worst $3n$ loop iterations to perform the addition, making the time complexity $\mathcal{O}(n)$. Similarly, if the answer is to be stored in a new polynomial, this requires n words of memory, making the space complexity also $\mathcal{O}(n)$.

2.7 Polynomial Multiplication

There are several different approaches to polynomial multiplication. Three of the most important types of polynomial multiplication algorithms are the following: the naive (schoolbook) algorithm, the Karatsuba algorithm, and Fast Fourier Transformation (FFT) algorithms.

2.7.1 Naive Polynomial Multiplication

This approach to multiplication involves multiplying every term in the first polynomial by every term in the second polynomial, then collecting all the coefficients with the same power of x together [7]. It is sometimes refered to as the "schoolbook method" because it is often taught in schools when polynomials are introduced. Due to its simplicity, this method is easy for humans to apply, unlike some of the other multiplication algorithms.

Example

Consider the following polynomials:

- $a(x) = 10x^2 + 6x + 5$
- $b(x) = x + 2$

Using the naive method, $a(x) \times b(x)$ would involve computing every possible term (there are $3 \times 2 = 6$ terms in this case):

- $a(x) \times b(x) = 10x^3 + 6x^2 + 5x + 20x^2 + 12x + 10$

Then, terms with the same power of x are collected together:

- $= 10x^3 + (20 + 6)x^2 + (5 + 12)x + 10$
- $= 10x^3 + 26x^2 + 17x + 10$

Algorithm Complexity

This algorithm can be expressed with the following pseudocode:

Listing 2.9: Naive Multiplication Pseudocode

```
fn naive_multiply(poly a, poly b) -> poly {
  let answer = zero polynomial of degree (deg(a) + deg(b))

  for i in 0 to deg(a) {
    for j in 0 to deg(b) {
      answer[i + j] += a[i] * b[j]
    }
  }

  return answer
}
```

Algorithm Analysis

Suppose the two polynomials being added are $a(x)$ and $b(x)$, with $m = \deg(a(x))$ and $n = \deg(b(x))$. It follows directly from the nested loops in the pseudocode that this algorithm requires $n \times m$ steps, making the time complexity $\Theta(nm)$. In the case that $n = m$, the running time of this algorithm can be simplified to $\Theta(n^2)$. The result of multiplying two polynomials of degree n and m is a polynomial of degree $n + m$. This results in a space complexity of $\Theta(n + m)$.

2.7.2 Karatsuba Multiplication

Karatsuba multiplication is more complicated than the naive algorithm but has a much better running time in return [8]. This algorithm is recursive in nature, so it makes sense to start by defining the base case of the algorithm. Consider the case of multiplying two linear polynomials $a(x)$ and $b(x)$ using the naive method:

- $a(x) = 6x + 2$
- $b(x) = 3x + 7$

which have the following product:

- $a(x) \times b(x) = (6x + 2)(3x + 7) = 18x^2 + 42x + 6x + 14 = 18x^2 + 48x + 14$

Assuming that this operation is not performed in a finite field, this always requires exactly 4 multiplications and 1 addition. The Karatsuba approach is based on an interesting observation that this is not the only way to multiply together two linear polynomials [9]. Consider the same two linears as above:

- $a(x) = 6x + 2 = a_1 x + a_0$
- $b(x) = 3x + 7 = b_1 x + b_0$

These polynomials can be multiplied together in a different way that gives the same answer. First, compute the following three temporary values:

- $d_0 = a_0 \times b_0 = 2 \times 7 = 14$
- $d_1 = a_1 \times b_1 = 6 \times 3 = 18$
- $d_{01} = (a_0 + a_1)(b_0 + b_1) = (6 + 2)(3 + 7) = 8 \times 10 = 80$

Combine the above d_0, d_1, d_{01} in the following way:

- $a(x)b(x) = d_1 x^2 + (d_{01} - d_0 - d_1)x + d_0$
- $a(x)b(x) = 18x^2 + (80 - 14 - 18)x + 14$
- $a(x)b(x) = 18x^2 + 48x + 14$

The pseudocode for this is relatively straightforward:

Listing 2.10: Karatsuba Linear Pseudocode

```
fn multiply_linears(poly a, poly b) -> poly {
  let d0 = a[0] * b[0]
  let d1 = a[1] * b[1]
  let d01 = (a[0] + a[1]) * (b[0] + b[1])

  return (d1)x^2 + (d01 - d0 - d1)x + d0
}
```

Upon inspection, this approach requires exactly 3 multiplications and 4 additions/subtractions. This means that the Karatsuba approach requires one fewer multiplications than the naive approach but requires three more additions/subtractions. Therefore, if the cost of multiplication is at least three times that of addition/subtraction, the Karatsuba approach is faster. This approach for multiplying linears can be recursively applied to polynomials of any degree with minor changes. The recursive approach relies on the fact that any polynomial can be divided into a higher and lower part almost trivially. Consider the following polynomial $p(x)$:

- $p(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$

This can be rewritten as the following:

- $p(x) = (a_0 + a_1x + ... + a_{\lfloor n/2 \rfloor}x^{\lfloor n/2 \rfloor}) +$
  $(a_{\lfloor n/2 \rfloor+1} + a_{\lfloor n/2 \rfloor+2}x + ... + a_nx^{\lfloor (n-1)/2 \rfloor})x^{\lfloor n/2 \rfloor+1}$

This polynomial is now effectively in the form $p(x) = p_0(x) + p_1(x)x^{\lfloor n/2 \rfloor+1}$, which looks a lot like the case of multiplying two linears. The main differences are the that coefficients are now polynomials and the x is replaced with an $x^{\lfloor n/2 \rfloor+1}$. Now the same approach used for multiplying linears can be applied using the $p_0(x)$ and $p_1(x)$ parts. Consider the following polynomials:

- $p(x) = 6x^3 + 2x^2 + 8x + 7 = (6x + 2)x^2 + (8x + 7) = p_1(x)x^2 + p_0(x)$
- $q(x) = 3x^3 + x^2 + 9x + 5 = (3x + 1)x^2 + (9x + 5) = q_1(x)x^2 + q_0(x)$

From this, compute d_0, d_1, d_{01} as before:

- $d_0 = p_0(x) \times q_0(x) = (8x + 7)(9x + 5) = 72x^2 + 103x + 35$
- $d_1 = p_1(x) \times q_1(x) = (6x + 2)(3x + 1) = 18x^2 + 12x + 2$
- $d_{01} = (p_0(x) + p_1(x)) \times (q_0(x) + q_1(x)) = ((6x + 2) + (8x + 7)) \times ((3x + 1) + (9x + 5)) = 168x^2 + 192x + 54$

Finally, finish the computation by combining the different parts:

- $p(x)q(x) = (d_1)x^n + (d_{01} - d_0 - d_1)x^{\lfloor n/2 \rfloor + 1} + d_0$
- $p(x)q(x) = (18x^2 + 12x + 2)x^4 + ((168x^2 + 192x + 54) - (72x^2 + 103x + 35) - (18x^2 + 12x + 2))x^2 + (72x^2 + 103x + 35)$
- $p(x)q(x) = 18x^6 + 12x^5 + 80x^4 + 77x^3 + 89x^2 + 103x + 35$

This step yields the final answer to the multiplication. This algorithm can be expressed recursively as follows [9][page 6-13] [10] for two polynomials of the same degree:

Listing 2.11: Karatsuba Multiplication Pseudocode

```
fn karatsuba(poly a, poly b) -> poly {

  let n = max(deg(a), deg(b)) + 1

  // Somehow split a(x) and b(x) into halves such that:
  // let a = d(x) + e(x)x^⌊n/2⌋+1
  // let b = g(x) + h(x)x^⌊n/2⌋+1

  if n = 1 {
    return a * b; // multiply constants
  }

  let t1 = d(x) + e(x)
  let t2 = g(x) + h(x)

  let d0 = karatsuba(d(x), g(x))
  let d1 = karatsuba(e(x), h(x))
  let d01 = karatsuba(t1, t2)

  return (d1)x^n + (d01 - d0 - d1)x^⌊n/2⌋+1 + d0
}
```

Algorithm Analysis

Although not immediately obvious, this approach is significantly better than the naive multiplication algorithm. One way to compare the two algorithms is using a table with the number of additions and multiplications required for both Karatsuba and naive multiplication [9][page 6]. Let n be the number of coefficients in the polynomials being multiplied i.e. $n = 2$ is a linear polynomial and $n = 4$ is a cubic. Then the naive algorithm requires n^2 multiplications and $(n-1)^2$ additions [7]. The Karatsuba approach requires $3T(n/2)$ multiplications and $3T(n/2)+6n$ additions, where $T(n/2)$ is the number of operations required to solve a subproblem of half the size.

Table 2.3: Karatsuba Operations Required

n	Karatsuba Mults	Karatsuba Adds	Naive Mults	Naive Adds
2	3	4	4	1
4	9	36	16	9
8	27	156	64	49
16	81	564	256	225
32	243	1,884	1,024	961
64	729	6,036	4,096	3,969
128	2,187	18,876	16,384	16,129
256	6,561	58,164	65,536	65,025
512	19,683	177,564	262,144	261,121
1024	59,049	538,836	1,048,576	1,046,529

The above recurrences can be solved to get the final running time. The recurrence before simplification is $T(n) = 3T(n/2) + \mathcal{O}(n)$ based on the fact the algorithm solves three subproblems of size $1/2$ with n additional work to combine the results [11]. This makes the overall time complexity of the algorithm $\mathcal{O}(n^{log_2(3)}) \approx \mathcal{O}(n^{1.585})$ after the recurrence is solved. This is a substantial reduction in the overall running time when compared to the naive algorithm's running time of $\mathcal{O}(n^2)$. The Karatsuba algorithm also requires additional memory to run. Every recursion step of the algorithm allocates $\mathcal{O}(n)$ memory, where n is the size of the subproblem. The problem is cut in half each recursion step, meaning that the overall problem sizes are as follows: n, $\frac{n}{2}$, $\frac{n}{4}$, ... , 2, 1. When summed together, $n + \frac{n}{2} + \frac{n}{4} + ... + 2 + 1 \approx 2n$. From this we can see that the algorithm also requires $\mathcal{O}(n)$ additional memory, similar to the naive algorithm.

2.7.3 Fast Fourier Transformations

Fast Fourier Transformations solve for the product using interpolation. Consider the polynomial $a(x) = 3x^2 + 7x + 4$. One of the core ideas behind interpolation is that this polynomial can also be uniquely identified using $(\deg(a(x)) + 1)$ distinct points [12][page 141]. This is sometimes referred to as the pointwise form of a polynomial. Creating a table of x and y values is relatively easy to do.

Simply take some distinct values of x and substitute them into the polynomial to get the corresponding y values. An example of a set of points that uniquely identify $a(x)$ are:

x	y
0	4
1	14
2	30

Given a set of points, it is also possible to find the corresponding polynomial. There are several possible ways to do this using algorithms such as Lagrange Interpolation or Newton's Divided Differences. A quick example using the basic formula for Lagrange interpolation [12][page 140] is as follows. Note that (x_1, y_1) is the first point in the table above, (x_2, y_2) the second, and (x_3, y_3) is the third:

- $P(x) = y_1 \times \frac{(x-x_2)(x-x_3)}{(x_1-x_2)(x_1-x_3)} + y_2 \times \frac{(x-x_1)(x-x_3)}{(x_2-x_1)(x_2-x_3)} + y_3 \times \frac{(x-x_1)(x-x_2)}{(x_3-x_1)(x_3-x_2)}$

Then substitute the points into the formula and simplify the result:

- $P(x) = 4 \times \frac{(x-1)(x-2)}{(0-1)(0-2)} + 14 \times \frac{(x-0)(x-2)}{(1-0)(1-2)} + 30 \times \frac{(x-0)(x-1)}{(2-0)(2-1)}$
- $P(x) = 2(x-1)(x-2) + (-14)(x)(x-2) + 15(x)(x-1)$
- $P(x) = 3x^2 + 7x + 4$

The idea behind this formula for Lagrange interpolation is that substituting the first point (x_1, y_1) into the equation results in a large amount of cancellation. The first term turns into $y_1 \times \frac{1}{1}$ after substituting x_1. The rest of the terms all go to zero. Thus, substituting x_1 into the equation gives back y_1. The same happens for the rest of the points, meaning this equation will always yield y_n when x_n is substituted into it. This can be expanded to work with an arbitrary number of points, although this approach is not necessarily the most efficient way to solve the problem.

Finally, two polynomials in pointwise form can be multiplied together. Multiplying two polynomials in pointwise form is equivalent to multiplying them

normally [13][page 137]. The only problem with multiplying in pointwise form is that it may need to be converted back to the standard representation in order to be useful. Consider $a(x)$ and $b(x)$, two polynomials of degree n, defined as follows:

- Let $a(x)$ be represented by (n+1) points $(x_0, a(x_0)), ..., (x_n, a(x_n))$
- Let $b(x)$ be represented by (n+1) points $(x_0, b(x_0)), ..., (x_n, b(x_n))$

The pointwise form of the product is equivalent to a termwise multiplication of the two polynomials:

- $a(x) \times b(x) = (x_0, a(x_0) \times b(x_0)), ..., (x_n, a(x_n) \times b(x_n))$

Thus, we can convert two polynomials into pointwise form, multiply them together, and convert the pointwise form of the answer back to a polynomial. This process yields an equivalent answer to performing the multiplication normally. This is the core idea behind Fast Fourier Transformations. The overall algorithm, summarized briefly, is therefore [14]:

1. Convert the polynomials to pointwise forms
2. Multiply the pointwise forms of the two polynomials
3. Interpolate to get the polynomial form of the answer

Before developing the full algorithm, we should briefly consider the running time of this basic solution to the problem. Let n be the degree of the two polynomial operands used in the multiplication. The first step, generating $n + 1$ points, takes $\mathcal{O}(n^2)$ operations when naively substituting values into the polynomial. Substituting a single value into a polynomial of degree n requires a total of $\mathcal{O}(n)$ additions/subtractions/multiplications. Repeated $n + 1$ times, once per point generated, is therefore expected to take $\mathcal{O}(n^2)$ time. The term-by-term multiplication requires $\mathcal{O}(n)$ operations, one multiplication for every point. Finally,

Lagrange interpolation using the above formula requires $\mathcal{O}(n^2)$ operations. In total, this is $\mathcal{O}(n^2)$ for the entire algorithm. This can be reduced to $\mathcal{O}(n \log n)$ using better algorithms.

2.7.4 A Better Approach

There are two parts of this algorithm that need to be improved in order to reduce the overall running time: converting from a polynomial to the pointwise form and converting from pointwise form to a polynomial. This explanation is based on content in [15] and [4].

One of the easiest approaches to optimizing this problem involves the complex numbers. The complex numbers are the real numbers with i adjoined, where $i^2 = -1$. A common way to write the complex numbers is in the form $a + bi$, where a and b are real numbers. However, another way to represent complex numbers is with polar notation. When representing a complex number in polar form, we instead use a radius r and angle θ, in radians. Complex numbers in polar form can be denoted as $re^{i\theta}$. The advantage of this form is that multiplication is very fast: multiply the two radii together to get the new radius and add the two angles together to get the new angle. A complex number in polar form can be converted back to the standard form by noting that $e^{i\theta} = cos(\theta) + isin(\theta)$.

With this brief definition of the complex numbers, it is now possible to define an n^{th} root of unity. In this context, unity means 1. So an n^{th} root of unity is an n^{th} root of 1; a number that can be multiplied together n times to give a final answer of 1. When working with just the real numbers there exist two trivial solutions to this problem: -1 and +1. The complex numbers allow for non-trivial solutions to the problem. Polar notation makes it particularly easy to find an n^{th} root of unity.

The first step in finding an n^{th} root of unity using polar notation is to set the radius to $r = 1$. Recall that multiplying in polar notation can be done by multiplying the two radii and adding the two angles. If the radius is 1, then squaring will also result in a radius of 1. Any number of these multiplications will always result in a radius of 1 at the end. This solves half of the problem (the radius). The angle part is also relatively straightforward. For this complex number to be an n^{th} root, it must be that $n \times \theta$ is a multiple of 2π. I.e. after performing n additions, the result is a multiple of 2π, an angle equivalent to zero on the unit circle. An obvious solution to this is $\theta = \frac{2\pi}{n}$, which gives 2π after being added together n times. Interestingly, this solution is not unique, and will work for any $\theta = \frac{2\pi k}{n}$, $k = 0, 1, ..., (n-1)$.

Converting a polynomial to pointwise form can be done in $\mathcal{O}(n \log n)$ time using roots of unity. Consider the polynomial $f(x) = a_0 + a_1 x + ... + a_n x^n$. Originally, we converted this to pointwise form by substituting a value into the polynomial and computing the result. However, another approach is to divide the polynomial into two halves - even coefficients and odd coefficients. The result are two polynomials:

- $f_{even}(x) = a_0 + a_2 x + a_4 x^2 + ...$
- $f_{odd}(x) = a_1 + a_3 x + a_5 x^2 + ...$

From this it follows that:

- $f(x) = f_{even}(x^2) + x \times f_{odd}(x^2)$

The reason for using this form is that $f(-x)$ can be computed almost trivially by noting that:

- $f(-x) = f_{even}(x^2) - x \times f_{odd}(x^2)$

This clearly follows from the fact that x^2 is always even, and is therefore unaffected by whether x is positive or negative. This optimization allows for the

computation of two points at once using the same information. The two halves, f_{even} and f_{odd} are also valid polynomials. Therefore, this process can be applied recursively to each half. Effectively, this changes the amount of work required to $2T(\frac{n}{2})$ with $\mathcal{O}(n)$ work to combine the pieces afterwards, for an overall recurrence of $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$. This recurrence solves to $\mathcal{O}(n \log n)$, exactly the running time required.

Listing 2.12: Pseudocode to Find the Pointwise Form

```
// vec<complex> a - a vector of the polynomial coefficients as
//        complex numbers. The result is the
//        pointwise form of this polynomial.
// m - number of points to generate
// w - a mth root of unity
// returns an array of complex points
fn fft_pointform(vec<complex> a, int m,
     complex w) -> vec<complex> {
  if m = 1 {
    return new vector[a[0]]
  }

  let evens = a[0], a[2], a[4], ... as a vector
  let odds = a[1], a[3], a[5], ... as a vector

  // Compute the two halves recursively
  let even_half = fft_pointform(evens, m / 2, w * w)
  let odd_half = fft_pointform(odds, m / 2, w * w)

  // Combine the two halves
  // The ordering of the resulting vector is
  // [f(w), f(w^2), f(w^3), ..., f(-w), f(-w^2), f(-w^3), ...]
  let f = vector of m complex numbers, initialized to 0 + 0i
  let x = complex number 1 + 0i

  for i in 0..(m / 2) {
    f[j] = even_half[j] + x * odd_half[j]
    f[j + m/2] = even_half[j] - x * odd_half[j]
    x = x * w
  }

  return f
}
```

The last piece missing is being able to interpolate (convert from pointwise form to polynomial form) in $\mathcal{O}(n \log n)$. This turns out to be almost trival using the

fft_pointform algorithm developed above. Let w be an m^{th} root of unity. Then, the algorithm can be reversed by using w^{m-1}, the inverse of an m^{th} root of unity. This inverse can be found by raising w to the $(m-1)^{th}$ power. The result of the fft_pointform algorithm, when using an inverse, is an array of complex numbers. Each entry is n times larger than it should be and the complex parts are all approximately zero. The final step is therefore to divide every entry by n and discard the complex parts. The result of this computation is the polynomial resulting from the multiplication of a and b. This requires 3 applications of an $\mathcal{O}(n \log n)$ algorithm and $\mathcal{O}(n)$ additional work, for a final answer of $\mathcal{O}(n \log n)$ work to multiply two polynomials.

Listing 2.13: Pseudocode for FFT Multiplication

```
fn fft_multiply(poly a, poly b) -> poly {
  let polysize = max(deg(a), deg(b)) * 2
  let m = next_power_of_2(polysize)

  let w = mth_root_of_unity(m)

  // Convert to pointwise form
  let points_a = fft_pointform(a)
  let points_b = fft_pointform(b)

  // Pointwise multiply
  let points_c = new vector
  for i in 0 to polysize {
    points_c.push(points_a[i] * points_b[i])
  }

  // Interpolation
  let w_inverse = pow(w, m - 1)
  let answer_coeffs = fft_pointform(points_c, m, w_inverse)
  // Scale by (1/n)
  for i in 0 to answer_coeffs.size() {
    answer_coeffs[i] = answer_coeffs[i] * (1 / n)
    answer_coeffs[i].imaginary = 0
  }

  return answer_coeffs as poly
}
```

2.8 Factoring Algorithms

The main problem explored in this section is that of factoring a univariate polynomial with coefficients in a Galois Field GF(p), p a prime. These polynomials are consequently all elements of $F_p[x]$. This problem is not equivalent to ordinary factoring.

Example

Consider the following polynomial that is ***not*** in a finite field:

- $f(x) = x^3 + 100x^2 + 72x + 2$

This polynomial is irreducible outside of a finite field. That is, there do not exist any polynomials of a lesser degree that divide $f(x)$ with remainder of zero. However, it is factorable over various finite fields. Suppose all the coefficients are modulo the prime $p = 101$. This problem has a solution:

- $f(x) = x^3 + 100x^2 + 72x + 2 \equiv (x + 10)(x + 40)(x + 50) \pmod{101}$

This factoring problem also has another, different, solution using the prime $p = 107$:

- $f(x) = x^3 + 100x^2 + 72x + 2 \equiv (x + 99)(x^2 + x + 80) \pmod{107}$

Certain polynomials are still irreducible, even in a finite field. Trying to factor $f(x)$ has already yielded a non-trivial example: $(x^2 + x + 80) \pmod{107}$. There is no way to factor this into a product of linears.

2.8.1 Squarefree Polynomials

The first step of many polynomial factorization algorithms is to ensure the input has no square factors. A square factor is simply a factor that repeats itself. An example is $(x + 1)^2$ present in the following factorization:

- $f(x) = (x+1)^2(x+10)(x+3)$

Some of the factorization algorithms can have problems if square factors happen to be present. In such cases, they will explicitly state that the input is to be a square-free polynomial. Regardless of whether or not the algorithm explicitly requires this, any square factors are themselves part of the factorization. As the process of detecting square factors is quick and easy, this can be a way to find a partial factorization before even applying the full algorithm. The algorithm to find square factors in a polynomial is the following [16] [2]:

1. Compute $h(x) = \gcd(f(x), f'(x))$
2. If $\deg(h(x)) = 0$ then STOP
3. Set $f(x) = \frac{f(x)}{h(x)}$
4. REPEAT with the new value of $f(x)$

The value of $f(x)$ when the algorithm eventually stops will contain no square factors.

Example

Consider the following polynomial:

- $f(x) = (x+2)^3(x+7)^2(x+9)(x+11) = x^7+11x^6+6x^5+14x^4+8x^3+22x^2+6 \pmod{29}$

Applying the above algorithm results in the following series of computations:

1. Compute $f'(x) = 7x^6 + 8x^5 + x^4 + 27x^3 + 24x^2 + 15x \pmod{29}$
2. Compute $h(x) = \gcd(f, f') = x^3 + 11x^2 + 3x + 28 = (x+2)^2(x+7) \pmod{29}$
3. Compute $f(x) = f(x)/h(x) = x^4 + 3x^2 + 11x + 23 \pmod{29}$

4. Repeat the loop to make sure $f(x)$ is now squarefree

5. Compute $f'(x) = 4x^3 + 6x + 11 \pmod{29}$

6. Compute $h(x) = \gcd(f, f') = 1 \pmod{29}$

7. $f(x)$ is squarefree. Stop.

2.8.2 Naive Factoring Methods

The naive method of factoring is a crude, brute force attempt to solve the problem. This method is sometimes taught to students in math classes where factoring polynomials larger than a quadratic is required, due to its relative simplicity. Given a polynomial $f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$, this means blindly trying all possible factors and hopefully finding one that works. There are two different brute force approaches, neither of which are particularly good at solving any large-sized problem.

Approach one is to substitute all the numbers from 0 to $(p - 1)$ into the polynomial $f(x)$ and see if the answer to the substitution is zero. If setting $x = c$ makes $f(x)$ zero, then $(x - c)$ is a factor of $f(x)$. Suppose $f(x)$ is the same as the initial factoring problem:

- $f(x) = x^3 + 100x^2 + 72x + 2 = (x + 10)(x + 40)(x + 50) \pmod{101}$

Based on this, we can see that one of the solutions would involve substituting $x \equiv -10 \equiv 91 \pmod{101}$, giving zero:

- $f(91) \equiv (91)^3 + 100(91)^2 + 72(91) + 2 \equiv 1588225 \equiv 0 \pmod{101}$

A weakness of this approach is that it can only find linear factors. It cannot find quadratic, cubic, or higher degree irreducible factors.

Approach two involves trial division by potential factors. It is possible to check if a polynomial has any linear factors by attempting to divide by every single

linear $x, (x+1), (x+2), ..., (x+(p-1))$. If any of these polynomials are factors, they will divide $f(x)$ with remainder zero. This can be extended to quadratic factors by attempting to divide by all quadratics (ax^2+bx+c), then all cubics, and so on. This method is also not efficient because it requires a large number of division operations.

Algorithm Analysis

The first method has a worst case running time of $\mathcal{O}(p)$ substitution operations, whereas the second method requires $\mathcal{O}(p^n)$ division operations to check every possible factor. In order to feasibly factor any polynomial of degree larger than 5, better algorithms are required.

2.8.3 Berlekamp's Algorithm for Polynomial Factorization

One of the original approaches to factoring in a finite field was Berlekamp's algorithm. Berlekamp's algorithm factors a polynomial over GF(p), p a prime, much faster than the naive approach. The basic details of this algorithm are taken from Berlekamp's original publication [17] as well as Divasón et al. [16]. Berlekamp's algorithm assumes that the input is a squarefree polynomial $f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ...a_nx^n$ with all coefficients a_i reduced modulo p. The squarefree algorithm specified above can be used to remove any square factors from the input polynomial in advance. It is assumed henceforth that the input $f(x)$ being factored is squarefree and $0 \leq a_i < p$, for all coefficients a_i.

The first step of Berlekamp's algorithm is the construction of a n by n matrix, with each matrix element reduced modulo p. The columns of this matrix are based on the coefficients of a series of polynomials $x^{ip} \pmod{f(x)}$. Each column vector has the coefficients organized as $[a_n, ...a_1, a_0]$ going from the top to the bottom. When computing the left row, set $i = (n-1)$. For every row to the

right, the value of i decreases by 1, until it reaches $i = 0$ in the right column. This matrix is known as Q in the rest of the algorithm.

The second step of Berlekamp's algorithm is the computation of the null space of $(Q - I)$. The null space can be found using Gaussian Elimination on $(Q - I)$ until the matrix is in row echelon form.

The third step of Berlekamp's algorithm is the computation of the free and restricted variables of the null space matrix. To find the free and restricted variables, highlight the first non-zero entry in each row. The columns that have a highlighted value are the restricted variables. The columns with nothing highlighted are the free variables.

Fourth, using these free and restricted variables, solve for the basis vectors. This is done by slightly modifying the row echelon form of $(Q - I)$. The goal is to solve a set of equations in n variables, so every zero row has a single 1 entry in one of the columns marked as a free variable. Each free variable column is also given a single 1 entry. This matrix can now be used to solve a system of equations. Call this modified $(Q - I)$ matrix M.

Fifth, for every free variable, solve the system $Mv = x$. The vector x is determined by which free variable is used. If x_3 is the free variable selected, then the vector would be $[0, 0, 1, 0, 0]$. In effect, x is a zero vector with a single 1 at the index i, repeated for each of the free variables x_i. The resulting set of v vectors is the set of basis vectors.

Finally, using the set of basis vectors, construct a set of polynomials $g(x) \in G$. Each of these polynomials use as coefficients the basis vectors. With these polynomials, computate $\gcd(f(x), g(x) - s)$, for all $0 \leq s < p$. One or more of these gcd computations is guaranteed to yield a factor if one exists.

The G polynomials derived from the basis vectors computed in the Berlekamp algorithm satisfy the property that $g(x)^p \equiv g(x) \bmod f(x)$, a similar idea to

Fermat's Little Theorem[4]. This means that the G polynomials form a subalgebra, known as the Berlekamp Subalgebra. Using this information, it can be shown that:

- $f(x) = \prod_{s \in GF(p)} \gcd(f(x), g(x) - s)$

This is why the computations in the last step of the Berlekamp algorithm yield a useful answer. The series of GCD computations, $\gcd(f(x), g(x) - s)$ with $s \in GF(p)$, are searching for one of the s values that yield a non-trivial factor. Not every s is guaranteed to yield a non-trivial factor, but eventually one of the s values will work.

Example

Consider the problem of trying to factor the following squarefree polynomial:

- $f(x) = 10 + 5x^1 + 1x^2 + 2x^3 + 5x^4 + 1x^5 \pmod{101}$

Computing the f^{ip} polynomials gives the following five polynomials:

i	$x^{ip} \bmod f(x)$
4	$94 + 55x + 47x^2 + 37x^3 + 40x^4$
3	$50 + 14x + 92x^2 + 79x^3 + 41x^4$
2	$26 + 32x + 10x^2 + 31x^3 + 54x^4$
1	$14 + 74x + 21x^2 + 94x^3 + 34x^4$
0	1

These are converted into the following Q matrix, where each polynomial becomes a column:

$$\begin{vmatrix} 40 & 41 & 54 & 34 & 0 \\ 37 & 79 & 31 & 94 & 0 \\ 47 & 92 & 10 & 21 & 0 \\ 55 & 14 & 32 & 74 & 0 \\ 94 & 50 & 26 & 14 & 1 \end{vmatrix}$$

[4]Fermat's Little Theorem says if p is a prime then $a^p \equiv a \pmod{p}$, for any a

Next compute $(Q - I)$:

$$\left|\begin{array}{ccccc} 39 & 41 & 54 & 34 & 0 \\ 37 & 78 & 31 & 94 & 0 \\ 47 & 92 & 9 & 21 & 0 \\ 55 & 14 & 32 & 73 & 0 \\ 94 & 50 & 26 & 14 & 0 \end{array}\right|$$

Using $(Q - I)$, compute the null space matrix (possibly using Gaussian elimination):

$$\left|\begin{array}{ccccc} 1 & 0 & 82 & 1 & 0 \\ 0 & 1 & 12 & 59 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{array}\right|$$

The leading ones in each column are highlighted in yellow. This indicates that the first two variables are restricted and the last three variables are free variables. That is, x_1, x_2 are restricted variables and x_3, x_4, x_5 are free variables. With this, we slightly modify the $(Q - I)$ null space matrix by adding a 1 to each zero row, and free variable column. This gives the M matrix:

$$\left|\begin{array}{ccccc} 1 & 0 & 82 & 1 & 0 \\ 0 & 1 & 12 & 59 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right|$$

The bottom three rows may be freely swapped, so the exact placement of the ones can vary a little bit. Using this matrix, solve the system of equations $Mv = x$. The x vectors in this case are:

free variable	x vector
x_3	$[0, 0, 1, 0, 0]$
x_4	$[0, 0, 0, 1, 0]$
x_5	$[0, 0, 0, 0, 1]$

The solution to $Mv = x$, for each x vector above, gives the following three solutions:

free variable	v solution vector
x_3	$[19, 89, 1, 0, 0]$
x_4	$[100, 42, 0, 1, 0]$
x_5	$[0, 0, 0, 0, 1]$

These three v vectors are the basis vectors. Using these, perform a series of GCD computations, $\gcd(f(x), g(x) - s)$ with $s \in GF(q)$. It is impractical to list the full computations for this step, but eventually this will produce the following 3 factors:

- $79 + 68x + 78x^2 + x^3$
- $88 + x$
- $41 + x$

Algorithm Analysis

Berlekamp's algorithm can be used to either find a full factorization, as demonstrated, or to test for irreducibility. In the case of irreducibility, the algorithm is significantly more efficient because the computation of gcds at the end can be skipped [17].

> **Theorem**: The number of distinct irreducible factors of $f(x)$ is equal to the number of basis vectors in the null space of $(Q - I)$

Looking at the above example, there are three basis vectors, and it produces three irreducible factors. If only one basis vector exists, this means there is only one polynomial that divides $f(x)$, itself. Therefore, Berlekamp's algorithm

can be used to test for irreducibility by checking the number of basis vectors. One basis vector implies irreducibility, while two or more basis vectors implies reducibility. The gcd computations are only necessary to extract the exact factors.

Let $M(n)$ be the cost of performing a polynomial multiplication on a polynomial of degree n. Then the time complexity of Berlekamp's algorithm is $\mathcal{O}(n^3 + M(n)\log(n)\log(p))$, using efficient algorithms for the elimination operations [18] [19]. The space complexity is $\mathcal{O}(n^2)$ additional memory, based on the storage of the matrices in the computations. When just checking for irreducibility, avoiding the costly gcd computations, the running time is a much simpler $\mathcal{O}(n^3)$.

2.8.4 Distinct Degree Factorization

One way to categorize factoring algorithms in finite fields is as either distinct degree factorization algorithms or equal degree factorization algorithms. A distinct degree factorization algorithm produces a partial factorization of a polynomial. Each of the partial factors produced is itself the product of all polynomial factors of a certain degree. In the case of the following distinct degree factorization algorithm, the output of the algorithm, applied to a polynomial $f(x)$, is an array $[f_1, f_2, f_3, ..., f_n]$ where f_i is the product of all factors of degree i in f. In the case that no factor of a particular degree exists, the entry can just be left as a 1. Comparatively, an equal degree factorization algorithm takes a polynomial with all factors the same degree and fully factors it into irreducibles. The results of a distinct degree factorization are consequently valid inputs to an equal degree factorization algorithm.

The Cantor-Zassenhaus and Shoup factorization algorithms both require some form of a distinct degree factorization algorithm to factor arbitrary polynomials. One possible solution to this problem is presented in Shoup [18] and mirrored in

[2]. It may be used as a part of either the Cantor-Zassenhaus or Shoup algorithm to produce a distinct degree factorization. The idea behind this algorithm is the following theorem[18][pg. 367]:

> **Theorem**: Let a, b be positive integers and p a prime. The polynomial $x^{p^a} - x^{p^b} \in F_p[x]$ is divisible by the product of the irreducible polynomials in $F_p[x]$ that have a degree that divides $a - b$.

The following algorithm takes as input a squarefree polynomial $f \in F_p[x]$ with p a prime. It outputs a vector $[f_1, f_2, ..., f_n]$ of polynomials, all elements of $F_p[x]$, where f_i is the product of all irreducibles of degree i. This algorithm applies the above theorem in a clever way. In the final nested loops, when constructing the answer vector, the construction of the I table allows the outer loop to test for partial factors of several different degrees at the same time. The inner loop then extracts the product of irreducibles for each of the degrees tested by the outer loop. The following pseudocode is partially based on [18][pg. 5-6] but the high level ideas behind the algorithm itself are a standard way to perform a distinct degree factorization [20]:

```
fn distinct_factor(poly f, int prime) -> vec<poly> {
  let n = deg(f)
  let β = floor(n/2)
  let l = floor(sqrt(β))
  let m = ceil(β/l)

  let h = vec of size (l + 1)
  for i in 0 to (l + 1) {
    h[i] = x^(p^i) mod f(x)
  }

  let H = vec of size (m + 1)
  for i in 1 to (m + 1) {
    H[i] = x^(p^(l*i)) mod f(x)
  }

  let I = vec of size (m + 1)
  for i in 1 to (m + 1) {
    let product = 1
    for j in 0 to l {
      product = product * (H_i − h_j) mod f(x)
    }
    I[i] = product
  }

  let answer = vec of size n, all entries initialized to 1
  let fstar = f
  for i in 1 to m {
    let g = gcd(fstar, I[i])
    fstar = fstar / g
    for j in (l − 1) down to 0 {
      let partial_factors = gcd(g, H[i] − h[j])
      f[l * i − j] = partial_factors
      g = g / partial_factors
    }
  }

  if deg(fstar) ≥ 1 {
    answer[deg(fstar)] = fstar
  }

  return answer
}
```

Algorithm Analysis

Suppose $M(n)$ is the cost of a polynomial multiplication of degree n. This optimized approach to distinct degree factoring has an asymptotic running time of $\mathcal{O}(M(n)(n\log(n) + \log(p)))$ and requires $\mathcal{O}(n^{1.5})$ additional space [18]. When working on large problems, a performance improvement may be obtained by using modular composition to compute the h and H tables. Modular composition means that the tables can be computed by a series of polynomial substitutions. In this case, the substitutions are relatively straightforward $h_{i+1} = h_i(h_1)$ and $H_{j+1} = H_j(H_1)$.

2.8.5 Cantor Zassenhaus

The Cantor-Zassenhaus algorithm factors a polynomial $f(x) = a_0 + a_1x + ... + a_nx^n \in F_p[x]$ with p a prime. The first step when applying the algorithm to an arbitrary polynomial is to check that the polynomial is squarefree. The second step is to apply a distinct degree factorization algorithm to split the polynomial into partial factors f_r, where each f_r is a product of all the irreducibles of degree r. The algorithms presented above can accomplish both of these steps. The main contribution by Cantor and Zassenhaus in [21] is the final step of the algorithm, an equal degree factorization.

The Cantor-Zassenhaus equal degree factorization algorithm centers around a function that produces at least one factor when applied to $f(x)$. This function can be repeatedly applied until it produces a full factorization. The following steps produce a partial factorization using the Cantor-Zassenhaus algorithm on inputs $f(x)$ (the polynomial being factored) and p (a prime):

1. Choose a random polynomial $b \neq 0 \in F_p[x]$ with degree less than $f(x)$
2. Compute $m = \frac{p^r-1}{2}$, where r is the degree of the irreducible factors

3. Compute $b^m \pmod{f(x), p}$

4. Compute the following three values. With high probability, one of them is a non-trivial factor:

   (a) gcd(b^m, f)

   (b) gcd($b^m + 1$, f)

   (c) gcd($b^m - 1$, f)

5. If none of the gcd calculations yield a factor, choose a different b and try again

6. Otherwise one of the gcd calculations has resulted in a partial factorization. If this partial factorization is not an irreducible, reapply the factoring algorithm to the partial factorization until it is factored into a product of irreducibles

In this context, a trivial factor is either $f(x)$ itself or a constant, both of which divide $f(x)$ but do not meaningfully help factor $f(x)$ into irreducibles. Also, all of the irreducible factors of $f(x)$ must have the same degree, referred to as r, as this is an equal degree factorization. Therefore, a polynomial is known to be irreducible if its degree is r. As a result, it is known at the start of the equal degree factorization algorithm that $f(x)$ has the following form:

- $f(x) = \prod_{i=1}^{n/r} g_i(x)$ where all $g_i(x)$ have degree r

This polynomial can be rewritten in the following way:

- $a(x) = \sum_{i=1}^{n/r} a_i e_i(x)$

Part of the constraints on this polynomial are:

- $a_i = 0, -1, +1$

- $a(x) \not\equiv 0, -1, +1 \bmod f(x)$

From this partial list of constraints, we can construct two sets S and T as follows:

- $S = \{i : a_i = 0\}$
- $T = \{i : a_i = 1\}$

If T contains at least one element, then $\gcd(f(x), a(x) - 1) = \prod_{i \in T} u_i(x)$ yields at least one non-trivial factor u of f(x). Similarly, if S contains at least one element, $\gcd(f(x), a(x)) = \prod_{i \in S} u_i(x)$ gives a non-trivial factor u of $f(x)$. The other part of the constraints imposed on $a(x)$ are consequently the following:

- $e_i(x) \equiv 1 \pmod{u_i(x)}$
- $e_i(x) \equiv 0 \pmod{u_j(x)}$ for all $j \neq i$

If a polynomial satisfies the above properties, then the algorithm will produce at least one irreducible factor. Unfortunately, there is no clever way to choose one of these polynomials $a(x)$ that satisfies all the above properties. Rather, a polynomial is chosen at random, and it is assumed it has the correct properties. The algorithm is then run on this random input, either producing a valid factor or failing. The approach of simply picking a random polynomial works well in this case because most polynomials satisfy the above properties.

Listing 2.14: Cantor-Zassenhaus Factorization Pseudocode

```
fn cz_factor(poly f, int p, int r) -> vec<poly> {
  let answers = new vec
  let remaining = f

  for i in 0 to (deg(f) / r) {
    if deg(remaining) = 0 {
      break loop
    }

    if deg(remaining) = r  {
      answers.add(remaining)
      break loop
    }

    let partial_factors = cz_partial_factor(remaining, p, r)

    for factor in factors {
      remaining = remaining / factor
      answers.add(factor)
    }
  }

  return answers
}

fn cz_partial_factor(poly f, int prime, int r) -> vec<poly> {
  if deg(f) = r {
    return f
  }

  loop {
    let b = rand_poly()
    let m = (p^r-1)/2
    let bm = b^m (modd f(x),p)

    // These partial factors are not always irreducible
    // If they are not, another application of the algorithm
    // is necessary to factor them
    let gcd_zero = gcd(bm, f);
    if gcd_zero is a non-trivial factor {
      return cz_factor(gcd_zero, p, r)
    }

    let gcd_plus = gcd(bm + 1, f);
    if gcd_plus is a non-trivial factor {
      return cz_factor(gcd_plus, p, r)
    }
```

```
    let gcd_minus = gcd(bm - 1, f);
    if gcd_minus is a non-trivial factor {
      return cz_factor(gcd_minus, p, r)
    }
  }
}
```

Algorithm Analysis

This algorithm is probabilistic due to b being randomly chosen. Despite some randomness, the algorithm has a very high chance of producing a non-trivial factor on every iteration. There is at least a $1/2$ chance of choosing a b such that one of the gcd computations yields a factor [21].

The exact running time of Cantor-Zassenhaus varies somewhat based on the implementation of the modular power (powmod) function. Cantor and Zassenhaus provide an overall running time of $\mathcal{O}(n^3 + n^2 \log q)$ with this in mind. This algorithm requires surprisingly little additional storage, storing just the irreducible factors and a few intermediate polynomials. The space complexity is consequently $\mathcal{O}(n)$ where $n = \deg(f)$. Note that using asymptotically faster algorithms it is possible to get this running time down to:

$$\mathcal{O}(n^2(\log(n)\log\log(n))(\log(p) + \log(n)))$$

although this is unlikely to be worth it unless working on large problems [22].

2.8.6 Shoup Factorization

The Shoup factorization algorithm was first presented by Shoup in his 1995 paper [18]. This algorithm uses the gcd to extract factors, similar to the Cantor-Zassenhaus algorithm. Let p be a prime and $f(x) = a_0 + a_1 x + ... + a_n x^n$ be a polynomial with coefficients $0 \leq a_i \leq (p-1)$. The first step in the Shoup algorithm is to verify that the polynomial $f(x)$ is squarefree. Next, apply a

Distinct Degree Factorization algorithm to produce an array of partial factors f_i, such that f_i is a product of irreducibles of degree i. The algorithms above can perform these two steps. The third and final step is an equal degree factorization on the partial factors f_i, to produce a full factorization of f.

The inputs to the Shoup equal degree factorization algorithm are an integer d, the degree of the irreducible factors, and a polynomial $f(x) \in F_p[x]$, a product of irreducibles of degree d. The output is $r = (\deg(f)/d)$ irreducible polynomial factors of $f(x)$. Using these inputs, the algorithm proceeds through the following steps:

1. If $\deg(f) = d$ then return f as an irreducible factor
2. Choose a random polynomial $g \in F_p[x]$ with $\deg(g) < \deg(f)$
3. Compute $h = T_d(g) \pmod{f(x)}$. See directly below.
4. Choose a random constant b
5. Compute $a = (b + h)^{(p-1)/2} - 1 \pmod{f(x)}$
6. Compute $v = \gcd(a, f)$
7. If $\deg(v) = 0$ then go back to step 4 and try a different value for b
8. Otherwise v is partial factorization of $f(x)$. Recursively apply the algorithm to v and f/v.

$T_d(x)$ is a special function, defined as the sum of several different powers of x^{p^i}:

- $T_k(x) = \sum_{0 \le i < k} x^{p^i}$

This can be implemented easily using the powmod function discussed previously. Using this function, the final pseudocode for the Shoup algorithm is the following:

Listing 2.15: Shoup Factorization Pseudocode

```
fn shoup_equal_degree_factor(poly f, int d, int p) -> vec<poly> {
  if deg(f) = d {
    return f
  }

  let g = rand_poly()
  let h = T_d(g)

  loop {
    let b = rand_int()
    let a = (b+h)^((p-1)/2) - 1 mod f(x)
    let v = gcd(a, f)
    if v is a trivial factor {
      continue; // try again
    }

    let part1 = shoup_equal_degree_factor(v, d, p)
    let part2 = shoup_equal_degree_factor(f / v, d, p)

    return join(part1, part2);
  }
}
```

Algorithm Complexity

Let $M(n)$ be the cost of a polynomial multiplication of degree n. Then, the Shoup equal degree factorization algorithm has an asymptotic running time of $\mathcal{O}(n^2 \log(n) + M(n) \log(n) \log(p))$. Gathen and Panario present a similar value of $\mathcal{O}(n^2 + n \log(p))$ when dropping all $\log(n)$ factors from the running time [20]. The algorithm requires $\mathcal{O}(n^{1.5})$ additional space to perform the computation. This is one of the best running times for polynomial factoring with non-deterministic algorithms [20].

Chapter 3

Algorithm Performance

Different multiplication and factoring algorithms have different constants that are ignored by a simple Big-O analysis of the algorithms. These constants may mean that an asymptotically faster algorithm only starts saving time at a certain problem size. One way to see the effect of these constants is to implement the algorithms and compare their actual running times. This section covers some implementation results for various algorithms detailed in the background, based on a Rust implementation. This implementation includes approximately 3550 lines of Rust code and documentation, in addition to approximately 850 lines of tests. Rust was chosen because it is a systems language with good performance, is a relatively popular language overall, and has language features designed around writing secure programs. These language features help to prevent various security issues such as memory errors and threading errors. All running times are based on a single threaded implementation running on an Intel i5-4690. The programs have access to 24 GB of RAM, but none of the sample problems highlighted are in any way limited by the available memory.

3.1 Long Division

Let a, b be polynomials with $\deg(a) = m$, $\deg(b) = n$, $m \geq n$. Then, as seen in the background section, the time complexity for long division of a/b is $\mathcal{O}(n(m-n))$. The following are two sample runs, the first involving divisions on polynomials such that $m = 2n$ and the second polynomials such that $m = 4n$:

Table 3.1: Long Division Running Times (m = 2n)

m	n	time (seconds)	$\frac{time}{(n)(m-n)} \times 10^9$
200	100	0.004	434.48
500	250	0.023	372.31
800	400	0.062	393.28
1100	550	0.122	404.89
1400	700	0.212	434.60
1700	850	0.294	407.36
2000	1000	0.405	405.92
2300	1150	0.512	387.50
2600	1300	0.689	408.01
2900	1450	0.830	395.23
3200	1600	1.009	394.33
3500	1750	1.245	406.58
3800	1900	1.434	397.23
4100	2050	1.663	395.79
4400	2200	1.962	405.45
4700	2350	2.187	396.19
5000	2500	2.483	397.30
5300	2650	2.843	404.97
5600	2800	3.162	403.43
5900	2950	3.557	408.78
6200	3100	3.941	410.16
6500	3250	4.314	408.46
6800	3400	5.461	472.47
7100	3550	7.579	601.41
7400	3700	6.583	480.93

Table 3.2: Long Division Running Times (m = 4n)

m	n	time (seconds)	$\frac{time}{(n)(m-n)} \times 10^9$
400	100	0.011	399.4
1000	250	0.090	483.90
1600	400	0.200	417.95
2200	550	0.391	431.39
2800	700	0.662	450.66
3400	850	0.836	386.13
4000	1000	1.171	390.36
4600	1150	1.635	412.31
5200	1300	2.077	409.72
5800	1450	2.444	387.61
6400	1600	2.983	388.43
7000	1750	3.643	396.61
7600	1900	4.170	385.08
8200	2050	5.030	399.00
8800	2200	5.792	398.94
9400	2350	6.571	396.67
10000	2500	7.436	396.63
10600	2650	8.246	391.41
11200	2800	9.262	393.79
11800	2950	10.436	399.74
12400	3100	11.820	409.99
13000	3250	12.323	388.92
13600	3400	13.531	390.18

The last column is the ratio of actual running time vs theoretical (Big-O) running time. The running time is increasing by about the same amount as the expected running time, giving an idea of the constant while also verifying that the expected running time of $n(m-n)$ is accurate in these cases.

3.2 Computing Remainders

The background section covered two possible ways to compute the remainder $a(x)$ mod $f(x)$. One approach is to perform the division $a(x)$ / $f(x)$, yielding a quotient and remainder as the result of the computation. Another approach is substituting precomputed powers x^i mod $f(x)$. Let $m = \deg(a)$ and $n = \deg(f)$,

then long division has expected running time of $\mathcal{O}(n(m-n))$ and precomputed powers have a running time of $\mathcal{O}(nm)$. An example of how this performs on some random inputs with $m = 2n$, $m = 5n$ and $m = 1.3n$ are as follows. The case of $m = 2n$ commonly occurs when taking the remainder after multiplying two field elements. $m = 5n$ is chosen to demonstrate the case when the operands are further apart in size, and $m = 1.3n$ is chosen to demonstrate the case where the operands are closer together in size:

Table 3.3: Remainder Running Times (m = 2n)

$m = 2n$	n	Long Division (ms)	Powers (ms)	Ratio of LD to Powers
20	10	0.1018	0.0476	2.13
50	25	0.3138	0.2705	1.16
100	50	1.6602	1.1424	1.45
200	100	6.265	6.0589	1.03
400	200	22.6519	17.9247	1.26
800	400	105.1052	91.2984	1.15
1600	800	273.9331	304.2688	0.90
3200	1600	1068.7148	1213.0614	0.88
6400	3200	4405.2627	5241.4847	0.84

Table 3.4: Remainder Running Times (m = 5n)

$m = 5n$	n	Long Division (ms)	Powers (ms)	Ratio of LD to Powers
50	10	0.6286	0.1856	3.38
125	25	1.3933	1.0443	1.33
250	50	9.5936	5.0577	1.89
500	100	20.9560	18.7054	1.12
1000	200	98.5686	115.5494	0.85
2000	400	458.2421	616.5629	0.74
4000	800	1998.0806	2100.9403	0.95
8000	1600	5499.0930	8136.9583	0.67

Table 3.5: Remainder Running Times (m = 1.3n)

$m = 1.3n$	n	Long Division (ms)	Powers (ms)	Ratio of LD to Powers
32	25	0.0984	0.0923	1.06
65	50	0.3491	2.449	0.14
130	100	1.2371	1.4016	0.88
260	200	5.7100	13.0366	0.43
520	400	62.2972	34.2877	1.81
1040	800	139.2233	196.99	0.70
2080	1600	541.4425	595.5528	0.90
4160	3200	1892.5455	1852.1299	1.02

The worst case running times of both algorithms are very similar, if not effectively identical for $m = 2n$. Regardless of this similarity, these numbers indicate that the powers method is faster for small problems, while long division is faster for larger problems. It makes sense that long division would scale better than the powers method. The power method always performs n calculations on each of the m iterations of the algorithm, whereas, each successive iteration of the long division algorithm reduces the problem size by a little bit. This results in similar asymptotic running times but means long division can do fewer computations under certain circumstances. This is a good example of how the asymptotic running time is a good starting point, but programming the algorithms can highlight differences in the overall performance.

3.3 GCD and Inverses

3.3.1 Integer GCD

The greatest common divisor problem can be solved using the Euclidean algorithm. Let x and y be integers, then a computation of $\gcd(x, y)$ is expected to take $\mathcal{O}(\log(n))$ divisions using a standard implementation of the Euclidean algorithm. In this case, $n = \max(x, y)$, effectively meaning the remaining problem

size shrinks by 1 bit per iteration. Consider the general problem of computing $\gcd(a, b)$ where:

- $a = p \times q$
- $b = p \times r$
- p, q, r are all L bit integers i.e. of approximately the same size

Computing $\gcd(a, b)$ therefore results in the value of p. An example of the running times for some random p, q, r values of specified size is:

Table 3.6: Integer GCD Running Time

bits in p, q, r $(= L)$	time (nanoseconds)	time / L
8	2	0.25
16	5	0.3125
32	8	0.25
64	18	0.2812
128	43	0.3359
256	71	0.2773
512	134	0.2617
1024	313	0.3056
2048	573	0.2797
4096	1188	0.2900
8192	2354	0.2873
16384	4783	0.2919
32768	9590	0.2926
65536	18853	0.2876
131072	38153	0.2910
262144	76517	0.2918
524288	153206	0.2922

The last column is again a ratio of actual running time to asymptotic running time, which should be about the same number if the algorithm is scaling as expected. In this case, the ratio remains about the same at approximately 0.29. This is fairly representative of how the Euclidean algorithm should perform in practice as a $\mathcal{O}(\log(n))$ algorithm.

3.3.2 Polynomial GCD

The Euclidean algorithm also operates on polynomials. Let a, b be polynomials with $m = \deg(a)$ and $n = \deg(b))$, $m \geq n$. The running time of this algorithm was shown to be $\mathcal{O}(mn(m-n))$ in the background section if division uses the long division algorithm. This is based on the fact that polynomial gcd decreases the degree of the remaining polynomials by one per iteration, requiring m iterations, as well as the expected $\mathcal{O}(n(m-n))$ cost of division. We can build an example problem similar to the integer version with the following restrictions:

- $a = f \times g$
- $b = f \times h$
- a, b, f, g, h are polynomials
- $\deg(f) = \deg(g) = \deg(h)$ i.e. $\deg(a) = \deg(b)$

Computing $\gcd(a, b)$ again yields a common factor of f. This particular problem setup is a very common application of the gcd algorithm. Under this specific set of conditions, $m = n$. Consequently, the running time simplifies to[1] $\mathcal{O}(mn(m-n)) = \mathcal{O}(nn(n-n)) = \mathcal{O}(n^2)$. Here is a table of running times, considering this simplification:

[1]The division algorithm requires one iteration in the case that both polynomials are the same degree, meaning $(m-n)$ is best replaced by a one and not a zero here

Table 3.7: Polynomial GCD Running Times

degree	time (seconds)	$\frac{time}{n^2} \times 10^9$
8	0.0001	2078.1250
16	0.0004	1688.6718
32	0.0015	1474.2187
64	0.0049	1204.1748
128	0.0199	1219.4274
256	0.0758	1156.6360
512	0.3339	1273.8357
1024	1.2709	1212.0513
2048	4.8842	1164.4937
4096	21.5758	1286.0180

Ignoring the very small problems that take less than a millisecond, this running time seems to be accurate. This simplification follows because the long divisions that occur during the computation are expected to follow this pattern:

1. Divide a degree n polynomial by a degree n polynomial
2. Divide a degree n polynomial by a degree $n-1$ polynomial
3. Divide a degree $n-1$ polynomial by a degree $n-2$ polynomial
4. ....
5. Stop when reaching 0

Dividing polynomials of almost exactly the same degree is the best case for long division $\Omega(n)$. Effectively, that means that polynomial GCD on two polynomials of the same degree is only doing n iterations requiring $\mathcal{O}(n)$ work each, for approximately $\mathcal{O}(n^2)$ total work. This is where the above simplification comes from. This allows the Euclidean algorithm to scale much more efficiently than originally expected for most problems.

3.3.3 Polynomial Inverses

The Extended Euclidean algorithm has similar time and space bounds to the normal Euclidean algorithm. Let the two operands be a and b with $m = \deg(a)$ and $n = \deg(b)$, $m \geq n$. Then, the overall running time is $\mathcal{O}(m(M+D))$, where M is the cost of a multiplication and D is the cost of a division. Consider an example problem set up similarly to the polynomial GCD:

- Let p be a prime
- Let I be an irreducible in $F_p[x]$
- Let x be a random field element of $GF(p^n)$ (therefore x is also a polynomial in $F_p[x]$)

The running time of $\mathcal{O}(n(M+D))$ can be simplified for this particular problem. First, note that with very high probability, the random field element will have degree one less than the irreducible, implying that in most cases $m \approx n$. The division cost will tend to operate in the best case for the same reason as the standard euclidean algorithm, giving $D \approx n$. Interestingly, $M \approx n$ as well because the multiplications also tend to operate in close to their best case. This means the actual running time should be very close to $(n(M+D)) \approx (n(n+n)) \approx n^2$ with these parameters:

Table 3.8: Polynomial Inverse Running Times

degree(I)	time (seconds)	$\frac{time}{n^2} \times 10^9$
8	0.0003	5120.3125
16	0.0009	3847.6562
32	0.0035	3467.6757
64	0.0134	3292.2607
128	0.0686	4191.9311
256	0.3213	4902.7099
512	0.9907	3779.3521
1024	4.5341	4324.1183
2048	21.0396	5016.2411
4096	63.1193	3762.2091

The running time (and algorithm constant) understandably increases compared to the standard Euclidean algorithm because more computations are required on each iteration of the algorithm. The running time of this algorithm is not as clean as many of the others but the estimate of n^2 scaling is not a bad guess. Although slower, the extended version of the Euclidean algorithm is still quite fast and able to handle extremely large inputs.

3.4 Polynomial Multiplication

The three polynomial multiplication methods described in the background section have theoretical running times of $\mathcal{O}(n^2)$ for the schoolbook method, $\mathcal{O}(n^{log_2(3)})$ for Karatsuba method, and $\mathcal{O}(n \log n)$ for the Fast Fourier Transformations (FFT) method. A simple way to compare these methods is to take some random polynomials of the same degree n and multiply them together, using each method:

Table 3.9: Comparison of Multiplication Running Times

n	Naive (ms)	Karatsuba (ms)	FFT (ms)	Naive:Karatsuba	Naive:FFT
10	0.0454	0.0397	0.0832	1.1435	0.5456
25	0.4721	0.1947	0.1293	2.4247	3.6511
50	1.0572	0.3085	0.2347	3.4269	4.5044
100	4.1647	1.0680	0.3931	3.8995	10.5945
200	17.6586	3.1007	0.9055	5.6950	19.5014
400	106.6711	25.4503	1.7047	4.1913	62.5747
800	558.2004	105.3403	8.2992	5.2990	67.2595
1600	1891.4647	74.8062	6.4819	25.2848	291.8071
3200	5299.5947	485.2856	30.8499	10.9205	171.7864

These algorithms are scaling roughly as expected. The schoolbook method is scaling relatively poorly with problem size, Karatsuba is scaling at a much slower rate; and FFT-based multiplication is scaling quite slowly overall, in line with a $\mathcal{O}(n \log n)$ running time. As before, it is also interesting to compare the algorithm running times to their respective asymptotic running times, to see if the algorithms are scaling by roughly the expected value every time the problem size is increased.

Table 3.10: Naive Multiplication Running Times

n	Schoolbook Time (ms)	time / n^2
10	0.0453	453.00
25	0.2501	400.16
50	1.0217	408.68
100	3.8665	386.65
200	14.9961	374.90
400	122.3381	764.61
800	441.9398	690.53
1600	1343.1662	524.67
3200	5446.0570	531.84

Table 3.11: Karatsuba Multiplication Running Times

n	Karatsuba Time (ms)	time / $n^{log_2(3)}$
10	0.0344	925.88
25	0.1054	673.09
50	0.3148	677.09
100	1.0301	746.24
200	2.6601	649.05
400	9.7352	800.03
800	23.7501	657.37
1600	70.5144	657.36
3200	211.7227	664.78

Table 3.12: FFT Multiplication Running Times

n	FFT Time (ms)	time / $n \log n$
10	0.0768	2311.91
25	0.0882	759.71
50	0.2644	936.94
100	0.4495	676.56
200	0.7692	503.14
400	1.5800	456.97
800	3.4332	444.99
1600	6.1499	361.11
3200	18.7093	502.12

The algorithms are again not scaling cleanly with the expected running times. However, the ratios are at least somewhat consistent at around 400-500 for

schoolbook multiplication, 650 for Karatsuba multiplication and 400-500 for the FFT method. Despite this, the algorithms with higher asymptotic running times are generally taking longer to complete (as expected). One interesting result here is that Karatsuba multiplication seems to always be running faster than the schoolbook method. This implies the cost of multiplication is over three times higher than addition for this particular CPU architecture. In addition, the FFT method seems to have a larger constant than the other algorithms. The FFT algorithm is only starting to substantially improve over the Karatsuba algorithm at around $n = 100$.

3.5 Factoring Algorithms

The final major class of algorithms covered in the background section are factoring algorithms. The naive algorithm is hopelessly inefficient on any practical problem, but the other algorithms can solve large problems relatively fast. For a fair comparison between the algorithms, the main sample problems covered here are equal degree factoring problems. The algorithms have expected running times of $\mathcal{O}(n^3+M(n)\log(n)\log(p))$ for Berlekamp's algorithm, $\mathcal{O}(n^3+n^2\log(q))$ for Cantor-Zassenhaus factoring and $\mathcal{O}(n^2\log(n)+M(n)\log(n)\log(p))$ for Shoup factoring. The first example problem is factoring a product of n linear polynomials (therefore, the algorithm is factoring a n^{th} degree polynomial) with each of the three algorithms:

Table 3.13: Berlekamp Factorization Running Times

n	time (seconds)	time / expected
4	0.7570	2315206.66
8	1.7503	1032302.44
12	2.8523	645715.85
16	4.6988	532129.73
20	6.7020	438452.34
24	9.1025	377028.26
28	11.3700	317902.91
32	15.4180	305175.69
40	25.2993	278273.96
48	36.1331	243646.79
56	59.0184	261514.82
64	98.6505	302553.41
96	407.8303	400570.61

Table 3.14: Cantor-Zassenhaus Factorization Running Times

n	time (ms)	time / expected
4	1.0535	3538.01
8	3.7498	2591.31
12	8.4872	2214.88
16	11.0631	1411.78
20	19.1333	1382.04
24	25.3358	1139.22
28	34.3625	1028.61
32	43.212	905.36
40	69.0126	789.82
48	97.9518	679.02
56	117.8978	532.42
64	164.1077	509.66
96	393.7388	386.25
128	642.5449	274.99
256	2,922.8000	164.80
512	14,749.7000	106.84
1024	79,443.5000	72.94
2048	616,390.8000	71.24

Table 3.15: Shoup Factorization Running Times

n	time (ms)	time / expected
8	4.1208	2995.78
12	8.2031	2558.99
16	13.1023	2275.39
20	21.522	2387.49
24	28.5307	2201.47
28	37.6704	2142.46
32	48.7465	2131.09
40	70.1393	1979.64
48	98.3757	1945.06
56	129.763	1900.59
64	165.7135	1872.64
96	356.255	1835.66
128	590.9386	1746.27
256	2,260.204	1747.06
512	8,507.1322	1719.27
1024	34,268.3551	1792.71
2048	144,490.9323	1936.47

A slightly more difficult problem is factoring a product of n quadratic irreducibles (making the overall polynomial being factored of degree $2n$). Running times for the 3 algorithms are as follows:

Table 3.16: Berlekamp Factorization Running Times

n	time (seconds)	time / expected
8	2.1694	1279518.08
16	11.1805	1266158.97
24	24.1482	1000222.85
32	40.4934	801503.35
40	69.6777	766404.18
48	96.3821	649907.57
56	146.7201	650127.18
64	212.3548	651275.75
80	394.6287	649145.84
96	710.9821	698326.03
112	1,191.4544	753615.78
128	1,956.7945	843132.23

Table 3.17: Cantor-Zassenhaus Factorization Running Times

n	time (ms)	time / expected
8	9.2503	6392.45
16	23.4058	2986.85
24	43.4275	1952.71
32	71.7528	1503.33
40	122.0647	1396.99
48	187.1089	1297.07
56	248.8702	1123.90
64	299.1382	929.03
80	452.8581	747.89
96	843.047	827.01
112	1,169.5069	736.37
128	1,287.4916	551.02
192	2,633.5096	345.76
256	4,244.0953	239.30
512	20,283.4919	146.93
1024	87,368.6142	80.22
2048	506,162.5858	58.50

Table 3.18: Shoup Factorization Running Times

n	time (ms)	time / expected
8	4.5494	790.06
16	21.803	953.18
24	44.7643	885.06
32	76.2068	861.17
40	112.9247	828.01
48	158.4961	816.67
56	211.4774	808.87
64	282.6705	835.31
80	475.5885	913.31
96	581.2588	784.89
112	772.1757	774.10
128	959.2757	742.86
192	2,158.8919	762.56
256	3,703.2278	748.41
512	16,838.3733	880.88
1024	54,799.3363	734.42
2048	224,309.9821	761.58

Overall, the Berlekamp algorithm is behaving relatively consistently across different input sizes and serves as a good point of comparison. The Cantor-Zassenhaus algorithm is running in substantially less time than Berlekamp on the same problem size. Cantor-Zassenhaus is also running noticably faster than its worst case running time on this particular set of inputs. The Shoup factoring algorithm is beating both Berlekamp's algorithm and Cantor-Zassenhaus by significant amounts and is running very close to its expected running time across various problem sizes. Overall, the better factoring algorithms are running much faster.

3.6 Summary of Results

Most of these algorithm implementations are running at close to their Big-O time estimates. These implementations have highlighted several facts, some of which

are not obvious from the theoretical analysis of the algorithms. First, when computing remainders, the running time of long division is $\mathcal{O}(n(m-n))$ while precomputed powers have a running time of $\mathcal{O}(nm)$. As n and m are positive integers, $nm \geq n(m-n)$, so long division has a better asymptotic running time and should scale better. Despite this, precomputed powers runs faster on smaller problems. This implies that the algorithm has a smaller constant, a result that makes sense given the simplicity of the algorithm. The Euclidean and Extended Euclidean algorithm implementations scaled almost exactly as expected on the chosen sample problems.

The three multiplication algorithms covered - the naive method, Karatsuba, and Fast Fourier Transformations - are all scaling roughly as the Big-O analysis indicates. In particular, on this CPU architecture, the naive method is strictly worse than the Karatsuba approach on all problem sizes. This implies that the cost of multiplication is very expensive compared to addition (at least three times the cost). In addition, Fast Fourier Transformations scale better than the Karatsuba approach but start off significantly slower on small problems, only beating Karatsuba at somewhere between $n = 50$ and $n = 100$. This indicates that the Fast Fourier Transformations approach has a significant constant and is only worth using on fairly large problems.

The factoring algorithm implementations are where the largest variance from the Big-O running times occur. In general, it is expected that Berlekamp is slower than Cantor-Zassenhaus and Cantor-Zassenhaus is slower than Shoup's algorithm. While this is still true in the sample implementations, Cantor-Zassenhaus is scaling significantly better than the Big-O running time. This may be because the sample problem chosen is not the worst case behaviour, or this implementation is using better algorithms than a standard implementation of Cantor-Zassenhaus described in the literature.

Chapter 4

Factoring Irreducibles over Irreducibles

4.1 Problem Description

All the algorithms looked at until now are related to univariate polynomial factoring problems, however, they can also be used to solve a more complicated factoring problem. Suppose now that we have two irreducible polynomials $a(x), b(x) \in F_p[x]$ with both polynomials sharing the same prime p. In addition, assume that $\deg(a) = \deg(b) = n$. Now, we want to factor $a(x)$ with respect to $b(x)$ and p. The univariate factoring algorithms can be used to solve this problem with some modifications to account for the extra information present. The algorithms in this chapter are the author's approach to this problem, one that is otherwise poorly covered in the literature.

Recall that one way to define a finite field is with an irreducible $I(x)$ of degree n and a prime p (giving a finite field $GF(p^n)$). As $b(x)$ is irreducible over $GF(p)$, it follows that $b(x)$ and p form one of these $GF(p^n)$ fields. The original factoring

problem involves coefficients in the field $GF(p)$. This new factoring problem involves polynomials whose coefficients are elements of $GF(p^n)$, defined by $I = b(x)$ and p. This approach allows for the modification of standard factoring algorithms such as Cantor-Zassenhaus and Shoup to solve the problem. The equal degree factoring algorithms work because the result of this new problem is guaranteed to be a product of linears (i.e. polynomials of the form $c_1(y)x+c_0(y)$, where $c_0(y)$ and $c_1(y)$ are elements of the field defined by $I = b(y)$ and q). The variable y is specifically used here for the coefficient polynomials as more than one variable is now present in the problem. From here on, it effectively becomes a problem involving both the variable x and y.

4.2 Examples

4.2.1 Factoring a Quadratic

Let $p = 25013$ be a prime. The following univariate polynomials are irreducible over $GF(p)$, a fact that can be verified using Berlekamp's algorithm:

- $a(x) = 21371 + 14261x + x^2$
- $b(x) = 13439 + 25012x + x^2$

The goal is to factor the irreducible $a(x)$ over $b(x)$ and p. First, note that $b(x)$ and p form a finite field because $b(x)$ is an irreducible. One approach to this problem is to try to factor $a(x)$, except the coefficients of $a(x)$ are now in the finite field $GF(p^n)$, with $I = b(x)$ and p. When approached this way, $a(x)$ factors into the following two roots, each linear in the variable x, with coefficients modd $b(y)$ and p:

- $f_1 = (1)x + (10627 + 18020y)$
- $f_2 = (1)x + (3634 + 6993y)$

It is not immediately obvious that this is correct, but $a(x) = f_1 \times f_2$ when the coefficients are reduced modd $(p, b(y))$:

- $f_1 \times f_2 = ((10627 + 18020y) + x)((3634 + 6993y) + x)$
- $f_1 \times f_2 = (10627 + 18020y)(3634 + 6993y) + (10627 + 18020y)x + (3634 + 6993y)x + x^2$
- $f_1 \times f_2 = (126013860y^2 + 139799291y + 38618518) + (14261 + 25013y)x + x^2$

Next, the coefficients need to be simplified. Simplifying the coefficients requires dividing them by $b(y) = 13439 + 25012y + y^2$ and then reducing the result mod $p = 25013$. The results of these intermediate calculations are:

1. $126013860y^2 + 139799291y + 38618518 \equiv (23379)(13439 + 25012y + y^2) + 21371 \equiv 21371$ modd $(p, b(x))$
2. $(25013y + 14261) \equiv 14261$ modd $(p, b(x))$

Therefore, once simplified, it turns out that the following is indeed true:

- $f_1 \times f_2 \equiv (21371) + (14261)x + x^2 \equiv a(x)$ modd $(p, b(x))$

4.2.2 Factoring a 5^{th} Degree Polynomial

The idea used in the previous example applies to irreducible polynomials of any degree. Let $p = 25013$ be the prime and let the two irreducibles in this case be 5^{th} degree polynomials:

- $a(x) = 21371 + 14261x + 23922x^2 + 13439x^3 + 23751x^4 + 1x^5$
- $b(x) = 13679 + 25012x + 1x^5$

It is again the case that $b(x)$ and p form a finite field for the coefficients. The math in this case becomes more difficult due to the higher degree of $b(x)$, but $a(x)$ still splits into a product of polynomials that are linear in the variable x:

- $f_1 = x + (10815 + 21351y + 14904y^2 + 17655y^3 + 17432y^4)$
- $f_2 = x + (1589 + 21336y + 7908y^2 + 20890y^3 + 16458y^4)$
- $f_3 = x + (18266 + 11759y + 21255y^2 + 15189y^3 + 1865y^4)$
- $f_4 = x + (4066 + 22768y + 18679y^2 + 9487y^3 + 19615y^4)$
- $f_5 = x + (14028 + 22838y + 12293y^2 + 11818y^3 + 19669y^4)$

The math is omitted here but it is again true that multiplying $f_1 \times f_2 \times f_3 \times f_4 \times f_5$, then simplifying all the coefficients modd $(p, b(y))$ would result in the polynomial $a(x)$. In order to solve this problem and develop a meaningful running time for the resulting algorithm, it is necessary to revisit and update the algorithms for division, gcd, and multiplication. The changes to these fundamental algorithms allow for the development of the final algorithms used to solve this problem.

4.3 Long Division Revisited

Ordinary polynomial long division has a running time of $\mathcal{O}(n(m-n))$ for two polynomials a and b with $\deg(a) = m$, $\deg(b) = n$, $m \geq n$. The new algorithm for long division on polynomials with polynomial coefficients is basically the same, except every operation on a coefficient becomes (a lot) more expensive. Recall that polynomial long division is essentially $(m - n)$ applications of the following set of steps, repeated until the intermediary remainder has a degree less than the divisor:

1. Multiply a value onto the divisor such that it will cancel the leading term of the intermediate remainder
2. Subtract this modified divisor from the intermediate remainder, canceling the highest power in the intermediate remainder

When operating with integer coefficients, the cost of these operations were assumed to be the following:

Operation	Cost
subtract	$\mathcal{O}(1)$
multiply	$\mathcal{O}(1)$

The original cost per iteration was therefore based on the following:

- $Cost = n \times \mathcal{O}(Multiply) + n \times \mathcal{O}(Subtraction)$
- $Cost = n(1) + n(1)$
- $Cost = n + n \in \mathcal{O}(n)$

Each iteration is expected to cost $\mathcal{O}(n)$, for the final running time of $\mathcal{O}(n(m-n))$. The costs for each of these operations when applied to polynomials of degree d, using the Karatsuba algorithm, is expected to be:

Operation	Cost
subtract	$\mathcal{O}(d)$
multiply	$\mathcal{O}(d^{log_2(3)})$

Therefore, the cost per iteration when using an irreducible of degree d should be:

- $Cost = n \times \mathcal{O}(Multiply) + n \times \mathcal{O}(Subtraction)$
- $Cost = n(d^{log_2(3)}) + n(d) = n(d + d^{log_2(3)}) \in \mathcal{O}(nd^{log_2(3)})$

giving a cost per iteration of $\mathcal{O}(nd^{log_2(3)})$ and overall running time of $\mathcal{O}(nd^{log_2(3)}(m-n))$ for the entire long division algorithm, if multiplication is done using the Karatsuba algorithm. More generally, the running time is $\mathcal{O}(n \times M(d) \times (m-n))$, where $M(d)$ is the cost of multiplying two polynomials of $d = \deg(b(x))$. The space requirements increase by a factor of d as well, due to the increased storage requirements of each coefficient, for a space complexity of $\mathcal{O}(d \times \max(m, n))$. A test using naive polynomial multiplication for the coefficients, and an irreducible of degree n ($M(d) = \mathcal{O}(n^2)$), gives the following running times, as an example:

Table 4.1: Long Division with Field Extension Running Times

m	n	time (seconds)	$\frac{time}{n^3(m-n)} \times 10^9$
6	2	0.0005615	17546.87
9	3	0.0012442	7680.24
12	4	0.0025697	5018.94
15	5	0.0049529	3962.32
18	6	0.0108586	4189.27
21	7	0.0167008	3477.88
24	8	0.0263428	3215.67
27	9	0.0360123	2744.42
30	10	0.0522947	2614.73
33	11	0.0785469	2682.42
36	12	0.1052628	2538.16
39	13	0.1377890	2412.18
42	14	0.2121874	2761.70
45	15	0.2397584	2367.98
48	16	0.3150897	2403.94
51	17	0.3856184	2308.51
54	18	0.4935785	2350.91
57	19	0.5984628	2296.11
60	20	0.7206848	2252.14
63	21	0.8716200	2240.88

4.4 GCD revisited

This algorithm behaves similarly when modified. Ordinarily, computing $\gcd(a, b)$ on two polynomials a and b with $m = \deg(a)$, $n = \deg(b)$, $m \geq n$, requires m iterations. The cost of these iterations is a division, which costs $\mathcal{O}(n(m-n))$ in the worst case or $\Omega(n)$ if $n \approx m$. The same basic idea holds here, except each iteration is more expensive. The cost of each iteration is still dominated by the division, which now has known running time of $\mathcal{O}(n \times M(d) \times (m-n))$, where $d = \deg(I)$. The result is, before any simplifications, a running time of $\mathcal{O}(mn \times M(d) \times (m-n)))$. The space requirements increase based on the degree of the irreducible d, resulting in a space complexity of $\mathcal{O}(md)$. A straightforward example sets $m = 2n$ and $d = n$. This allows for the running time to be simpli-

fied to $\mathcal{O}(mn \times M(d) \times (m-n))) = \mathcal{O}((2n)(n) \times M(n)(n-n)) = \mathcal{O}(n^2 M(n))$. Using naive multiplication ($M(n) = \mathcal{O}(n^2)$) causes this to nicely simplify to $\mathcal{O}(n^4)$ overall:

Table 4.2: GCD with Field Extension Running Times

m	n	d	time (seconds)	$\frac{time}{n^4} \times 10^9$
4	2	2	0.0003746	1463.28
6	3	3	0.0012391	956.09
8	4	4	0.0032988	805.37
10	5	5	0.0056068	560.68
12	6	6	0.0110568	533.21
14	7	7	0.0192335	500.66
16	8	8	0.0303014	462.36
18	9	9	0.0468673	446.45
20	10	10	0.0730250	456.40
22	11	11	0.1330926	568.15
24	12	12	0.1557454	469.42
26	13	13	0.2027382	443.65
28	14	14	0.2660945	432.91
30	15	15	0.6557226	809.53
32	16	16	0.5089717	485.39
34	17	17	0.5859317	438.46

4.5 Addition Revisited

The running time of standard polynomial addition for two univariate polynomials a and b with $n = MAX(\deg(a), \deg(b))$ is $\mathcal{O}(n)$. In the modified version of the algorithm, n iterations are still required. Each of these iterations has increased cost compared to the ordinary case, as each coefficient is a polynomial. Let I be the irreducible with $\deg(I) = d$, then each iteration involves a $\mathcal{O}(d)$ polynomial addition. In total, the algorithm has time complexity $\mathcal{O}(nd)$. Similarly, $\mathcal{O}(nd)$ space is required to store all the values.

4.6 Multiplication Revisited

4.6.1 Naive Method

The multiplication algorithms are modified similarly. The same number of iterations are required, but each iteration costs more than it did previously. Assume again that the multiplications are on two polynomials a and b, with $m = \deg(a)$ and $n = \deg(b)$. Recall that the pseudocode for naive polynomial multiplication is the following:

```
fn naive_multiply(poly a, poly b) -> poly {
  let answer = zero polynomial of degree (deg(a) + deg(b))

  for i in 0 to deg(a) {
    for j in 0 to deg(b) {
      answer[i + j] += a[i] * b[j]
    }
  }

  return answer
}
```

The only notable difference now is that $a[i] \times b[j]$ is a multiplication of polynomials of $\deg(I) = d$, instead of a multiplication of two integers. The overall cost of this algorithm is therefore $\mathcal{O}(M(d) \times nm)$. $M(d)$ has a cost between $\mathcal{O}(dlogd)$ and $\mathcal{O}(d^2)$, depending on the algorithm used for polynomial multiplication. Overall, this results in the following costs:

Method	$M(d)$	Cost
FFT	$\mathcal{O}(d \log d)$	$\mathcal{O}(nm(d \log d))$
Karatsuba	$\mathcal{O}(d^{log_2(3)})$	$\mathcal{O}(nm(d^{log_2(3)}))$
Naive	$\mathcal{O}(d^2)$	$\mathcal{O}(nm(d^2))$

Multiplying two polynomials of the same degree ($n = m$) with an irreducible of degree n results in the following simplification:

Method	$M(n)$	Cost
FFT	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^3 \log n)$
Karatsuba	$\mathcal{O}(n^{log_2(3)})$	$\mathcal{O}(n^{2+log_2(3)}) \approx \mathcal{O}(n^{3.58})$
Naive	$\mathcal{O}(n^2)$	$\mathcal{O}(n^4)$

4.6.2 Karatsuba

This can be improved by using the Karatsuba approach. When operating on simple univariate polynomials, the problem is split in 3 parts, each of size 1/2. Each of these three sub-problems requires $\mathcal{O}(1)$ multiplies. In total, $T(n) = 3T(\frac{n}{2}) + \mathcal{O}(1)$ multiplies are required, a recurrence[1] that solves to $\mathcal{O}(n^{log_2(3)})$. Let I be the irreducible used for the coefficient math, with $\deg(I) = d$. Then, the overall time requirement is $\mathcal{O}(M(d) \times n^{log_2(3)})$, with $M(d)$ varying from $\mathcal{O}(dlogd)$ to $\mathcal{O}(d^2)$. This gives an overall expected running time of the following:

Method	$M(d)$	Cost
FFT	$\mathcal{O}(d \log d)$	$\mathcal{O}(n^{log_2(3)}(d \log(d)))$
Karatsuba	$\mathcal{O}(d^{log_2(3)})$	$\mathcal{O}(n^{log_2(3)}(d^{log_2(3)}))$
Naive	$\mathcal{O}(d^2)$	$\mathcal{O}(n^{log_2(3)}(d^2))$

For comparison, if $d = n$, then the following simplifications apply:

Method	$M(n)$	Cost
FFT	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{1+log_2(3)} log(n)) \approx \mathcal{O}(n^{2.585} \log n)$
Karatsuba	$\mathcal{O}(n^{log_2(3)})$	$\mathcal{O}(n^{2log_2(3)}) \approx \mathcal{O}(n^{3.17})$
Naive	$\mathcal{O}(n^2)$	$\mathcal{O}(n^{2+log_2(3)}) \approx \mathcal{O}(n^{3.585})$

Using the approach of Karatsuba coefficient multiplication, we can test this algorithm on some random values. An example of the running times are the following:

[1]Note that $T(n) = 3T(\frac{n}{2}) + \mathcal{O}(1)$, the number of multiplications required, and $T(n) = 3T(\frac{n}{2}) + \mathcal{O}(n)$, the overall work required, both solve to $\mathcal{O}(n^{log_2(3)})$

Table 4.3: Karatsuba with Field Extension Running Times

n = d	time (seconds)	time / $n^{3.17}$
10	0.0017	1189.94
15	0.0029	568.64
25	0.0276	1056.81
35	0.0863	1139.89
50	0.2209	945.29
65	0.4449	830.88
80	1.1866	1149.66
90	1.4984	1000.52
100	1.7993	861.23
125	2.4155	571.19
150	7.8056	1037.43
175	11.6451	950.92
200	16.24242	869.75
400	141.5135	847.79

4.7 Powmod Revisited

Recall that computing a modular power involves the computation of a^b in a finite field. The method of computing powers by repeated squaring requires $\mathcal{O}(\log b)$ iterations. Each iteration requires at most two modular multiplications. The running time is consequently $\mathcal{O}(M \log b)$, where M denotes the cost of a modular multiplication. For integers, modular multiplication is very fast and can be assumed to take constant $\mathcal{O}(1)$ time in many problems. These iterations become more expensive when operating on polynomials. This particular instance of the powmod problem has the following inputs:

- A prime p
- An irreducible I_{coeff}, for the coefficient field
- An irreducible I_{outer}, for the finite field
- An integer b, the power
- A polynomial a, the value being multiplied. With high probability, $n = \deg(a) \approx \deg(I_{outer})$

A modular multiplication can be performed by multiplying then dividing by the irreducible. For this choice of parameters, this is a multiplication of two polynomials of degree n, resulting in a polynomial of degree $2n$. Then, this polynomial of degree $2n$ is divided by a polynomial of degree n. These operations have the following costs when using the Karatsuba algorithm for the overall multiplication:

Operation	Cost
Multiplication	$\mathcal{O}(n^{1.585} \times M_{coeff})$
Division	$\mathcal{O}(n^2 \times M_{coeff})$
Mult + Div	$\mathcal{O}((n^{1.585} + n^2) \times M_{coeff}) \in \mathcal{O}(n^2 \times M_{coeff})$

This gives a rough cost per iteration of $\mathcal{O}(n^2 \times M_{coeff})$. Repeated $\log b$ times, this gives the final running time of $\mathcal{O}(n^2 \times M_{coeff} \times \log b)$. If it happens that $n = \deg(I_{coeff})$, then using the Karatsuba approach to multiply the coefficients results in a cost of $\mathcal{O}(n^2 \times n^{1.585}) = \mathcal{O}(n^{3.585})$ per iteration, for an overall running time of $\mathcal{O}(n^{3.585} \log b)$. Overall, each iteration becomes a lot more expensive, which will dramatically affect the running time of the Cantor-Zassenhaus and Shoup factoring algorithms with a field extension.

4.8 Cantor Zassenhaus

Recall that the Cantor-Zassenhaus algorithm has three steps:

1. Verify the polynomial is squarefree
2. Apply a distinct degree factorization algorithm
3. Apply the Cantor-Zassenhaus equal degree factorization algorithm to each of the partial factors obtained in step 2

This particular factoring problem involving a field extension, where both irreducibles have the same degree, has the advantage of not requiring the first two steps. In this instance, any inputs are guaranteed to split into a product of distinct linears [2]. If all factors are distinct then clearly no squares are present, and if all factors are linears, then clearly a distinct degree factorization does not produce any relevant information. Instead, it is possible to skip directly to the final step of the Cantor-Zassenhaus equal degree factorization. Factoring with

a field extension effectively means all the coefficients are in a more complicated field. This results in the same basic pseudocode, but the increased complexity affects the running time and space requirements.

Listing 4.1: Cantor-Zassenhaus with Field Extension

```
// Performs a Cantor-Zassenhaus equal degree factorization.
// This version of the algorithm uses a field extension for
// meaning the coefficients are in GF(p^n)
fn cz_factor_fe(poly f, finitefield coeff_field) -> vec<poly> {

  let answers = new vec
  let remaining = f

  for i in 0 to deg(f) {
    if deg(remaining) = 0 {
      break loop
    }

    if deg(remaining) = 1 {
      answers.add(remaining)
      break loop
    }

    let factors = cz_extract_factors_fe(remaining, coeff_field)
    for factor in factors {
      remaining = remainder(remaining / factor)
      answers.add(factor)
    }
  }

  return answers;
}

// Returns at least one factor from the polynomial f
// Coefficients have a field extension so are elements of GF(p^n)
fn cz_extract_factors_fe(poly f, finitefield coeff_field) -> vec<poly> {
  if deg(f) = 1 {
    return f
  }

  loop {
    let b = rand_poly()
    let d = deg(field.irreducible_polynomial)
    let m = (q^d - 1) / 2

    let bm = b^m (modd f(x), p)
```

```
    // These partial factors are not always irreducible
    // If they are not, another application of the algorithm will factor them
    let gcd_zero = gcd(bm, f)
    if gcd_zero is a non-trivial factor {
      return cz_factor_fe(gcd_zero, field)
    }

    let gcd_plus = gcd(bm + 1, f)
    if gcd_plus is a non-trivial factor {
      return cz_factor_fe(gcd_plus, field)
    }

    let gcd_minus = gcd(bm - 1, f)
    if gcd_minus is a non-trivial factor {
      return cz_factor_fe(gcd_minus, field)
    }
  }
}
```

Algorithm Analysis

For irreducibles of degree n, in the worst case, n applications are required in the cz_factor_fe() function. Each of these iterations is comprised of the following set of steps:

- Apply cz_extract_factors_fe()
- Long division

Out of these two, the cz_extract_factors_fe() function should be more expensive because it contains a powmod operation. The running time in the worst case should therefore be based on n applications of the cz_extract_factors_fe() function. The cz_extract_factors_fe() function computes the following:

- A random polynomial
- A powmod operation
- Three GCD computations

Generating a random polynomial of this type is a $\mathcal{O}(n^2)$ operation, gcd is a $\mathcal{O}(n^{3.585})$ operation and powmod is a $\mathcal{O}(n^{3.585} \log b)$, when using Karatsuba multiplication. The dominating term here is always the powmod operation. The ordering $n^{3.585} \log b \geq n^{3.585} \geq n^2$ follows from the fact that b is a prime and therefore must have a minimum value of 2 ($\log_2(2) = 1$). With good luck, every attempt to factor the polynomial will split it into two polynomials of approximately equal degree. With bad luck, the split results in polynomials of degree $(n-1)$ and 1. This means the worst possible case is attempting to factor a degree n polynomial, then a degree $(n-1)$ polynomial, etc until eventually reaching a linear. This results in a final running time of $\mathcal{O}(n^{4.585} \log b)$ when using Karatsuba multiplication. Here is an example of the running times using Karatsuba multiplication:

Table 4.4: Cantor-Zassenhaus with Field Extension Running Times

n	time (seconds)	Karatsuba / $n^{4.585} \log b$
2	0.0033	0.4763
4	0.0422	0.1262
6	0.7501	0.2335
8	1.5291	0.0956
10	8.2592	0.1486
12	24.6982	0.1607
14	52.7769	0.1453
16	91.9447	0.1201
18	175.4302	0.1188
20	389.8996	0.1467
22	569.1273	0.1258
24	1115.4872	0.1517
26	993.9954	0.0865
28	1626.8295	0.0936
30	2285.5376	0.0895

The ratio of actual running time to theoretical running time varies a bit with problem size but is staying around 0.10 to 0.15 for most problems. The values are close enough to be reasonable for a non-deterministic algorithm.

4.9 Shoup

The Shoup factoring algorithm can be modified in a similar way to the Cantor-Zassenhaus factoring algorithm when using a field extension. Ordinarily, it follows the same three basic steps as Cantor-Zassenhaus:

1. Verify the polynomial is squarefree
2. Apply a distinct degree factorization algorithm
3. Apply the Shoup equal degree factorization algorithm to each of the partial factors obtained in step 2

This is the same factoring problem as Cantor-Zassenhaus, meaning the first two steps are, again, always true for every valid input to the problem. The bulk of

the effort can consequently be focused on the equal degree factorization step. With this in mind, the Shoup algorithm can now be modified to handle all of the coefficients in the more complicated $GF(p^n)$ finite field.

```
fn shoup_fe(poly f, finitefield coeff_field) -> vec<poly> {

  if deg(f) = d {
    return f
  }

  let n = deg(f)
  let g = rand_poly()

  let h = Td(g)

  loop {
    let b = rand_poly() with degree 0
    let d = deg(coeff_field.irreducible)
    let power = (p^d - 1)/2
    let a = ((b+h)^power - 1) (modd f(x), p)

    if a is a constant {
      continue // try again
    }

    let v = gcd(a, f)
    if v is a trivial factor {
      continue // try again
    }

    let part1 = shoup_fe(v, coeff_field)
    let part2 = shoup_fe(f / v, coeff_field)

    return join(part1, part2)
  }
}
```

Algorithm Analysis

The running time of this algorithm is similar to Cantor-Zassenhaus because each application of the factoring algorithm performs a similar set of steps:

1. Generate a random polynomial

2. Compute powmod

3. Compute polynomial gcd

Generating a random polynomial here is a $\mathcal{O}(n)$ operation, gcd is a $\mathcal{O}(n^{3.585})$ operation and powmod is a $\mathcal{O}(n^{3.585} \log b)$ operation when using Karatsuba multiplication. Again, the dominating term here is the powmod operation based on the fact that if b is a prime, then it must be that $b \geq 2$. Each application therefore has an expected running time of $\mathcal{O}(n^{3.585} \log b)$. Ideally, the factoring algorithm splits the polynomial into two partial factors of equal degree, however with sufficiently bad luck it can produce factors of degree $(n-1)$ and 1. As a result, n applications of the factoring algorithm may be required, giving a running time of $\mathcal{O}(n^{4.585} \log b)$. There is some randomness involved in the actual running time due to the choices for b, but overall the running times are similar between the two algorithms.

n	time (seconds)	time / $n^{4.585} \log b$
2	0.0051	0.0142
4	0.0432	0.0051
6	0.4050	0.0074
8	1.5205	0.0075
10	5.4382	0.0096
12	17.5132	0.0135
14	24.5549	0.0093
16	55.4130	0.0114
18	106.2590	0.0127
20	165.3311	0.0122
22	277.5472	0.0132
24	452.0602	0.0145
26	645.1081	0.0143
28	994.5089	0.0157
30	1301.1410	0.0150
32	1754.3969	0.0150
34	2487.8228	0.0161

Table 4.5: Shoup with Field Extension Running Times

4.10 Summary of Results

In this chapter, two approaches to factoring with a field extension were developed alongside an analysis of their running times. The first approach is a modified version of Cantor-Zassenhaus while the second approach is a modified version of Shoup's algorithm. This problem is implemented in at least one vendor language, Maple, but their approach is not documented. The algorithms in this chapter are the author's approach to this problem that is otherwise poorly covered in the literature.

Chapter 5

Cantor-Zassenhaus Arbitrary Polynomial Factoring Algorithm

A novel application of the Cantor-Zassenhaus factoring algorithm is that it can be used to factor arbitrary polynomials in a finite field without an accompanying distinct degree factorization algorithm. In this case, an arbitrary polynomial is one with irreducible factors of various degrees. To do this, we need the function to render a polynomial squarefree as well as the cz_factor() function, both presented in the background section. These functions can be used to compose the following algorithm:

```
fn cz_arbitrary(poly f, int p, int max_attempts) -> vec<poly>{
  let remainder = f
  let factors = new vec
  let max_factor_degree = deg(f) - 1

  for d in 1 to max_factor_degree {
    if deg(remainder) < d {
      if deg(remainder) >= 1 {
        factors.add(remainder)
      }
      return factors
    }

    // try max_attempts times to extract a factor of
    // that degree
    let partial_factors = cz_factor(remainder, d, p,
            max_attempts)

    for factor in partial_factors {
      remainder = remainder / factor
      factors.add(factor)
    }

    if partial_factors.size() > 0 {
      continue with same d
    }
  }

  return factors
}
```

This algorithm works because, even though all the factors are not the same degree, the polynomial f still satisfies all the requirements to extract a factor using the Cantor-Zassenhaus algorithm. If a factor exists of degree d the algorithm will still find it with the same high probability as before.

5.1 Summary of Results

In this chapter, an approach to factoring with Cantor-Zassenhaus that does not require a distinct degree factorization algorithm was presented. This algorithm is a unique approach to factoring univariate polynomials found by the author while developing the algorithm to factor with a field extension.

Chapter 6

Conclusions and Future Work

Building a cryptographic system, particularly using finite field extensions, involves many algorithms. Some of these algorithms are the fundamental building blocks of polynomial operations, while others build on these fundamental operations to solve more complicated problems. The most important algorithms covered are summarized in the following table:

Algorithm	Running Time	Space Requirements
Polynomial Long Division	$\mathcal{O}(n(m-n))$	$\mathcal{O}(MAX(m,n))$
Precomputed Powers	$\mathcal{O}(nm)$	$\mathcal{O}(n)$ or $\mathcal{O}(nr)$
Polynomial GCD	$\mathcal{O}(mn(m-n))$	$\mathcal{O}(m)$
Polynomial Inverse	$\mathcal{O}(m(M(n,m)+n(m-n)))$	$\mathcal{O}(m)$
Polynomial Addition	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Naive Multiply	$\mathcal{O}(n^2)$	$\mathcal{O}(n+m)$
Karatsuba Multiply	$\mathcal{O}(n^{log_2(3)})$	$\mathcal{O}(n)$
FFT Multiply	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$
Powmod	$\mathcal{O}(M(n) \log b)$	$\mathcal{O}(n)$
Berlekamp Factor	$\mathcal{O}(n^3 + M(n) \log(n) \log(p))$	$\mathcal{O}(n^2)$
Cantor-Zassenhaus Factor	$\mathcal{O}(n^3 + n^2 \log q)$	$\mathcal{O}(n)$
Shoup Factor	$\mathcal{O}(n^2 \log n + M(n) \log n \log p)$	$\mathcal{O}(n^{1.5})$

The main contribution of this thesis is taking the above algorithms and modifying them to solve the problem of factoring with a field extension. This particular problem is poorly covered in the literature. In addition to the problem of factoring with a field extension, this thesis also describes a novel way to factor an arbitrary polynomial using just the Cantor-Zassenhaus equal degree factorization algorithm. This approach may be useful if a distinct degree factorization algorithm is not available. The final running times of the modified Cantor-Zassenhaus and Shoup factoring algorithm implementations covered here are:

Algorithm	Running Time
Cantor-Zassenhaus Field Extension	$\mathcal{O}(n^{4.585} \log b)$
Shoup Field Extension	$\mathcal{O}(n^{4.585} \log b)$

This solves the desired problem of factoring with a field extension.

Future Work

Based on the performance analyses in this thesis, the running time of the algorithms to factor with a field extension are dominated by the powmod function. Further improvements to the computation of powmods would lead to a better overall running time for these algorithms. These algorithms have also been used in the development of a new approach to public key encryption, currently under review for publication. The scope and details of the cryptosystem are beyond the content of this thesis.

Bibliography

[1] Jim Blandy and Jason Orendorff. *Programming Rust.* O'Reilly, 1st edition, 2018.

[2] Rudolf Lidl and Harald Niederreiter. *Finite Fields.* Cambridge University Press, 2nd edition, 1997.

[3] Donald Knuth. *The Art of Computer Programming: Seminumerical Algorithms, 3rd ed.* Addison-Wesley, 2002.

[4] Thomas H Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, 3rd edition, 2009.

[5] Joachim von zur Gathen. *Modern Computer Algebra.* Cambridge University Press, 3rd edition, 2013.

[6] Paul E. Black. *Repeated Squaring.* In Dictionary of Algorithms and Data Structures [online]. 12 June 2013. Accessed on December 30, 2019. Available from: `https://www.nist.gov/dads/HTML/repeatedSquaring.html`.

[7] Alessandro De Piccoli, Andrea Visconti, and Ottavio Rizzo. Polynomial multiplication over binary finite fields: new upper bounds. *Journal of Cryptographic Engineering*, 2019.

[8] David Harvey. The karatsuba middle product for integers. *Journal of Symbolic Computation*, 47:954–967, 2012.

[9] André Weimerskirch and Christof Paar. Generalizations of the karatsuba algorithm for polynomial multiplication. *IACR Cryptology ePrint Archive*, 2006.

[10] Shahram Jahani Azman Samsudin and Kumbakonam Govindarajan Subramanian. Efficient big integer multiplication and squaring algorithms for cryptographic applications. *Journal of Applied Mathematics*, 6:1–9, 2004.

[11] Ling Ding and Éric Schost. Code generation for polynomial multiplication. In *Proceedings of the International Workshop on Computer Algebra in Scientific Computing*, pages 66–78, 2009.

[12] Timothy Sauer. *Numerical Analysis, 2nd ed.* Pearson, 2011.

[13] Robert Moenck. Practical fast polynomial multiplication. In *Proceedings of the third ACM symposium on Symbolic and Algebraic Computation*, pages 136–148, 1976.

[14] Bruno Guerrieri. *Polynomials Multiplication Using the Fast Fourier Transform*. Maplesoft documentation [online]. Accessed on January 15, 2020. Available from: `https://www.maplesoft.com/applications/view.aspx?SID=3446&view=html`.

[15] Aiswarya Prakasan. *Understanding Fast Fourier Transform from scratch — to solve Polynomial Multiplication*. Published by Medium. February 2017. Accessed on December 16, 2019. `https://medium.com/@aiswaryamathur/understanding-fast-fourier-transform-from-scratch-to-solve-polynomial-multiplication-8018d511162f`.

[16] Jose Divasón, Sebastiaan Joosten, René Thiemann, and Akihisa Yamada. A formalization of the berlekamp-zassenhaus factorization algorithm. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 17–29, 2017.

[17] E. R. Berlekamp. Factoring polynomials over finite fields. *The Bell System Technical Journal*, 46(8):1853–1859, 1967.

[18] Victor Shoup. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation*, 20(4):363–397, 1995.

[19] Michael Rabin. Probabilistic algorithms in finite fields. *SIAM J. COMPUT*, 9:8 pages, 1980.

[20] Joachim Von Zur Gathen and Daniel Panario. Factoring polynomials over finite fields: A survey. *J Symbolic Computation*, 31:3–17, 2001.

[21] David Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(154), 1981.

[22] Victor Shoup. Factoring polynomials over finite fields: Asymptotic complexity vs. reality. In *Proceedings of the IMACS Symposium*, pages 124–129, 1993.

[23] Daniel J. Bernstein. Multidigit multiplication for mathematicians. 09 2001.

[24] Robert T Moenck. Practical fast polynomial multiplication. In *Proceedings of the 1976 ACM symposium on Symbolic and Algebraic Computation*, pages 136–148, 1976.

[25] Joachim Von Zur Gathen and Daniel Panario. Factoring polynomials over finite fields: A survey. *J. Symbolic Computation*, 31:3–17, 2001.

[26] D. G. Cantor. On arithmetical algorithms over finite fields. *J. Comb. Theory*, 50:285–300, 1989.

[27] G. E. Collins. Factoring univariate integral polynomials in polynomial average time. *Proceedings of EUROSAM '79, Marseille, France*, 72:317–329, 1979.

[28] P. Fleischmann and P. Roelse. Comparative implementations of berlekamp's and niederreiter's polynomial factorization algorithms. *Finite Fields and Applications, Proceedings of the Third International Conference, Glasgow*, 233:73–83, 1996.

[29] Victor Shoup. Factoring polynomials over finite fields: asymptotic complexity vs. reality. *Proc. Int. IMACS Symp. on Symbolic Computation, New Trends and Developments, Lille, France*, 17:124–129, 1993.

Vita

Candidate's full name: Alec Sobeck

University attended (with dates and degrees obtained):

- Bachelor of Computer Science, UNB, 2013-2017
- Masters of Computer Science (In Progress), UNB, 2018-2020

Publications: None

Conference Presentations: None