Efficient Polynomial Multiplication Algorithms

William Cashman

October 2020

An Honours thesis submitted for the degree of Bachelor of Philosophy (Honours) - Science of the Australian National University

Australian National University

Declaration

The work in this thesis is my own except where otherwise stated.

William Cashman

Contents

Chapter 1

Introduction

Polynomial multiplication is a fundamental problem in computational mathematics. Not only does it have numerous practical applications, but it can be generalised to a broader class of problems such as integers multiplication, calculating the Discrete Fourier Transform and evaluating convolutions. Despite being widely used, the first significant algorithmic advancement was in 1962 by Karatsuba [1] who presented his discovery a week after attending a seminar by Kolmogorov who conjectured that no such improvement was possible. This development, combined with the increasing demand for efficient digital processing, quickly saw a pique of interest in the mathematical community with several subsequent multiplication algorithms being produced shortly after. Many of the algorithms developed in this 10 year period such as the Toom-Cook algorithm, Rader's trick, and FFT based algorithms, are still popular today. As the computational sciences sought to understand increasingly complex systems, there was a need to multiply larger polynomials for fundamental problems such as solving systems of polynomial equations, computing Gröbner bases, and evaluating convolutions.

The focus of this thesis is on studying the most significant advancements in the field, starting from the first improvement over the schoolbook method by Karatsuba, and ending with the recent improvement by Harvey and van der Hoeven [27]. There are many variables that can affect the efficiency and validity of polynomial multiplication algorithms such as the number of indeterminates, the coefficient algebra, and the number of non-zero terms. The two variable of most interest to us are sparsity and the coefficient algebra of the polynomials. Analysis of the algorithms will be performed within two different contexts, namely their practical performance for a suitable range of inputs, and their asymptotic

nature as the size of the inputs grows increasingly large. Along with this thesis we have implemented several of these algorithms as part of our nPoly polynomial library which can be found at https://github.com/willcsm/nPoly. We provide a commentary on key implementation details on these algorithms and have conducted empirical tests to support the theoretical results.

Due to their simplicity and generality, most computer algebra systems still rely on old multiplication schemes which work across a broad range of inputs at the cost of performance. A minor goal of this thesis is to analyse the practical complexity of such methods to develop a heuristic for computer algebra systems to select the most efficient algorithm for the given inputs.

1.1 Structure of Thesis

Before we can begin analysing algorithms, we first need to formalise the computational model our algorithms are executed in. Since there is no ubiquitous model, Chapter 2 provides a mathematical refresher and summary of necessary complexity-theoretic definitions and notation. In particular, we present a brief introduction to the two computation models we will be using throughout this thesis; namely the multi-tape Turing machine and an adaptation of the Random Access Machine (RAM) for rings.

Chapter 3 gives an overview of several important classical algorithms. These are currently the most widely implemented algorithms due to their simplicity, generality, and efficiency for polynomials of small to moderate degree sizes. Popular computer algebra systems such as Macaulay2 [33], Maxima [34], NTL [36], and Magma [15] all use at least one of these algorithms.

Chapter 4 looks at the *evaluation-interpolation strategy* for multiplying polynomials, and shows how algorithms such as Karatsuba's from Chapter 3 can be re-expressed in this framework. We will provide an introduction to the Discrete Fourier Transform (DFT) and show how it can be used to evaluate convolutions, followed by a theorem showing how we may apply the Cooley-Tukey Fast Fourier Transform (FFT) to multiply polynomials in $O(n \log n)$ ring operations, which provides the foundations for the popular class of FFT-based algorithms. We finish with a generalisation of Schönage and Strassen's integer multiplication algorithm for polynomials over commutative rings. The FFT-based approach marks the beginning of the more exotic algorithms which go beyond the coefficient-agnostic transformations in Chapter 3, and which are often only used in specialised appli-

cations where the degree of the polynomials are large.

Chapter 5 looks at algorithms for polynomials in integer rings. Multiplications in this ring circumvents the main shortcoming associated with algebras from previous sections, namely the increasing computational cost of ring operations in unbounded algebras (e.g. $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Z}$). This enable our algorithms to be optimised an improved practical complexity. This area is of particular interest when computing polynomials of large degree in $\mathbb{Z}[x]$ [21][19], and other areas optimised for speed such as cryptography and signal processing. Efficiency is achieved by mapping our inputs into appropriately chosen rings which admit efficient Number Theoretic Transforms (NTT); a generalisation of the FFT for integer rings.

Chapter 6 looks at the most recent advancement in the theoretical complexity of polynomial multiplication, that is, Harvey and var der Hoeven's $O(n \log n)$ integer multiplication algorithm. Despite the algorithm [27] being formulated for integers, we will demonstrate how this result can be directly applied to the problem of polynomial multiplication using the techniques from Chapter 3. The theoretical bound is achieved by constructing a multi-dimensional convolution that controls the error in its finite precision arithmetic. We will also present another algorithm by Harvey and van der Hoeven that achieves the same bound but is conditional on an unproven hypothesis. It takes a more intuitive approach which naturally generalises the result for finite fields.

Chapter 7 will introduce an algorithm for sparse polynomial multiplication. This is particularly useful in modern computer algebra applications and situations involving multivariate polynomials. We also present an empirical analysis of the performance of several of the algorithms in this thesis in our nPoly polynomial library[35].

Chapter 2

Preliminaries

In order to formalise the analysis of algorithms, we first need to specify a *computation model* to define the valid operations we can perform, and a broader memory model. In this thesis, we will be using the Turing machine model and Random Access Machine (RAM) model for commutative rings.

There are numerous variations in this type of analysis that are of interest in the field of complexity theory, however, we will not consider them here as we are primarily focused on the mathematical techniques employed in the algorithms under study. For a rigorous formalisation of these variants, we refer the reader to Chapter 4 of [16].

2.1 Computational Models

A *computation model* is a framework which is comprised of a *machine type* and a *space measure*. The machine type specifies the operations that can be performed on given data and their associated computational costs e.g. Turing machine, counting machine. The space measure specifies how we may coordinate accessing data e.g. single-tape vs multi-tape. The goal of computational complexity theory is to develop algorithms that can take a range of inputs and produce a desired result using only the operations permitted by the computation model such that the total computational cost is minimised.

The two most popular computation models are the *Turing Machine*, and the *Random Access Machine* (RAM). Either can emulate the other in polynomial time [5], although the Turing machine's complexity is often larger as the machine can only operate locally on individual symbols from a predefined alphabet. This is in contrast to the standard Random Access Machine, which operates on unbounded

natural numbers and is able to access data stored far away from the previous data that was processed in constant time (known as *indirect addressing*)[1] which is not the case in the Turing model.

The single-tape Turing machine simulates a mechanical automaton operating on an infinite tape. The tape is divided into cells which can either be empty or contain a symbol from a predefined alphabet. The machine has a set of internal states, of which it can only be in one at any given time.
The machine starts at an initial internal state and position in the tape and reads the symbol from the cell directly beneath it. It contains a set of instructions which specify that depending on the symbol beneath it and its current state, it can choose to:

- Erase or write a symbol.
- Move the machine one place to the left or right along the tape, or remain in the same position.
- Assume a different internal state.

The multi-tape Turing model has access to multiple tapes at once. Though any algorithm on a multi-tape model can be emulated by single-tape model in quadratic time[30].

The Random Access Machine (RAM) model more closely emulates a modern computer for small to moderate input sizes. It contains a set of registers which can each hold an unbounded natural number. The machine operates by reading a list of instructions to manipulate the numbers stored inside the registers. We will consider two different machine types in this model, namely the standard type that operations on unbounded natural numbers which we will refer to as the "bit oriented-model", and another which operations on elements of a ring. The cost of addition and multiplication by powers of two in the bit model is linearly proportional to the size of the integers in bits. In the "ring-oriented model", the goal is to minimise the number of ring operations required by the algorithm (we will still say an algorithm is "faster" than another, to indicate it uses fewer ring operations)

[1] In Cook's original formulation of the RAM model, he described a function $l(X)$ (informally) as being the cost of storing a number X into one of the machine's registers. From this he derived the cost of addition, subtraction, and indirection. At the time, Cook left the definition of l unspecified, but suggested two obvious choices of l: either constant or $\log |X|$. Nowadays it is convention to take $l(X)$ to be constant.

The RAM model is adequate for most complexity analysis concerning practical algorithms, and so we will use this model to simplify our analysis of the classical algorithms and the algorithms designed for improved practical efficiency. However, when calculations become increasingly large, the RAM model becomes less a realistic representation of how computers fundamentally operate. Therefore in Chapter 6 when we present algorithms for asymptotically superior integer multiplication, we will adopt the Turing model.

2.2 Practical and Theoretical Complexity

Once we have defined the costs of the various operations, we can begin to analyse the complexity of our algorithms. There are two paradigms we use to evaluate the efficiency of algorithms in, namely their *practical complexity* and their *theoretical complexity*. In this thesis, the practical complexity of an algorithm refers to the precise number of computations performed in the computation model. The theoretical complexity refers to the asymptotic nature of the algorithm, and is used to study its behavior for large input sizes.

We say that an algorithm has a good *practical complexity* if it is the fastest for a specific range of input parameters and say that it has a good *theoretical complexity* if it is the fastest asymptotically, i.e. for large enough inputs.

An excellent example of this distinction is found between the Karatsuba, Schonage-Strassen (SS), and Harvey-van der Hoeven (H-vdH) integer multiplication algorithms[2]. Considering their asymptotic complexity with respect to the multi-tape Turing model, H-vdH is the fastest, followed by SS, and then Karatsuba. However, SS only becomes faster than Karatsuba when the number of bits in the integers exceed 2^{15} [29]. H-vdH algorithm's base case is at $2^{1729^{12}}$ bits, and so it is the smallest size that the algorithm could (provably) beat the SS algorithm. Despite the H-vdH and SS algorithms having much better theoretical complexity, Karatsuba easily out-performs them both in almost all practical use cases. For this reason, many computer algebra systems, (e.g. Maxima) only implement Karatsuba's algorithm [34].

Definition 1 (Big-O Notation)**.** For a non-negative functions $g : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, we write $\mathcal{O}(g)$ to denote the set of all $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ where there exists an $n_0 \geq 0$

[2]Though it is unclear now, Chapter 2 shows how this is directly applicable to polynomial multiplication

and $c > 0$ such that

$$f(n) \leq cg(n) \qquad \forall n \geq n_0. \tag{2.1}$$

Hence for all n large enough, g is a constant factor away from dominating f. Informally this means that g grows asymptotically faster than or at the same rate as f. The requirement: $n_0 \geq 0$, is necessary to ignore the behaviour of the algorithms for small inputs.

If we let $\mathcal{C}(n)$ denote the computational cost of an algorithm, then it is a common problem when analysing algorithms to find the slowest growing function g such that $\mathcal{C} \in \mathcal{O}(g)$, thus obtaining the tightest bound on its complexity.

It is convention that when $\mathcal{O}(f)$ is used in an expression, $\mathcal{O}(f)$ can be replaced by any representative in the class. This is commonly seen in recursive formulations of the computation cost of algorithms using the divide-and-conquer strategy. For instance, suppose that we are given a problem of size n, and the algorithm recursively calls itself on two subproblems of size $n/2$ and combines the results of the subproblems in linear time. Rather than introducing another function to denote the recombination step we may simply write

$$\mathcal{C}(n) = 2\mathcal{C}(n/2) + \mathcal{O}(n),$$

with the understanding that $\mathcal{O}(n)$ may be replaces by pN for any positive constant $p > 0$.

Though big-O notation is defined for functions, it is more common in this kind of analysis to classify functions in a set standard asymptotic complexity classes. In these situations we will abuse notation and write statements such as $f \in \mathcal{O}(n^2)$ to denote the fact that f grows at the same rate or slower than the function $g(n) = n^2$. This convention is used to avoid unnecessarily naming functions[3].

An immediate consequence of the definition of big-O notation is $f \in \mathcal{O}(g+h)$ whenever $f \in \mathcal{O}(g)$ and h does not grow asymptotically faster than f, for example $n^2 \in \mathcal{O}(n^2+n)$. Another fact that is easily derived is: $\log n \in \mathcal{O}(n^\epsilon)$ for all $\epsilon > 0$; hence any polynomial function with a positive exponent grows faster than any logarithmic function. From this we can then derive the statement that $n^k \in O(b^n)$ for any constants $k \in \mathbb{R}$ and $b > 1$. We often do not write an explicit base for logarithms as logarithms with different bases are a constant multiple of the other and thus are in the same asymptotic complexity class.

[3]It computer science it is convention to write the statement $f \in \mathcal{O}(g)$ as $f(n) = \mathcal{O}(g(n))$. Our abuse of notation is quite tame in comparison.

2.3 Polynomial Representations

The representations of polynomials inside a program can significantly affect both the practical and theoretical complexity of algorithms. This section is not essential for chapters up to Chapter 7, though it is useful for understanding the impact that sparsity and the number of indeterminates can have on the performance of an algorithm.

In computer algebra systems, there are three commonly used representations of polynomials:

Coefficient vectors Stores only the coefficients of the (expanded) polynomial. For instance, $2 + x + 3x^3 + x^5$ is stored as $(X)(2\ 1\ 0\ 1\ 0\ 1)$.

We can also store multivariate polynomials such as $2xy - x^2 + 3y$ by taking the maximum degree of the two variables, which is 2 for x and 1 for y and expanding over all possibilities as

$$2xy - x^2 + 3y = \ 0 + 3y + 0x + 2xy + -x^2 + 0x^2y$$

is stored as $(XY)(0\ 3\ 0\ 2\ -1\ 0)$.

Sparse vectors Stores a vector of tuples containing the coefficients and the monomials of the polynomials. As an example, we could store $2xy^2 - x^2 + 2x + 5y^2 + 17x^2y$ as a sparse vector,

$$(X\ Y)\ (2\ 1\ 2)\ (-1\ 2\ 0)\ (2\ 1\ 0)\ (5\ 0\ 2)\ (17\ 2\ 1).$$

where the term $cx^\alpha y^\beta$ is stored as $(c\ \alpha\ \beta)$ and terms are store contiguously.

Canonical Rational Expression (CRE) Here we use the isomorphism $R[x, y] \cong R[x][y]$ to represent polynomials recursively by using one of the other techniques above to represent a multivariate polynomial as a univariate polynomial whose coefficients are also polynomials. For example, the polynomial $2xy^2 - x^2 + 2x + 5y^2 + 17x^2y$ can be reorganised as

$$(-1 + 17y)x^2 + (2 + 2y^2)x + 5y^2$$

and can then be encoded as

$$X; (; Y; (-; 1; 0); (17; 1);); (; Y; (2; 0); (2; 2);); (; Y; (5; 2););$$

Note that this is an idealised form and that in practice different algebra systems will vary the representations to suit their purposes.

Throughout this paper we will assume that polynomial are stored in their coefficient vector format, as we will be proving the worse case complexity for these algorithms which occurs when the polynomials are dense.

CREs may see like the canonical representation due to its simplicity and ability to recursively apply algorithms for univariate polynomials to multivariate polynomials. However, it is not very efficient in practice due to the large number of expensive memory operations that need to be performed to access its elements. Since the information of each monomial is distributed over multiple locations in memory, certain operations are unnecessarily costly, such as extracting the lead term of the polynomial. Furthermore, it enforces a lex monomial ordering onto the terms of the polynomial, which can cause conflicts with Gröbner basis calculations that involve other monomial orderings. Modern computer algebra systems such as Maple and Macaulay2, now tend to prefer the sparse term representation [24][33].

It is worth mentioning there is another representation where the monomials are stored in a hash map. It is primarily used when multiplying sparse polynomials as we don't need to be as careful when managing our memory and we can completely bypass the more complicated algorithm for sparse multiplication in Chapter 7, at the cost of memory efficiency. However since the monomials are not stored in any kind of order, its applications are more limited.

The *sparsity* of a polynomial refers to the number of non-zero terms with respect to the polynomial's degree in each variable, or rather, the ratio of the number of zero terms to non-zero terms in its coefficient vector. The steps a program executes is often solely dependent on the maximum degrees of the input polynomials and so even if the inputs are sparse, it is often more efficient to store it in its coefficient vector representation and padded with zeros, which can cause them to be terrifically inefficient. This is the case for a popular class of algorithms we will introduce in Chapter 4 known as Fast Fourier Transform-based algorithms. This is often worse for multivariate polynomials as they are almost always sparse in practice. For example, consider multiplying the polynomial $x^2y + y^6z^4$ by $xyz + x^3z^5$ with the FFT. We can see that the maximum possible degrees of the x, y and z variables in the result are 5, 7 and 9 respectively. We also then need to round the numbers up to the nearest power of two to obtain 8, 8 and 16. Therefore in order to use an FFT-based algorithm, we would need to perform the FFT on $8 \times 8 \times 16 = 1024$ terms. Hence, FFT-based algorithms

perform poorly for many moderate-sized multivariate polynomial multiplication problems. In Chapter 7 we will present a technique that is dependent only on the number of non-zero terms in the inputs, which would be far more suitable for this situation.

In light of this, we will only formulate the complexity of such algorithms as function of the maximum of the degrees of the two inputs in each of the indeterminates. The exception to this, are algorithms for sparse polynomials.

Chapter 3

Classical Algorithms

Here we review the classical algorithms that were the leading methods of their time up until the FFT algorithm's introduction in the late 1960s. Despite having poorer theoretical complexities compared to their modern counterparts, they often have a far superior practical complexity for smaller input sizes, whilst also having the advantage of being independent of the coefficient algebra. For these reasons, they are still the most widely implemented algorithms in modern computer algebra systems.

Let K be a commutative ring. Then let $f, g \in K[x]$ denote two polynomials

$$f(x) = \sum_{i=0}^{n} f_i x^i,$$
$$g(x) = \sum_{i=0}^{n} g_i x^i.$$

Their product is defined as

$$ab = \left(\sum_{i=0}^{n} a_i x^i\right)\left(\sum_{j=0}^{m} b_j x^j\right) = \sum_{i=0}^{n} a_i x^i \left(\sum_{j=0}^{m} b_j x^j\right), \tag{3.1}$$

or alternatively as

$$ab = \sum_{k=0}^{n+m} \left(\sum_{i+j=k} a_i b_j\right) x^{i+j}. \tag{3.2}$$

Some algorithms naturally lends themselves to one formulation more than the other, so it is good to bear both in mind.

3.1 Schoolbook Multiplication

In the schoolbook method of polynomial multiplication, we evaluate the expression (3.1) by calculating each term in the outer sum individually as it is written.

We can see the inner sum requires $\mathcal{O}(m)$ ring operations. To evaluate the outer sum, we must compute the inner sum n times; thus in total this algorithm requires at most $\mathcal{O}(nm)$ ring operations.

Though it has the worse theoretical complexity of all the algorithms in this paper, it is valid over any coefficient algebra, and most implementations out-perform other algorithms for inputs with degree less than 10. In Chapter 7 we will present a multiplication algorithm for sparse polynomials which removes the dependency on the degrees of the polynomials and has complexity $\mathcal{O}(\#f\#g \log(\min\{\#f, \#g\}))$ where $\#f$ and $\#g$ are the number of non-zero terms in polynomials f and g respectively.

Notice that this is very similar to the schoolbook method for integer multiplication. Suppose we have an integer $a_n \cdots a_1$ where a_i is the i^{th} digit in the decimal representation; in other words

$$a_n \dots a_1 = \sum_{i=1}^{n} a_i 10^{i-1}.$$

To calculate the product $c = a \cdot b$, we could first convert them into polynomials via the substitution $x = 10$

$$a(x) = \sum_{i=1}^{n} a_i x^i, \qquad b(x) = \sum_{i=1}^{m} b_i x^i,$$

We can then use techniques for multiplying polynomials in $\mathbb{Z}[x]$ to first calculate $c(x) = a(x)b(x)$ and then evaluate at $x = 10$ to recover the solution $c(10) = a(10)b(10) = a_n \dots a_1 \cdot b_m \dots b_1$. Indeed this is the exact method that is commonly taught in schools, only without the substitution of variables.

However, there is a fundamental difference between integer and polynomial multiplication: in integer multiplication we must "carry" digits, e.g. $5 + 7 = 12$, but $5x + 7x \neq x^2 + 2x$. For this reason, polynomial multiplication is also known as "carry-less" multiplication. This does not present a problem when using polynomials for integer multiplication because the carry operation can always be applied when undoing the substitution. Although the converse is not true; we cannot easily undo a carry to obtain the correct polynomial. 3.4 discusses of popular solution for this case.

3.2 Karatsuba's Algorithm

Karatsuba's algorithm was the first algorithm to surpass the $\mathcal{O}(n^2)$ bound for input polynomials of degree n. Karatsuba (1960) showed his algorithm to Kolmogorov who then went on to present it at various events and even publish a paper crediting Karatsuba in 1962 [1]. Later, Karatsuba would famously write that he only learnt of the paper's existence when it was in its reprints [14]. The algorithm partitions each of the input polynomials into two smaller polynomials and calculates the product recursively by apply the same algorithm to multiply the smaller polynomials; a technique now known as *divide-and-conquer*.

Let $a, b \in K[x]$ be polynomials of degree n. Let $m = \lceil n/2 \rceil$ and $a = a_1 x^m + a_0$ and $b = b_1 x^m + b_0$, where a_0, a_1, b_0, b_1 are polynomials of degree at most $m - 1$. Naturally we have

$$ab = a_1 b_1 x^{2m} + (a_1 b_0 + a_0 b_1) x^m + a_0 b_0. \tag{3.3}$$

Computed naïvely, the above expression requires four multiplications. However, Karatsuba noticed that the expression could be rewritten to use only three multiplications at the cost of an extra addition

$$a_1 b_0 + a_0 b_1 = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0. \tag{3.4}$$

This is a desirable trade since computing addition is, in general, much faster than multiplication.

Theorem 1 (Karatsuba's Algorithm)**.** Let $a, b \in K[x]$ be polynomials of degree at most n, and let $\mathcal{C}_K(n)$ denote the number of ring operations require to compute the product $c = ab$ via Karatsuba's algorithm. Then

$$\mathcal{C}_K(n) \leq 3\mathcal{C}_K(n/2) + \mathcal{O}(n).$$

Moreover, $\mathcal{C}_K \in \mathcal{O}(n^{\log_2 3})$.

Proof. From the formulation above, we can see there are three polynomial multiplications of degree $m = \lceil n/2 \rceil$ and six additions. Computing additions is linear in the size of the polynomials and so the time to compute all six additions can be bounded above by cn for some positive constant $c > 0$. The multiplications are handled recursively with Karatsuba's algorithm, so we obtain the recursive expression for the complexity of the algorithm

$$\mathcal{C}_K(n) = 3\mathcal{C}_K\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn,$$

Since the base case, $n = 1$ is handled with normal ring operations in K we have $\mathcal{C}_K(1) = 1$.

We will use induction to show that $\mathcal{C}_K(n) \leq Mn^{\log_2 3} - 2cn$ for $M = 2c + 1$.

When $n = 1$, we have $\mathcal{C}_K(1) \leq Mn^{\log_2 3} - 2cn$, therefore the base case holds.

Now assume that $\mathcal{C}_K(k) \leq Mk^{\log_2 3} - 2ck$ for all $k < n$. Then

$$\begin{aligned}
\mathcal{C}_K(n) &= 3\mathcal{C}_K\left(\frac{n}{2}\right) + \mathcal{O}(n) \\
&\leq 3(M(\frac{n}{2})^{\log_2 3} - 2c\left(\frac{n}{2}\right)) + cn \\
&= Mn^{\log_2 3} - 3cn + cn \\
&= Mn^{\log_2 3} - 2cn
\end{aligned}$$

Therefore we conclude that $\mathcal{C}_K(n) \leq Mn^{\log_2 3} - 2cn$ for all $n \geq 1$, and therefore $\mathcal{C}_K(n)$ is bounded above by $Mn^{\log_2 3}$, and hence $\mathcal{C}_K \in \mathcal{O}(n^{\log_2 3})$. □

Karatsuba's algorithm remains one of the most practically efficient algorithms for moderate input sizes. Many of the algorithms we cover in this paper have a far better theoretical complexity, but in practice, they only begin to out-perform Karatsuba's algorithm for large inputs. This fact combined with Karatsuba's algorithm being straightforward to implement, means that it remains incredibly popular, and is used in many modern computer algebra systems e.g. Maxima [34].

The Toom-Cook algorithm for polynomial multiplication follows a similar method can be seen as a generalisation of Karatsuba's algorithm. The Toom-k algorithm has complexity $n^{\log_k(2k-1)}$ and so Karatsuba's algorithm is the special case when $k = 2$[1]. We won't give a proper presentation of the algorithm but will briefly show the generalisation when we discussion the connection between Karatsuba's algorithm and the Chinese Remainder theorem in Section 4.1.

3.3 Implementation Remarks

As it has been presented, Karatsuba's algorithm works best on dense polynomials, since the algorithm recursively calls itself proportional to the degree of the polynomial rather than the number of terms; hence the recursion depth is

[1]The GNU Multiple Precision Arithmetic library performs Karatsuba's algorithm at as low as 10 machine words. It then uses variants of the Toom-Cook algorithm from around 80 to 3000 words [32]

logarithmic in the degree of the inputs. Alternatively, rather than splitting of the degree of the polynomials, we could split based on the number of non-zero terms. This way the recursion depth would be $\min\{\log_2 \#f, \log_2 \#g\}$. The minimum comes from the fact that once we recurse to a polynomial of length one, the optimal strategy would then be to multiply the single term with the other polynomial regardless of how many other terms the polynomial has. We could then use the sparse multiplication algorithm in Chapter 7 as a base a base case for the algorithm.

The typical way of performing Karatsuba's method is to recursively call the function on the three subproblems, then combine them together as

$$c_2 x^{2m} + (c_1 - c_0 - c_2)x^m + c_0.$$

When evaluating this expression, one might try to first compute $(c_1 - c_0 - c_2)$, then perform the addition $(c_1 - c_0 - c_2)x^m + c_0$, then compute $c_2 x^{2m}$ and finally $c_2 x^{2m} + (c_1 x^m + c_0)$. However, if we use a coefficient vector, these operations can be done in one pass of a vector of size $3m$, and naturally generalises itself to Single Instruction Multiple Data (SIMD) instructions to obtain more than a 2x speedup (which is done in our implementation of nPoly[35]).

3.4 Kronecker Substitution

Kronecker substitution is a powerful tool for reducing polynomial multiplication problems to simpler multiplications that already admit efficient algorithms (and implementations). It achieves this by grouping together or expanding certain parts of the input polynomials to emulate a polynomial with a different number of indeterminates or coefficient algebra. It is most commonly used to reduce multivariate polynomial multiplication to the univariate case, and univariate polynomial multiplication to the integer case, or vice-versa. In fact both Karatsuba's algorithm and the schoolbook integer multiplication algorithms use a form of Kronecker substitution. We will also show how we can use the $\mathbb{Z}[x]$ multiplication techniques to be applied to polynomials in finite fields.

Note that this transformation is not necessarily an isomorphism of rings. A simple counter example can be seen when we use elements in $\mathbb{Z}$ to multiply polynomials in $\mathbb{Z}[x]$.

3.4.1 Kronecker Substitution for Multivariate polynomials

Let $f(x_1, \ldots, x_n) = \sum_I^n a_{i_1,\ldots,i_n} x_1^{i_1} \cdots x_n^{i_n} \in K[x_1, \ldots, x_n]$ be a multivariate polynomial where I is a multi-index set, and let $d(i)$ be a strict upper bound on the degree of the polynomial in the x_i variable. Then we can transform f into a univariate polynomial $f_u \in K[x]$ via the ring homomorphism

$$\begin{aligned}\phi_f : K[x_1, \ldots, x_n]/x^{d(1)} - 1, \ldots, x^{d(n)} - 1 &\to \frac{K[x]}{x^{d(1)\cdots d(n)} - 1} \\ x_i &\mapsto x^{d(1)\cdots d(i-1)}.\end{aligned}$$

Notice that this depends on the order of the variables $\{x_1, \ldots, x_n\}$. This transformation is equivalent to computing the *mixed radix representation* with respect to the base $[d(1), \ldots, d(n)]$. Note that ϕ_f is not isomorphism unless $d(1), \ldots, d(n)$ are coprime (Lemma 4).

We can show this is a bijection by constructing an inverse map. Suppose we have $x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ which we transform into x^α where $\alpha = \alpha_1 + \alpha_2 d(1) + \cdots + \alpha_n d(1) \cdots d(n-1)$. Then to recover the original monomial, we first divide α by $d(1)$ to return the remainder $r_1 = \alpha_1$ and quotient $q_1 = \alpha_2 + \alpha_3 d(2) + \ldots + \alpha_n d(2) \ldots d(n-1)$. We can then repeat this procedure until no quotient remains. The result is then $x_1^{r_1} \ldots x_n^{r_n} = x_1^{\alpha_1} \ldots x_n^{\alpha_n}$.

Proposition 1. Let $f, g \in K[x_1, \ldots, x_n]$ and $h = fg$. Then let $d_f(i)$, $d_g(i)$ and $d_h(i)$ be the maximum degree of x_i in f, g, and h respectively, hence $d_h(i) = d_f(i) + d_g(i)$. Then using the map ϕ_h

$$\phi_h : x_i \mapsto x^{d_h(1)\cdots d_h(i-1)},$$

we can evaluate fg as

$$fg = \phi_h^{-1}(\phi_h(f)\phi_h(g)).$$

Proof. Let $x_1^{\alpha_1} \ldots x_n^{\alpha_n}$ and $x_1^{\beta_1} \ldots x_n^{\beta_n}$ be monomials in f and g respectively. Then

$$\begin{aligned}\phi_h^{-1}(\phi_h(x_1^{\alpha_1} \ldots x_n^{\alpha_n}) \cdot \phi_h(x_1^{\beta_1} \ldots x_n^{\beta_n})) &= \phi_h^{-1}(x^{\alpha_1 + \cdots + \alpha_n d(1)\ldots d(n-1)} \cdot x^{\beta_1 + \cdots + \beta_n d(1)\ldots d(n-1)}) \\ &= \phi_h^{-1}(x^{(\alpha_1+\beta_1) + \cdots + (\alpha_n+\beta_n) d'(1)\ldots d'(n-1)}) \\ &= x_1^{\alpha_1+\beta_1} \ldots x_n^{\alpha_n+\beta_n}.\end{aligned}$$

Since $\alpha_i < d_f(i)$ and $\beta_i < d_g(i)$ for all $i \leq n$, $\alpha + \beta < d_h(i)$, we are able to correctly recover the coefficients in the last line.

Since any two monomials in f and g are multiplied correctly, we conclude

$$fg = \phi_h^{-1}(\phi_h(f)\phi_h(g)).$$

□

This map can be used to multiply multivariate polynomials using univariate multiplication. Let $M_K(n)$ denote the complexity of multiplying univariate polynomials in $K[x]$, and let $f, g \in K[x_1, \ldots, x_n]$ with $h = fg$. Then $M_K(\deg_h(1), \ldots, \deg_h(n)) = M_K(\deg_h(1) \cdots \deg_h(n)) + \mathcal{O}(\deg_h(1) \cdots \deg_h(n)$. Where the first term accounts for the univariate multiplications, and the second term is for reorganising the polynomial data.

3.4.2 Implementation Remarks

If the polynomial is not too sparse and we wish to have the smallest possible value for the $d(i)$, then instead of performing the conversion directly on each monomial, it would be beneficial to subtract adjacent monomials and use this value to calculate the conversion more efficiently. We would then iterate through the list and add the difference to an accumulator variable, then output the accumulator variable at that step as the transformed value. This would reduce the number of operations need to compute the transform.

We could use a sparse vector representation, in which case we have $\#f$ monomials in the output and so and the problem is reduced to multiplying univariate polynomials with $\#f$ non-zero terms. Note further that if we round each of the $d(i)$ up to the nearest power of two, then we do not need to transform the indices. This is because in polynomial multiplication we only need to multiply monomials together, hence computationally, we only need to add their degrees together. Suppose we have two multi-indices $(\alpha_1, \ldots, \alpha_n)$ and $(\beta_1, \ldots, \beta_n)$ stored in memory where the i^{th} index uses $d_h(i)$ bits. Then we may perform word additions operation as usual on these arrays. We don't need to worry about each of the elements overflowing since $\alpha_i + \beta_i < d_h(i)$ for all $i \leq n$. We can also compare these using a reverse lexicographical ordering to get the same monomial ordering as if we had transformed the monomials into univariate monomials. Thus there is no addition memory reorganisation cost when using coefficient vectors. The only drawback is that this will potentially use n more bits of memory for each monomial; though this isn't usually a problem for modern computers. Since the division operation is typically computationally expensive on most computer architectures [26] this would be quite a desirable optimisation.

3.4.3 Using Integers to Multiply Polynomials

Kronecker substitution may also be used to multiply integer using polynomials in $\mathbb{Z}[x]$ and vice-versa. As mentioned in Section 3.1, the carry operation makes it more difficult to multiply polynomials using integer multiplication than the converse. The trick is to evaluate the polynomials at a value large enough such that no carrying occurs in the multiplication process. This technique is used in the Magma computer algebra system to allow it is to use the optimised integer multiplication algorithm [15].

Suppose we have $f, g \in \mathbb{Z}[x]$ and for now lets assume f and g have positive coefficients. Then let B be an absolute bound on the coefficients of the inputs and let 2^ℓ be the smallest power of two greater nB^2. Then evaluating the polynomials at 2^ℓ and multiplying the two resulting integers together gives

$$f(2^\ell)g(2^\ell) = \sum_{i=0}^{n+m} \left(\sum_{j+k=i} a_j b_k \right) 2^{\ell i}.$$

Since $\sum_{j+k=i} a_j b_k, \leq 2^\ell$ by our choice of 2^ℓ no carries can occur; thus we can recover the solution by undoing the substitution $2^\ell = x$. For the case when the coefficients are negative, we may use a "balanced representation" for our coefficients where the number series goes from $-2^\ell, -2^\ell+1, \cdots, -1, 0, 1, \ldots, 2^\ell-1$. This is also known as the "twos-complement" representation. In this case we need to use one extra bit of space, though this does not affect the complexity of the calculations.

Let $I(n)$ denote the cost of multiplying two integers with n bits and $M_{\mathbb{Z}}(n)$ is the time for multiplication of polynomials in $\mathbb{Z}[x]$ of degree n. Then using the substitution method from Section 3.1, we obtain $I(n) = \mathcal{O}(M_{\mathbb{Z}}(n))$. For the converse, we obtain the bound $M_{\mathbb{Z}}(n) = \mathcal{O}(I(n \log(nB^2)))$. In particular, if $\log n \in \mathcal{O}(\log B)$, then this can be simplified to $\mathcal{O}(I(n \log B)$.

3.4.4 Finite Fields

We will now introduce a lemma to use Kronecker substitution to convert polynomial multiplication in a finite field $\mathbb{F}_{p^k}$ for prime p, and $k \in \mathbb{N}$ to multiplication in $\mathbb{F}_p[x]$. This is used by Shoup's NTL library for number theory [36].

Lemma 1. Let $M_q(n)$ be the complexity of polynomial multiplication on the bit-oriented model over a finite field $\mathbb{F}_q$ with $q = p^k$ for some prime p. Then

$$M_q(n) \leq M_p(2nk) + \mathcal{O}(nM_p(k))$$

which reduces us the problem to the case where $k = 1$.

Proof. Let $\mathbb{F}_q = \mathbb{F}_p[x]/f(x)$ for a polynomial $f(x) \in F_p[x]$ of degree k. Notice the result of multiplying two polynomials of degree k has degree at most $2k$; we need to pad the elements of $\mathbb{F}_q$ with zeros to allow for polynomials of degree $2k$ in the result. Hence the newly expanded polynomial in $\mathbb{F}_p[x]$ has $2nk$ terms; thus multiplying them requires $M_p(2nk)$ ring operations. To transform the result back into a polynomial in $\mathbb{F}_q[x]$, we need to coerce the coefficients – which are polynomials of length $2k$ – back into $F_q[x]$, which we can achieve by dividing them by $f(x)$. Using Newton substitution (Theorem 9.6 [22]), polynomial division has the same asymptotic complexity as multiplication. Therefore coercing all the coefficients back into $\mathbb{F}_q[x]$ can be achieved in $\mathcal{O}(nM_p(k))$ ring operations. □

We can also use the technique from the previous section to convert multiplication in any integer ring $\mathbb{Z}/N\mathbb{Z}[x]$ for $N \in \mathbb{N}$ into integer multiplication. We first coerce elements in $\mathbb{Z}/N\mathbb{Z}$ into $\mathbb{Z}$, via the isomorphism

$$p + N \in \mathbb{Z}/N\mathbb{Z} \mapsto p \in \mathbb{Z}.$$

We first coerce the elements in the integer ring with the integers $0, \ldots, p-1 \in \mathbb{Z}$. Then use our normal multiplication techniques in $\mathbb{Z}[x]$. After computing the result, we must coerce all the coefficients back into $\mathbb{F}_p$, which can be done using the Euclidean algorithm in $\mathcal{O}(\log(p)(\log(np^2) - \log(p))) \in \mathcal{O}(\log(np)\log(p))$ time (Section 2.4 [22]), since we know that $\mathcal{M}_{\mathbb{Z}}(n)$ is at least linear time, we can conclude that the cost of the last operation is negligible compared to the cost of the multiplication. Therefore the complexity is

$$M_p(n) = \mathcal{O}(I(n \log np^2)) + \mathcal{O}(n \log p \log np).$$

Chapter 4

Evaluation and Interpolation

The algorithms we have seen so far have been direct calculations of the multiplication formula (3.1) introduced in Chapter 2. Soon after the rediscovery of the Fast Fourier Transform in 1965, the *evaluation-interpolation strategy* became a more popular choice for evaluating convolutions. Rather than evaluating the multiplication formula directly, in this strategy we sample the input polynomials at several points and interpolate the result from the product of the samples. Although initially this would appear to take more time that than the direct approach, it is remarkably efficient in practice and allows us to use techniques from Fourier analysis to multiply polynomials in $\mathcal{O}(n \log n)$ ring operations. In fact, we will revisit Karatsuba's algorithm from the previous chapter and show how it too can be interpreted as using this framework.

We will present an algebraic formulation of the Fast Fourier transform and see how it can be used to evaluate convolutions. We will then see a more clever use of the technique in Schönage and Strassen's integer multiplication algorithm generalised for polynomials which is currently used as part of the GMP Multiple Precision Arithmetic library for integers exceeding around 16000 bits [32].

For an intuitive explanation of how it works, consider the multiplication $h = fg \in K[x]$. Then it follows that $h(\alpha) = f(\alpha)g(\alpha)$ for all $\alpha \in K$. Thus by evaluating the polynomials f and g at $\deg f + \deg g + 1$ distinct points, we can multiply the samples together and interpolate h. We can correctly recover h since $\deg h \leq \deg f + \deg g$ and any polynomial of degree n can be interpolated from $n + 1$ distinct samples.

More formally, let $n = \deg f + \deg g$ and $f, g \in K[x]/(x^n - 1)$. Then evaluation and interpolation are the two directions of the isomorphism given by the Chines

Remainder Theorem

$$K[x]/(x^n - 1) \to K[x]/(x - \alpha_1) \times \cdots \times K[x]/(x - \alpha_n),$$

The process of coercing a and b into $K[x]/(x - \alpha_i)$ is called the *evaluation* step. The step where we recover the original polynomial in $K[x]/f(x)$ is called the *interpolation* step.

Evaluation at a single point can be performed in $\mathcal{O}(n)$ time via Horner's Rule and is asymptotically optimal[1]. Thus evaluating each sample individually at n distinct points is therefore $\mathcal{O}(n^2)$ time. Similarly, we may use Lagrange interpolation to interpolate a degree n polynomial in $\mathcal{O}(n^2)$ time. Neither of these methods provide any improvement over the standard schoolbook method, so this chapter will present an asymptotically faster algorithm for both evaluation and interpolation.

4.1 Chinese Remainder Theorem

The Chinese Remainder Theorem (CRT) is a classical theorem in the field of number theory. Where Kronecker substitution reorganises polynomials of one form to another, we will use the CRT to convert multiplications in one ring to multiplications in a collection of smaller rings to reduce the overall complexity. Since all current multiplication algorithms are super-linear in nature, the collective cost of the multiplication in the smaller rings is significantly less than the cost of applying an algorithm to the original ring. The difficulty however, is in evaluating the isomorphism efficiently enough to justify such an approach.

Theorem 2 (Chinese Remainder Theorem)**.** Let K be commutative ring and $I_1, \ldots, I_k \subseteq K$ mutually coprime ideals i.e. $I_i + I_j = K$.

Then

$$\frac{K}{I_1 \cdots I_k} \cong \frac{K}{I_1} \times \cdots \times \frac{K}{I_k}$$

We will be interested in the particular case where K is a polynomial ring and I_j are linear, i.e. $I_j = (x - \alpha_j)$ for some $\alpha_j \in K$. In this case we can evaluate the map $K/(I_1 \ldots I_k) \to K/(I_1) \times \cdots \times K/(I_k)$ by evaluating the polynomials at the points α_j.

[1] since we must perform some operation with each of the n terms in the polynomial at least once, so it must at least be linear

By associating polynomials with their coefficient vectors, we can consider these evaluation operations as multiplication by the Vandemonde matrix

$$\begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^n \\ 1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^n \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & \alpha_n & \alpha_n^2 & \cdots & \alpha_n^n \end{pmatrix}.$$

The inverse map can be found by inverting the matrix or, more simply, via the Lagrange interpolation formula

$$f(x) = \sum_{i=0}^{n-1} L_i(x) x^i, \qquad L_i := \prod_{j \neq i} \frac{x - j}{x_i - x_j},$$

where $(x_i)_{i=0}^{n-1}$ are the samples.

Note however that each of these operations require $\mathcal{O}(n^2)$ ring operations.

In general, evaluating the isomorphism in either direction is a computationally expensive task. However, there classes of rings in which the isomorphisms can be efficiently evaluated in a divide-and-conquer approach to improve the theoretical complexity significantly.

4.2 Karatsuba's Algorithm as Evaluation-Interpolation

Before introducing the DFT, we will first show how Karatsuba's algorithm may be viewed as a method for evaluating the Chinese Remainder Theorem for the isomorphism

$$\frac{K[x]}{x^2 - x} \cong \frac{K[x]}{x} \times \frac{K[x]}{x - 1}, \tag{4.1}$$

along with an *evaluation at infinity*[2].

Let $f, g \in K[x]$, both with degree n and denote their product as $h = fg$. Then we will apply Kronecker substitution with $y = x^{\lceil n/2 \rceil}$, to obtain the polynomials $f', g', h' \in K[x][y]/(y^2 - y)$ and may write

$$f(x)(y) = f_1(x)y + f_0(x), \qquad g(x)(y) = g_1(x)y + g_0(x).$$

for $f_0, f_1, g_0, g_1 \in K[x]$ with degree less than $\lceil n/2 \rceil$.

[2]There are variations that use $x^2 + x$ as the divisor which may be more computationally efficient in certain circumstances. See Knuth, Art of Computer Programming Vol 2. [17]

Compute the isomorphism (4.1) by evaluating at the points $y = 0$ and $y = 1$

$$f'(x)(0) = f'_0(x), \qquad f'(x)(1) = f'_1(x) + f'_0(x)$$
$$g'(x)(0) = g'_0(x), \qquad g'(x)(1) = g'_1(x) + g'_0(x).$$

We can then apply the Chinese remainder theorem to recover h'

$$h'(y) = f'_0 g'_0 + ((f'_0 + f'_1)(g'_0 + g'_1) - f'_0 g'_0)y \in K[x][y]/(y^2 - y). \tag{4.2}$$

The problem now is $h' \in K[x][y]/(y^2-y)$, however $h \in K[x]$ may have a non-zero y^2 term. Therefore we must recover this term by *evaluating at infinity*.

Since we know the coefficient of this term would be $f_1 g_1$, if we were to coerce h into $K[x][y]/(y^2-y)$ we would obtain $f_0 g_0 + ((f_0+f_1)(g_0+g_1)+f_1 g_1)x$. Thus to undo this operation in our result (4.2) we obtain $f_0 g_0 + ((f_0+f_1)(g_0+g_1) - f_0 g_0 - f_1 g_1)x + f_1 g_1 x^2 \in K[x]$ which matches our original formulation of Karatsuba's algorithm.

As was mentioned before, the Toom-Cook algorithm is a generalisation of Karatsuba's algorithm; rather than mapping our polynomials using the isomorphism (4.1), we use the map[18]

$$\frac{K[x]}{(x-(k-1))(x-(k-2))\cdots(x+k-1)} \cong \frac{K[x]}{x-(k-1)} \times \frac{K[x]}{x-(k-2)} \times \cdots \times \frac{K[x]}{x+k-1}$$

4.3 The Fast Fourier Transform

The Discrete Fourier Transform (DFT) is an powerful mathematical technique for performing Fourier analysis on a broad selection of applications. Most notably, it is foundational in the field of signal processing when it can be used to transform data between the sample space and the frequency domain. The Fast Fourier Transform (FFT) is an algorithm for evaluating the DFT (and its inverse) in $\mathcal{O}(n \log n)$ ring operations. The naïve algorithm for evaluating the DFT takes $\mathcal{O}(n^2)$ operations, which renders it infeasible for many practical applications. Soon after Cooley and Tukey's landmark paper on the FFT in 1965[2] spawned a flurry of development across many fields. One example is the development of FFT-based algorithms for polynomial multiplication, which was the first $\mathcal{O}(n \log n)$ algorithm for the problem.

The DFT is the evaluation of a function at *roots of unity*, and its inverse is a means of recovering the original function from samples. Hence this can be seen as an instance of the evaluation-interpolation strategy. In this section we formulate both the DFT and FFT and explore the relationship between the DFT and polynomial multiplication.

4.3.1 The Discrete Fourier Transform

Let R be a commutative ring. We are familiar with the complex roots of unity given as the solutions to the equation $x^n - 1$ for $n \in \mathbb{N}$. However many rings admit elements that behave similarly and can be used to construct analogues of the DFT and FFT for commutative rings.

Definition 2 (Roots of Unity)**.** Let R be a commutative ring and let N be an integer, an element $\alpha \in R$ is a:

1. *root of unity*, if $\alpha^N = 1$, and is a
2. *principal root of unity*, if $\alpha^p - 1$ is not a zero divisor for all $1 \leq p < N$.

It is clear that the complex roots of the equation $x^N - 1$ satisfy the above definitions. Though there are many other coefficient algebras that also have roots of unity for a particular N. A popular choice is the ring $K[x]/(x^n + 1)$ which has x as a $2n^{\text{th}}$ root of unity (also called a *synthetic root of unity*). We will revisit this ring in Section 4.4 when discussing Schönage and Strassen's integer multiplication scheme for polynomial rings. Chapter 5 will establish results for the existence of roots of unity in integer rings as well as methods for finding them.

Lemma 2. Let ω be a principal N^{th} root of unity. Then:

1. $\omega^{N/2} = -1$
2. If N is divisible by two, then $\sum_{i=0}^{N-1} \omega^{ik} = 0$.

Proof. Note that

$$0 = \omega^{kN} - 1 = (\omega^k - 1) \sum_{i=0}^{N-1} \omega^{ik}$$

Since $\omega^k - 1$ is not a zero divisor, it must be the case that $\sum_{i=0}^{N-1} \omega^{ik} = 0$.

Suppose 2 divides N. Then since $0 = \omega^N - 1 = (\omega^{N/2} - 1)(\omega^{N/2} + 1)$ and $\omega^{N/2} - 1$ is not a zero divisor, we conclude $\omega^{N/2} = -1$. □

Definition 3 (Discrete Fourier Transform)**.** Let K be a commutative ring with a principal N^{th} root of unity $\omega \in K$[3], and let $u \in K^n$.

The Discrete Fourier Transform is the map $\mathcal{F}_N : K^N \to K^N$ defined by

$$(\mathcal{F}u)_k = \sum_{i=0}^{N-1} u_i \omega_N^{-ik} \tag{4.3}$$

[3]The ω_N^k in the equation are commonly referred to as the *twiddle factors*

If N is invertible in K, the inverse $\mathcal{F}_N^{-1}$ is given by

$$(\mathcal{F}u)_k = \frac{1}{N}\sum_{i=0}^{N-1} u_i \omega_N^{ik} \tag{4.4}$$

Alternatively, we can interpret this as, given samples $x_0, \ldots, x_{n-1}$, we can construct the polynomial $f(x) = \sum_{i=0}^{n-1} f_i x^i$. Then the DFT induces the isomorphism

$$\frac{K[x]}{x^n - 1} \cong \frac{K[x]}{x-1} \times \frac{K[x]}{x-\omega} \cdots \times \frac{K[x]}{x-\omega^{n-1}}, \tag{4.5}$$

which we know are isomorphic from the Chinese Remainder Theorem. Hence we can consider this forward DFT as the evaluation of the polynomial f at the roots of unity, and the inverse as an interpolation.

Note that when computed directly, each of the N sums in the forward or inverse DFT take $\mathcal{O}(N)$ ring operations. Hence computing one transform takes $\mathcal{O}(N^2)$ ring operations, which is still no better than the schoolbook method.

From (4.5) we can deduce the following expression, known as the convolution property.

$$u * v = \mathcal{F}^{-1}(\mathcal{F}u \cdot \mathcal{F}v) \tag{4.6}$$

4.3.2 The Fast Fourier Transform Algorithm

The Fast Fourier Transform (FFT) is an algorithm for calculating the DFT with N samples in $\mathcal{O}(N \log N)$ ring operations. First developed by Gauss in 1805 [9] and rediscovered by Cooley-Tukey in 1965, this algorithm has had a profound impact on the course of human computing. Due to its efficiency on modern computer hardware, the FFT was able to make many previously incomputable problems in signal processing tractable. There are many variations of the Fast Fourier Transform, such as the Dutt-Rokhlin method for non-uniform FFTs[13][4]; but here we will analyse the original algorithm described in the landmark Cooley-Tukey paper [2].

Theorem 3 (Fast Fourier Transform)**.** Let K be a commutative ring that admits primitive roots of unity of order N, where N is an invertible power of two. Let $\mathcal{C}_F(N)$ be the number of ring operations requires to perform the DFT of N samples in K. Then $\mathcal{C}_F \in \mathcal{O}(N \log N)$.

[4]The algorithm in Chapter 6 is similar to a non-uniform FFT, though we will not go into the details there and instead refer the reader to Section 4.4.3 of [27] for a comparison of the two methods.

Corollary 1. Let K be a commutative ring that supports an FFT with transform length n a power of two. Then for $f, g \in K[x]$ polynomials of degree n we can evaluate the product $h = fg$ in $\mathcal{O}(n \log n)$ ring operations.

The FFT uses a divide-and-conquer approach whereby the DFT is split into two smaller sub-DFTs of $N/2$ elements, such that we the full DFT can be recovered in $\mathcal{O}(N)$ ring operations. Since at each level in the recursion we would expect a cost of $\mathcal{O}(N)$ and we can halve N at most $\log_2 n$ times, we might guess that the total cost is $\mathcal{C}_F \in \mathcal{O}(N \log N)$.

The following lemma which formalises this intuition and the general algorithm for computing the DFT efficiently.

Lemma 3. Let K be as in Theorem 3, then

$$\mathcal{C}_F(N) \leq 2\mathcal{C}_F(N/2) + \mathcal{O}(N). \tag{4.7}$$

Proof. Let $u \in K^n$, and so we wish to compute $\mathcal{F}u$.. Throughout this proof we will write $(\mathcal{F}u)_k$ with the understanding that $X_k = X_{k \mod N}$.

The crucial step in the FFT, is to partition the sum into a sum of x terms with an even index and another with an odd index. Beginning with the formula for $(\mathcal{F}u)_k$ (4.3), we get

$$\begin{aligned}
(\mathcal{F}u)_k &= \sum_{j=0}^{N-1} u_j \omega_N^{-jk} \\
&= \sum_{j=0}^{N/2-1} u_{2j} \omega_N^{-2jk} + \sum_{j=0}^{N/2-1} u_{2j+1} \omega_N^{-(2j+1)k} \\
&= \sum_{j=0}^{N/2-1} u_{2j} \omega_N^{-2jk} + \omega_N^{-k} \sum_{j=0}^{N/2-1} u_{2j+1} \omega_N^{-2jk} \qquad (4.8) \\
&= \sum_{j=0}^{N/2-1} u_{2j} \omega_{N/2}^{-jk} + \omega_N^{-k} \sum_{j=0}^{N/2-1} u_{2j+1} \omega_{N/2}^{-jk} \qquad (4.9)
\end{aligned}$$

where the last line follows from the fact that ω_N^2 is a primitive root of unity of order $N/2$, and so $\omega_N^2 = \omega_{N/2}$. Hence the two sums in (4.9) are both DFTs of length $N/2$ of the coefficient vectors $(u_0, u_2, \ldots, u_{N-2}), (u_1, u_3, \ldots, u_{N-1}) \in K^{N/2}$.

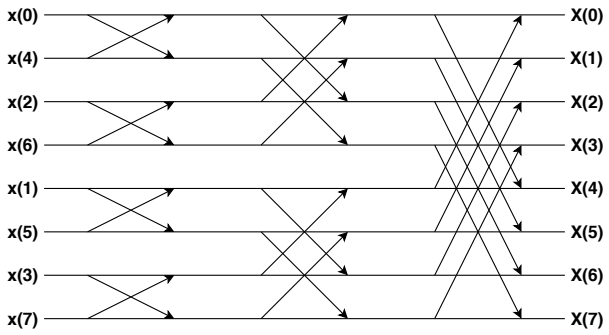

Figure 4.1: The FFT algorithm for $N = 8$.

Now lets apply this method to $(\mathcal{F}u)_{k+N/2}$

$$\begin{aligned}(\mathcal{F}u)_{k+N/2} &= \sum_{j=0}^{N/2-1} u_{2j}\omega_{N/2}^{-j(k+N/2)} \;+\; \omega_N^{-(k+N/2)} \sum_{j=0}^{N/2-1} u_{2j+1}\omega_{N/2}^{-j(k+N/2)} \\ &= \sum_{j=0}^{N/2-1} u_{2j}\omega_{N/2}^{-jk} \;-\; \omega_N^{k} \sum_{j=0}^{N/2-1} u_{2j+1}\omega_{N/2}^{-jk},\end{aligned}$$

since $\omega_{N/2}^{N/2} = 1$ and $\omega_N^{N/2} = -1$ from Lemma 2.

Observe that the sub-DFTs in $(\mathcal{F}u)_k$ are the same as the sub-DFTs in $(\mathcal{F}u)_{k+N/2}$, the only difference is that in $(\mathcal{F}u)_k$ they were added and in the case of $(\mathcal{F}u)_{k+N/2}$ they were subtracted. The combinations of the two sums $(\mathcal{F}u)_k$ and $(\mathcal{F}u)_{k+N/2}$ is known as a *butterfly*. Butterflies are the building blocks for the entire FFT algorithm which is illustrated in Figure 4.1. Each column of arrows denotes the butterflies for a certain recursion depth.

Computing each of the two smaller DFTs takes $2\mathcal{C}_F(N/2)$ ring operations and computing the $N/2$ butterflies takes takes $\mathcal{O}(N)$ ring operations.

Altogether this gives

$$\mathcal{C}_F(N) \leq 2\mathcal{C}_F(N/2) + \mathcal{O}(N).$$

□

Proof of Theorem 3. Replace the $\mathcal{O}(N)$ term in Lemma 3 with kN for any $k > 0$.

$$\mathcal{C}_F(N) \leq 2\mathcal{C}_F(N/2) + kN$$

We will show by induction that $\mathcal{C}_F(N) \leq kN \log_2 N$ for all N a power of two.

We know this is true for $N = 1$, because in that case the DFT is simply the identity so $\mathcal{C}_F(1) = 0 \leq k(1 \log_2 1)$. Suppose $\mathcal{C}_F(M) \leq kM \log_2 M$ for all M a

power of two less than N. Then

$$\begin{aligned}\mathcal{C}_F(N) &\leq 2\mathcal{C}_F(N/2) + pN \\ &\leq pN\log_2(N/2) + pN \\ &= pN\log_2 N - pN\log_2 2 + pN \\ &= pN\log_2 N\end{aligned}$$

Therefore by the principle of mathematical induction, we conclude $\mathcal{C}_F(N) \leq \mathcal{O}(N\log N)$ for all N a power of two. □

4.3.3 Mixed-radix FFT

In our previous formulation, we recursively split the DFT into two smaller DFTs each time; this is known as the radix-2 FFT. In this section, we will generalise the FFT algorithm for an arbitrary radix, though in practice this is often much less efficient than the radix-2 implementation. Padding the DFT changes the underlying signal and so in general, the padded result will no yield the correct solution to the original DFT. However as we showed, when using the DFT to multiply polynomials, we do not have this constraint and so we have the advantage of being able to pad the input polynomials with zeros in order to achieve a transform with a power of two length. Though padding DFT can become twice as long in each of the inputs, but in most implementations, a mixed radix-FFT with non-power of two lengths can be much more than double the time, as we don't have access to fast bit operations.

Though there are certain situations where we may choose to use a mixed-radix FFT, namely if K does not admit roots of unity whose order is a power of two. For example $\mathbb{Z}/p^r\mathbb{Z}$ for any prime $p \in \mathbb{N}$ and $p^r - 1$ is not a power of two. We will look at the existence of roots of unity and their transform sizes in Chapter 5.

Theorem 4 (Mixed Radix FFT)**.** Let K be a commutative ring with roots of unity of order N where N is invertible. Then if $m \in \mathbb{N}$ divides N, we have

$$\mathcal{C}_F(N) \leq m\mathcal{C}_F(\frac{N}{m}) + \frac{N}{m}\mathcal{C}_F(m) + \mathcal{O}(n).$$

Proof. Here we will partition $(\mathcal{F}_N u)_k$ into m smaller sums, and denote $k = p_1\frac{N}{m} + p_2$

$$
\begin{aligned}
(\mathcal{F}_N u)_{p_1 \frac{N}{m} + p_2} &= \sum_{j=0}^{m-1} \sum_{i=0}^{\frac{N}{m}-1} x_{mi+j} \omega_N^{-(mi+j)(p_1 \frac{N}{m} + p_2)} \\
&= \sum_{j=0}^{m-1} \left(\sum_{i=0}^{\frac{N}{m}-1} x_{mi+j} \omega_N^{-mip_2} \right) \omega_N^{-j(p_1 \frac{N}{m} + p_2)} \\
&= \sum_{j=0}^{m-1} \left(\omega_N^{-jp_2} \sum_{i=0}^{\frac{N}{m}-1} x_{mi+j} \omega_N^{-mip_2} \right) \omega_N^{-\frac{N}{m} p_1 j} \qquad (4.10)
\end{aligned}
$$

Just as the original FFT splits the sum into two sub-DFTs – one with all the coefficients at odd indices and the other the even indices – the i^{th} sum in (4.10) consist of all coefficients a_r where $r = i \mod m$. The reason we split up the $\omega_N^{j(\frac{N}{m} p_1 + p_2)}$ term in the last line is so that we can recombine all of these sub-DFTs using another DFT of size $\frac{N}{m}$; we simply need to multiply each of the inner DFTs by $\omega_N^{-jp_2}$.

Calculating all the sub-DFTs takes $\mathcal{C}_F(\frac{N}{m})$ time since there are m of size $\frac{N}{m}$. Multiplying the inner DFTs by roots of unity requires $\mathcal{O}(N)$ ring operations. Combining them together is equivalent to calculating $\frac{N}{m}$ sub-DFTs of size m, which can be done it at least $\frac{N}{m}\mathcal{C}_F(m)$ ring operations.

In total this operation requires

$$\mathcal{C}_F(N) = m\mathcal{C}_F(\frac{N}{m}) + \frac{N}{m}\mathcal{C}_F(m) + \mathcal{O}(N)$$

ring operations. □

Note that in the Turing machine, the m inner sub-DFTs also require us to reorganise u into m consecutive vectors of size $\frac{N}{m}$. This can be achieved in $\mathcal{O}(N \log(\min\{m, \frac{N}{m}\}\text{bit}(K)))$ time where $\text{bit}(K)$ is the number of bits required to store an elements of K [28]. We will revisit this fact in Section 6 when formalising our computations in the Turing model.

From Theorem 4, we can see that changing m does not change the asymptotic complexity. Since we consider m to be fixed $\mathcal{C}_F(m)$ is a constant, and so asymptotically this is equivalent to the recursive formula $m\mathcal{C}_F(\frac{N}{m}) + \mathcal{O}(N)$ whose solution is $\mathcal{O}(N \log N)$ as before; there is only a constant factor of different between different choices of m.

Remark 1. Our formulation of the FFT is only valid for powers of two, so we may need to pad our inputs with zeros to satisfy this requirement. In order to

avoid this padding as much as possible, we may consider constructing a mixed-radix FFTs that generalises the FFT to transform sizes that are not powers of two. However, we note that these tend to be much more difficult to implement efficiently and in typical cases, it is more efficient to pad the polynomials in order to use the standard FFT than it is to use a mixed-radix DFT. This is because modern computers can perform many operations much more efficiently in base 2, and forcing a different radix can cause operations to be several times slower.

One such optimisation in radix-2 is the *reverse bit encoding*, where we can organise $\mathcal{F}_n u$ in memory so that the entire FFT algorithm can be performed in-place (all data transformation happen inside the array the original data was passed in). This is done by putting $\mathcal{F}_N u$ at index $\text{rev}(i, n)$, which is the function which reverses the order of the n-bit representation of i.

One technique to avoid excessive padding is to only pad to a transform size that is the product of a power of two and several small primes. This is so that the bulk of the computation can be done using the previous method and then when we recurse to a small base case we directly compute the sum in the DFT. Many practical implementations have a base case of around $N = 10$ even when N is a power of two, so this incurs no performance penalty.

4.4 Schönage and Strassen Integer Multiplication Algorithm

The problem with multiplying polynomials in $\mathbb{Z}[x]$ is that $\mathbb{Z}$ does not admit roots of unity. One can consider $\mathbb{Z}[x]$ as a subring of $\mathbb{C}[x]$ and then apply the FFT algorithm, rounding to the nearest integer to convert the result back into $\mathbb{Z}[x]$. This is a popular approach for medium-sized inputs, however, if the error becomes too large, then rounding to the nearest integer will yield the incorrect result, and so one must be able to guarantee the error is suitably controlled throughout the process. If we were to use the algorithms defined in the previous section with enough precision to guarantee the solution is correctly recovered then complexity would no longer be $\mathcal{O}(n \log n)$ as the number of the bits requires to store the ring elements is quite large in accordance to current error bounds on the complex DFT (Theorem 5.1 [37]).

In Chapter 6, we present an algorithm that achieve achieves $\mathcal{O}(n \log n)$ complexity in the Turing model by carefully choosing DFT transforms lengths and controlling the error. For now, we will look at the most popular algorithm for mul-

tiplication of polynomials in $\mathbb{Z}[x]$ with large degree sizes: Schönage and Strassen's second integer multiplication algorithm for commutative rings.

Schönage and Strassen's originally formulated their algorithm for integers, however we will present a generalisation of the algorithm for commutative rings where 2 is a unit based on the presentation in [22].

Proposition 2. Let K be a commutative rings where 2 is a unit. Then write $M_K(n)$ to denote the time to multiply polynomials $f, g \in K[x]$ with degree at most n, then

$$\mathcal{C}(n) \leq t\mathcal{C}(t+1) + \mathcal{O}(n \log t)$$

Note that K may not admit roots of unity, or perhaps it only has roots of unity for small transform sizes in which case this algorithm will be more efficient for large n than classical ones such as Karatsuba's. This algorithm was generalised further by Cantor and Kaltofen to arbitrary non-associative, non-commutative algebras[12].

The Schönage and Strassen integer multiplication algorithm applies Kronecker substitution to transform polynomials in $\mathbb{Z}[x]$ into polynomials with coefficients in $\mathbb{Z}[x]/(x^n+1)$, which admit particularly efficient roots of unity, called *synthetic roots*.

Let K be a ring such that 2 is a unit in K, and let $n = 2^k$ for some $k \in \mathbb{N}$, and $D = K[x]/(x^n+1)$. In general, D is not a field, however observe that X is a primitive root of unity of order $2n$ since $X^{2n} = (-1)^2 = 1$. Then consider the problem of multiplying polynomials $f, g \in K[x]$ to obtain $h = fg$. In previous sections we performed multiplications in $K[x]/(x^n-1)$ for $\deg f + \deg g < n$. However in this algorithm we will multiply the polynomials in $K[x]/(x^n+1)$ which is known as the *nega-cyclic convolution*.

Proof. Let $m = 2^{\lfloor k/2 \rfloor}$ and $t = n/m = 2^{\lceil k/2 \rceil}$. Apply Kronecker substitution with $y = x^m$ to obtain $f', g' \in K[x][y]$, that is, we break up the polynomials into polynomials whose coefficients of polynomials of size approximately $\sqrt{n}$. Now it is sufficient to compute $f'g' \in K[x][y]/(y^t+1)$ since We now take the primitive $4m^{\text{th}}$ root of unity $\omega = x \in K[x]/(x^{2m}+1)$. We wish to compute $h' \in K[x,y]$ with $deg_y h' < t$ satisfying the previous equations . Comparing coefficients of y^j for $j \geq t$ we see that $\deg_x q' \leq \deg_x(f'g') < 2m$ and conclude that

$$\deg_x h' \leq \max\{\deg_x(f'g'), \deg_x q'\} < 2m.$$

Thus we may perform the calculation of h' in the field $K[x]/(x^n + 1)$. With $f^* = f' \mod (x^{2m} + 1)$, and similarly for g and h. We have

$$f^* g^* \equiv h^* \mod (y^t + 1).$$

Since the three polynomials have degrees in x less than $2m$ by the previous equation, reducing them modulo $x^{2m} + 1$ is just taking a different algebraic meaning of the same coefficient array, in particular, the coefficients of $h' \in K[x][y]$ can be read off the coefficients of $h^* \in D[y]$.

Computationally "nothing happens" when mapping h' to h^*, but we are now in a situation where we can apply the machinery of the FFT. Since t equals either m or $2m$, D contains a primitive $2t^{\text{th}}$ root of unity η Thus one of the previous equations is then

$$f^*(\eta y) g^*(\eta y (\equiv h^* * \eta y) \mod ((\eta y)^t + 1)$$

Given $f^*(\eta y)$ and $g^*(\eta y)$ we can use the FFT to compute $h^*(\eta y)$ with $\mathcal{O}(t \log t)$ operations in D, using three t-point FFTs. A multiplication in D is again a negatively wrapped convolution over R which can be handled recursively.

This gives the recursive formula

$$\mathcal{C}(n) \leq t\mathcal{C}(m + 1) + \mathcal{O}(n \log t)$$

□

Theorem 5. Let K be a commutative rings where 2 is a unit. Then we may multiply polynomials $f, g \in K[x]$ with degree at most n, in $\mathcal{O}(n \log n \log \log n)$ ring operations.

Proof. Then first we ascertain the recursion depth of the function. We may assume the base case occurs when the degree of the polynomials is less than or equal to two. Then the maximum recursive depth $i \in \mathbb{N}$ satisfies

$$n^{\frac{1}{2^i}} \leq 2$$
$$n \leq 2^{2^i}$$
$$\log \log n \leq i$$

Thus by expanding the recursive formula in 2 fully we obtain

$$\begin{aligned}\mathcal{C}(n) &\leq t\mathcal{C}(t) + \mathcal{O}(n \log n) \\ &\leq \sum_{i=0}^{\log \log n} n^{\frac{1}{2^i}} \log n^{\frac{1}{2^i}} \\ &\leq \sum_{i=0}^{\log \log n} (n \log n) \\ &= n \log n \log \log n\end{aligned}$$

□

Therefore we may multiply integers of length n can be performed with $\mathcal{O}(n \log n \log \log n)$ machine operations. For a 32 bit compute we could simply set $K = \mathbb{Z}/31\mathbb{Z}$, so all operations in K take constant time, and use Kronecker substitution to convert the integer into a polynomial in $K[x]$.

Chapter 5

Integer Rings

In this section, present technique for multiplying polynomials over integer rings. We will formalise the notion of a discrete convolution and establish important theoretical results for the existence of roots of unity in integer rings of certain order. Ad we will see, not all integer rings admit roots of unity, and even if they do, they may have small order e.g. $\mathbb{Z}/2^{2k+1}\mathbb{Z}$ only admits roots of unity of size 2. A large portion of this chapter will be devoted to understanding when such a ring admits roots of unity of high order to enable such transformations. As such, the algorithms in this chapter will not present any new theoretical bound on multiplication; the aim is to create algorithms that are practically efficient for moderate input sizes.

Integer rings are commonly used to control the size of the coefficients throughout the multiplication process for polynomials in $\mathbb{Z}[x]$. We can first transform the polynomials in $\mathbb{Z}[x]$ into several polynomials in integer rings that admit fast FFTs, and then reconstruct the result in $\mathbb{Z}[x]$ via the Chinese Remainder Theorem. This is a very popular choice for large parallel computing as it is easily implemented on a Graphics Processes Unit (GPU) [20] [21] [25].

The techniques in this chapter are commonly used throughout cryptography and signal processing. These applications offer a large amount of flexibility over the choice of the integer ring, and so Section 5.2 discusses criteria for choosing rings which admit efficient arithmetic.

It is important to differentiate the use cases as each imposes a different set of restrictions on the ring, and subsequently, the optimisations we can apply. The most favourable to choose are those that offer efficient arithmetic in common computer architectures. Though we also that the methods in this chapter may be supplemented with a fast division scheme such as Barret Reduction [11]

which improves the efficiency of divisions in the integers rings by replacing them with multiplications. This is often much more efficient since the regular division operation in most modern CPUs is not as efficient as multiplications [26].

Bear in mind, that one can still use the algorithms from Chapter 3 and Chapter 4 if we first use Kronecker substitution to transform the inputs from $\mathbb{Z}/N\mathbb{Z}[x]$ to a problem in $\mathbb{Z}[x]$. However, it is often more efficient to perform calculations directly in $\mathbb{Z}/N\mathbb{Z}$ since the elements are always bounded and so arithmetic operations as executed in a time proportional to N.

5.0.1 Convolutions

Convolutions are fundamental in the field Fourier analysis as the DFT is the unique function that has the *convolution property*. We will formulate this property and show how polynomial multiplication can viewed to provide a second intuition as to how the FFT can be used to multiply polynomials. Though we will show polynomial multiplication and convolutions to be equivalent problems, the language of convolutions is often more apt in the context of signal processing and finite automata where the techniques in this section are commonly applied.

Definition 4 (Discrete Convolution)**.** Let K be a commutative ring, and $u, v \in K^n$. The *convolution* of u and v is defined as

$$(u * v)_k = \sum_{i=0}^{n-1} u_i v_{k-i} \in K^n,$$

where u_{k-i} is understood to mean $u_{k-i \mod n}$.

Proposition 3. Let K be a commutative ring. Then for any $n \in \mathbb{N}$, we have the isomorphism of rings

$$\left(\frac{K[x]}{x^n - 1}, \times\right) \cong (K^n, *).$$

Proof. The map $K[x]/(x^n - 1) \to K^n$ is given by associating the coefficients of a polynomial with its corresponding vector. That is

$$\sum_{i=0}^{n-1} f_i x^i \mapsto (f_0, \ldots, f_{n-1}). \tag{5.1}$$

This is clearly a bijection.

To see that this is an isomorphism, consider two polynomials $f, g \in K[x]/(x^n - 1)$, given by $f(x) = \sum_{i=0}^{n-1} f_i x^i$ and $g(x) = \sum_{i=0}^{n-1} g_i x^i$. Then

$$\begin{aligned} f(x)g(x) = \sum_{k=0}^{2n-1} \left(\sum_{i+j=k} f_i g_j \right) x^k &= \sum_{k=0}^{n-1} \left(\sum_{i+j=k \mod n} f_j g_i \right) x^k \\ &= \sum_{k=0}^{n-1} \left(\sum_{j} f_j g_{k-j} \right) x^k \mod (x^n - 1). \end{aligned}$$

Hence the coefficients of the polynomial are exactly the those given by the convolution of f and g, thus the map 5.1 induces the isomorphism in the proposition. □

Corollary 2. For $n \in \mathbb{N}$, Polynomial multiplication in $K[x]$ and convolutions in $K^n/(x^n - 1)$ for any $n \in \mathbb{N}$ are equivalent problems.

Proof. To multiply polynomials using a convolution, let $f, g \in K[x]$ and $n = \deg f + \deg g + 1$. Then since $(K[x]/(u^n - 1), \times) \cong (K^n, *)$, we may use a convolution to evaluate $h = fg \in K[x]/(x^n - 1)$. Since $\deg f, \deg g, \deg h < n$, we can recover the true answer $h \in K[x]$.

To use multiplication to solve convolutions, we convert the vectors $f', g' \in K^n$ into polynomials $f, g \in K[x]$. Then multiply as usual and apply the substitution $x^n = 1$. This requires $\mathcal{O}(n)$ ring operations, and since we know that multiplication has at least linear complexity, this does not change the total asymptotic complexity. □

Definition 5 (Convolution Property)**.** Let u and v be two sequences of length n in a ring K. Then let $\mathcal{F}_n$ and $\mathcal{F}_n^{-1}$ denote the discrete Fourier transform and its inverse. Then

$$u * v = \mathcal{F}_n^{-1}(\mathcal{F}_n u \cdot \mathcal{F}_n v)$$

where $\cdot$ represented element-wise multiplication of the two sequences.

This can be proved by applying Corollary 2 and the analogous result for polynomials in Chapter 4.

We mentioned previously that padding the DFT with zeros changes the underlying signal and may lead to an incorrect result in the general case; Bluestein's algorithm allows us to overcome this issue by expressing a DFT as a convolution.

Proposition 4 (Bluestein's Algorithm)**.** Let $\mathcal{C}_F(n)$ denote the number of ring operations required to compute a DFT, and $\mathcal{M}_K(n)$ denote the number of ring operations to compute an n-dimensional convolution in K^n. Then

$$\mathcal{C}_F(n) \leq \mathcal{M}_K(n) + \mathcal{O}(n).$$

Proof. Recall the Fourier coefficients of the DFT of a sequence $u \in K^n$ are given as

$$(\mathcal{F}_n u)_k = \sum_{j=0}^{N-1} u_j \omega_n^{-jk}$$

where ω is an N^{th} root of unity.

Using the identity $jk = \frac{1}{2}(j^2 + k^2 - (j-k)^2))$ we can rewrite the formula as

$$(\mathcal{F}_n u)_k = \omega_N^{k^2/2} \sum_{j=0}^{N-1} (a_j \omega_N^{j^2/2}) \omega_n^{(j-k)^2/2}.$$

If we then take $f = (u_n \omega_N^{j^2/2})_{j=0}^{N-1}$ and $g = (\omega_N^{j^2/2})_{j=0}^{N-1}$ we have the formula

$$(\mathcal{F}_n u)_k = g_k \cdot \left(\sum_{j=0}^{N-1} f_j g_{k-j} \right) = g_k \cdot (f * g)_k. \tag{5.2}$$

Thus we can use the convolution to evaluate the DFT. Calculating f and g requires $\mathcal{O}(n)$ ring operations, and after calculating the convolution we must multiply the result elements-wise, which can also be done in $\mathcal{O}(n)$ ring operations. Therefore, calculating (5.2) gives

$$\mathcal{C}_F(n) \leq \mathcal{M}_K(n) + \mathcal{O}(n).$$

□

Therefore since element-wise multiplications is $O(n)$, we can see that the problem of calculating convolutions has the same complexity as computing the DFT.

It may seem counterintuitive as we would then use the DFT to evaluate the convolution. However, the fundamental difference is that when evaluating a convolution, we may choose an DFT of any transform length greater than $N-1$, which we are unable to do with the original DFT. Thus we can use Bluestein's algorithm[3] to use more efficient FFT algorithms for specific transform sizes e.g. highly composite sizes (FFT), prime sizes (Rader's Transform, Definition 6).

5.1 Number Theoretic Transform

Consider the problem of evaluating the DFT of a sequence of n elements of the integer ring $\mathbb{Z}/N\mathbb{Z}$ for some $N \in \mathbb{N}_{>0}$. Recall that the FFT may be applied in any field which admits roots of unity, and where 2 is invertible. We can always

use the Schönage and Strassen algorithm when N is odd, however it may only outperform classical algorithms such as Karatsuba's at exceedingly large degree sizes. If N is prime, we can use Rader's algorithms to reorder the inputs as a convolution of length $N-1$. Thus if $N-1$ is highly composite, this enables use to apply FFT-based algorithms.

Though nothing practical changes, when discussing the DFT for integer rings, it is more commonly called The Number Theoretic Transform (NTT).

5.1.1 Rader's Algorithm

Rader's algorithm is a method to evaluate the DFT of prime size N by expressing it as a cyclic convolution of size $N-1$. This is similar to Bluestein's algorithm but is much more efficient in practice since we don't need to calculate any additional sequences or vector products. This is not as efficient for arbitrary finite fields, but in many applications such as cryptography, we may choose to perform our calculations over fields where $N-1$ is highly composite to facilitate an efficient NTT. This is a common trick used in industry software such as the FFTW library which uses Rader's algorithm to gain a 2x speedup in certain situations [31].

Definition 6 (Rader's Algorthm)**.** Let p be prime, then there exists a primitive element of $\mathbb{F}_p$, that is, a $g \in \mathbb{F}_p$ such that $(g^i)_{i=1}^{N-1}$ is a permutation of $1, \ldots, N-1$. Then by applying the map $i \mapsto g^i, k \mapsto g^k$ the DFT $\mathcal{F}u$ of $u \in \mathbb{F}_p^n$ can be realised as a convolution of the two vectors $(u_{g^i})_{i=0}^{n-1}$ and $\omega_N^{g^i}$,

$$A_k - a_0 = \sum_{i=1}^{n-1} a_i \omega_N^{ik} \qquad \mapsto \qquad A_{g^k} - a_0 = \sum_{i=1}^{n-1} a_{g^i} \omega_n^{g^{i+k}}$$

If $N-1$ is highly composite, we can apply the standard FFT algorithms to evaluate the sum efficiently. Therefore the ideal case is when $N = 2^k + 1$ for some k and where N is prime. In the special case that n is also a power of two, N is a *Fermat number*. We will revisit these in Section 5.2 as they are especially convenient when performing arithmetic operations.

5.1.2 Existence of NTT

Now we will establish theoretical results for the existence of roots of unity in integers rings. In this section we will primarily follow the treatment in [6].

Theorem 6. If there exists a root of unity of order n the ring of integers modulo an integer N, if and only if n and N are relatively prime, then n divides $\gcd(p_1 - 1, \ldots, p_k - 1)$ where $p_1, \cdots, p_k$ are the distinct prime divisors of N.

Proof. Suppose α is a root of unity of order n, that is $\alpha^n \equiv 1 \mod N$. Then via the Chinese Remainder Theorem we must also have $\alpha^n \equiv 1 \mod p_i^{r_i}$ for all i and hence $\alpha^n \equiv 1 \mod p_i$. Since n and N are coprime, we have that n and $p_i^{r_i}$ are coprime, and so by Euler's Theorem, n must divide the Euler totient function $\varphi(p_i^{r_i}(p_i - 1)$.

Since n and N are relatively prime, $n \nmid p_i^{\alpha_i}$ therefore $n \mid p_i - 1$ for all $1 \leq i \leq k$. Thus $n | \gcd\{p_1 - 1, \ldots, p_k - 1\}$.

To show existence, note that if $n|(p_i - 1)$, then there exists integers of order n in $\mathbb{Z}/p_i^{\alpha_i}$. Applying the Chinese Remainder Theorem on these elements in the reverse direction, we can recover an element $\alpha \in \mathbb{Z}/N$ which has order n.

Since N and p_i are relatively prime, n is invertible in $\mathbb{Z}/N$. Thus the conditions of Theorem 3 are satisfied. □

Note that the transforms in the subfields can recursively apply Rader's Algorithm if the exponent of the prime divisor is one.

5.2 Practically Efficient NTTs

We now look at finding a suitable integer N such that the NTT in $\mathbb{Z}/N\mathbb{Z}$ admits an efficient implementation. In this doctoral thesis on the NTT implementations for microprocessors [8], Martin outlined several requirement for efficient implementations of the NTT. The first three concerning the existence of a transform are summarised by Theorem 6, followed by

- The transform length N should be highly composite to facilitate fast transform algorithms.
- Multiplication by the roots of unity must have a simple binary representation to facilitate fast multiplications.
- N has a simple binary representation to facilitate fast divisions.

We note that if the last point does not hold then we may look at using Montgomery Multiplication or Barret Reduction to covert divisors into multiplications [11].

Taking N to be a power of two would be the canonical choice as it admits the fastest arithmetic on modern computers. However, Theorem 6 tells us the maximum transform length would be 1 in that case; thus, it is completely unsuitable for an NTT. The next class of numbers we might consider are those of the form $2^k - 1$ for some $k \in \mathbb{N}$. Numbers of this form are known as *Mersenne numbers*, and their corresponding transforms are Mersenne Number Transforms (MNT). Mersenne Numbers admit highly efficient transforms though we will not discuss them here, and refer the reader to the discussion in [10] [23].

The next case we consider numbers of the form $2^k + 1$. When k itself is a power of two, it has less small divisors, e.g. if k is odd then 3 divides $2^k + 1$, hence the maximum transform length is 2. Numbers of the form $2^{2^\ell} + 1$ for $l \in \mathbb{N}$ are known as Fermat Numbers, and their corresponding transforms are Fermat Number Transforms (FMT).

The Fermat Numbers up to $\ell = 4$ are all primes, and therefore can be calculated very efficiently using the Rader Transform (Theorem 6). It can then be shown [6] that the roots can be given by the explicit formula $2^{2^{t-2}}(2^{2^{t-1}} - 1)$ and that all the factors of Fermat numbers are of the form $c2^{\ell+2} + 1$ for $c \in \mathbb{N}$. Therefore by Theorem 6 the maximum transform length is $2^{\ell+2}$.

When this field was first pioneered in the 1970s, a significant drawback of Fermat Number Transforms is that they are not as memory efficient as other candidates, as they require $k + 1$ bits of memory to hold only a little more than 2^k values. However this is not a real concern for modern computers. Agarwal and Burrus (Section VI [6]) suggested allowing the last number to overflow and rounding it to -1, 0 or -2 since there is a low probability (2^{-k} chance) that the forgotten number will arise. This technique was originally suggested in the context of Fourier Transforms for fast signal processing, and so such an error is likely to be insignificant in those contexts. However, it is unsuitable for our purposes where we are trying to obtain the true solutions.

Chapter 6

Asymptotic Bounds

In this chapter, we will look at one of the most significant recent developments in the field of integer multiplication, namely the algorithms presented by Harvey and van der Hoevern (2019, [27]) for multiplication of n-bit integers in $\mathcal{O}(n \log n)$ time in the Turing model. This has solved a long-standing problem in complexity theory and is conjectured by Schönage and Strassen to be asymptotically optimal [4].

Though this doesn't provide an immediate analogue for polynomial multiplication, it does given us a greater technique for evaluating complex convolution, which could lead to a generalisation for polynomials some time and so it is still worthy to discuss. Indeed, we could try to apply Kronecker substitution to convert polynomials in $\mathbb{Z}[x]$ into integers and apply this algorithm, however if B is a bound for the coefficients of the polynomial, this would give a complexity of

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

which is no better than the Schönage and Strassen algorithm from Chapter 4. This is due to the fact that when we use Kronecker substitution, we must pad sections of the integer with many zeros to account for the largest potential integer size to avoid a "carry" from occurring. It turns out that the amount of padding causes the complexity to be greater that the current best algorithms for multiplication in $\mathbb{Z}[x]$.

Along with their main paper, Harvey and van der Hoeven released a second paper [28] which presents another algorithm that achieves the same asymptotic bound and is easier to understand, but is conditional on an unproven hypothesis on the distribution of primes. This conditional algorithm has the advantage of being easily adapted for polynomial multiplication over more general coefficient

algebras [28]. In this thesis we have chosen to focus on the unconditional algorithm but we will point out similarities and differences between the two algorithms throughout our presentation.

Both methods take advantage of the fact that there exist higher dimensional DFTs in higher that are particularly efficient to compute. The algorithms proceed as follows: First convert the integer multiplication problem into polynomial multiplication in the ring $\mathbb{Z}[x]$, and then into a convolution in $\otimes_{i=1}^{d}\mathbb{C}^{s_i}$ for suitably picked primes $s_1, \ldots, s_n$. Since the transform lengths in each dimension are prime, the conditional algorithm applies Rader's trick to evaluate the convolution using a multidimensional FFT. However the FFT is only efficient when $s_1 - 1, \ldots, s_d - 1$ have many small prime divisors; a guarantee that has not yet been proven. The unconditional algorithm avoids this situation by using a technique the authors call *Gaussian Resampling*, to reduce the convolution of prime lengths, into a convolution in $\otimes_{i=1}^{t_i}\mathbb{C}^{t_i}$ where $t_i > s_i$ is the smallest power of two greater than s_i.

This technique relies heavily on the Archimedean property of the complex numbers, and so there is not an obvious generalisation of the algorithm for more general algebras e.g. integer rings. This contrasts the majority of previous attempts to improve the theoretical bound which kept the polynomials in $\mathbb{Z}[x]$ and used only discrete algebraic transformations to evaluate the multiplication to obtain an exact result at every step.

Since most of the calculations will be performed in the coefficient ring $\mathbb{C}$, we must carefully track the loss of precision in our intermediate results. As discussed in Chapter 4, using the standard FFT algorithm with enough precision to guarantee correctness increases the complexity of the overall algorithm past the $\mathcal{O}(n \log n)$ bound. In the Turing model, the complexity of fundamental operations such as addition and multiplication is directly proportional to the number of bits used to store the number. This means that as we increase the size of the size of our integers, simple arithmetic operations take longer. In the end, we must ensure the total error of the coefficients is strictly less than $1/2$ so that rounding to nearest integer returns the correct result.

This section will prove the following theorem

Theorem 7 (Theorem 1 [27])**.** Multiplication of integers of size n can be calculated in time $\mathcal{O}(n \log n)$ in the Turing model.

First we will present a lemma which allows us to convert polynomial multiplication in a single variable into multiplication in multiple variables.

Lemma 4. For distinct primes $s_1, \ldots, s_d$

$$\frac{\mathbb{Z}[x]}{(x^{s_1 \cdots s_d} - 1)} \cong \frac{\mathbb{Z}[x_1, \ldots, x_d]}{(x_1^{s_1} - 1, \ldots, x_d^{s_d} - 1)}. \tag{6.1}$$

Proof. Observe that the group of monomials in $\mathbb{Z}[x]/(x^{s_1 \cdots s_d} - 1)$ under multiplication is isomorphic to the group $\mathbb{Z}/(s_1 \ldots s_d \mathbb{Z})$ under addition. By the Chinese Remainder theorem

$$\frac{\mathbb{Z}}{s_1 \ldots s_d \mathbb{Z}} \cong \frac{\mathbb{Z}}{s_1 \mathbb{Z}} \times \cdots \times \frac{\mathbb{Z}}{s_d \mathbb{Z}}$$

The same reasoning shows that the right hand side is isomorphic to the group of monomials in $\mathbb{Z}[x_1, \ldots, x_d]/(x_1^{s_1} - 1, \ldots, x_d^{s_d} - 1)$.
Thus there exists an isomorphism, $x \mapsto x_1 \cdots x_d$, from the monomials of $\mathbb{Z}[x]/(x^{s_1 \cdots s_d} - 1)$ to the group of monomials of $\mathbb{Z}[x_1, \ldots, x_d]/(x_1^{s_1} - 1, \ldots, x_d^{s_d} - 1)$.
This extends naturally to an isomorphism between the two rings in (6.1). □

Note that we could formulate this map as Kronecker substitution, but from the theory we have developed so far that would not give us the isomorphism needed for the algorithm.

Now we may introduce a high-level overview of the main algorithm. Let $a, b \in \mathbb{Z}$ be n-bit integers, then to calculate the product $c = ab$:

1. Choose distinct primes $s_1, \ldots, s_d \approx n^{1/d}$ and $s_1 \cdots s_d \geq n/\lceil \log_2 n \rceil$ (subject to certain conditions we will explain later).

   Convert the a and b into polynomials in $\mathbb{Z}[x]/(x^{s_1 \cdots s_d} - 1)$ via Kronecker substitution with $x = 2^{\lceil \log_2 n \rceil}$. Then apply Lemma 4 to transform the polynomials into the multivariate polynomials in $\mathbb{Z}[x]/(x^{s_1 - 1, \ldots, x^{s_d} - 1})$.

2. By viewing $\mathbb{Z}$ as a subring of $\mathbb{C}$, convert the multivariate polynomials into a d-dimensional convolution over $\mathbb{C}$ via Proposition 3.

3. Use *Gaussian Resampling* to approximate the convolution in $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$ by a convolution $\otimes_{i=1}^{d} \mathbb{C}^{t_i}$ where t_i is the next power of two greater than s_i.

4. Efficiently evaluate the convolution in $\otimes_{i=1}^{d} \mathbb{C}^{t_i}$ and convert the convolution back into $\otimes_{i=1}^{d} \mathbb{C}^{s_i}$.

5. Convert the result back into $\mathbb{Z}[x]$, and finally recover the solution in $\mathbb{Z}$.

Steps 3 and 4 are the only difficult steps that cannot be accomplished from the theory in previous chapters. In this section, we will cover all aspects of the proof

presented by Harvey and van der Hoeven, excluding the details of the Gaussian Resampling technique used in Step 3, and certain number theoretic results that show the existence of a certain distribution of primes, as they assume a level of understanding of Fourier analysis and number theory that is beyond the material presented in this thesis. Section 6.1 will introduce the notation and formulate the error model used throughout the rest of the chapter. Section 6.2 will develop the tools necessary to perform step 4, and Section 6.3 will prove Theorem 7.

6.1 Notation

Fix a precision of p bits.

The main two coefficient algebras we use throughout this chapter are $\mathbb{C}$ and $\mathscr{R} = \mathbb{C}[y]/(y^r + 1)$ where $r < 2^{p-1}$ is a power of two. Note that this is the same algebra as in the Schönage-Strassen integer multiplication algorithm and is used a similar way. We will preference performing DFTs in the $\mathscr{R}$ coefficient ring (also known as *synthetic DFTs*) as multiplications by the roots of unity simply amounts to a nega-cyclic permutation of the coefficients. Therefore it is more computationally efficient than the case for $\mathbb{C}$, and incurs no loss of precision.

If V is an m-dimensional vector space and $v = (v_1, \ldots, v_m) \in V$ with respect to some basis, we define the norm

$$\|v\| = \max_j \|v_i\|.$$

This is an appropriate choice of normal since if we make an approximation $\tilde{c}$ for $c = ab$, and we can show that $\|c - \tilde{c}\| < \frac{1}{2}$, then we can conclude that rounding the results to the nearest integer will produce the correct answer.

We write $\tilde{\mathbb{C}}$ to denote the space of elements in $\mathbb{C}$ with a fixed precision of p, that is,

$$\tilde{\mathbb{C}} = \{x + iy \mid x, y \in \mathbb{Z}, |x|, |y| < 2^p\}.$$

Let $\mathbb{C}_\circ$ denote the for the unit ball in $\mathbb{C}$, and so $\tilde{\mathbb{C}}_\circ = 2^{-p}\tilde{\mathbb{C}}$ and $V_\circ$ the unit ball in V with respect to the norm above.

Now define ε as the error function that measures the error of an approximation. More formally, for an approximation $\tilde{w} \in \tilde{V}_\circ$ of $w \in V_\circ$ we express the error associated with $\tilde{w}$ as

$$\varepsilon(\tilde{w}) = 2^p \|\tilde{w} - w\|,$$

Notice that the error is measured in multiples of 2^p. So if the approximation is incorrect by one bit, then the difference between the approximation and the exact

solution is 2^{-p} and the error given by ε is one.

Using this definition, we can derive several fundamental properties of ε under arithmetic operations which we will state without proof. For a more formal explanation, see (Section 2 [27]). Notice that all the operation are normalised to keep the norm of the results less than 1, so we can keep the result it in p bits of precision. Indeed, we will need to restructure some algorithms previously presented in this paper to accommodate for normalised intermediate expressions.

Note that in the standard DFT of length n, $\mathcal{F}_n$, we have $\|\mathcal{F}_n\| \leq n$. So we will perform the normalised DFT, given by

$$(\mathcal{F}_n u)_k = \frac{1}{n} \sum_{j=0}^{n-1} a_j \omega_N^{-jk}.$$

for $u \in K^n$.

The convolution formula then becomes

$$\frac{1}{n}(u * v) = N\mathcal{F}_n^{-1}(\mathcal{F}_n u \cdot \mathcal{F}_n v).$$

This then generalises to the multi-dimensional case $F_{n_1,\ldots,n_d}$ as

$$(\mathcal{F}_n u)_k = \frac{1}{n_1 \cdots n_d} \sum_{j_1=0}^{n_1-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,\ldots,j_d} \omega_{n_1}^{-j_1 k} \omega_{n_d}^{-j_d k}$$

and

$$\frac{1}{n_1 \cdots n_d}(f * g) = (n_1 * \cdots * n_d)\mathcal{F}^{-1}_{n_1,\ldots,n_d}(\mathcal{F}_{n_1,\ldots,n_d} u \cdot \mathcal{F}_{n_1,\ldots,n_d} v) \tag{6.2}$$

Throughout this chapter we will be careful to ensure that the error at every step is small enough to guarantee a correct recovering of the final result. Thus we will frequently use the following Lemmas whose proofs may be found in Section 2 of [27].

Proposition 5. Let V be a m-dimensional vector space. Assume we are given $u, v \in V_\circ$ and approximations $\tilde{u}, \tilde{v} \in \tilde{V}_\circ$, and $1 \leq c \leq 2^p$, we have

1. (E.1 Addition) We may compute an approximation $\tilde{w} \in \tilde{V}_\circ$ for $w = \frac{1}{2}(\tilde{u} \pm \tilde{v}) \in V_\circ$ such that $\varepsilon(\tilde{w}) < 1$ in time $O(mp)$.

2. (E.2 Scaling) If $\|u\| \leq c^{-1}$, and $w = cu \in V_\circ$, then we may compute and approximation $\tilde{w} \in \tilde{V}_\circ$ such that $\varepsilon(\tilde{w}) < 2c \cdot \varepsilon(\tilde{u}) + 3$ and in time $O(mp^{1+\delta})$.

3. (E.3 $\mathbb{C}$ Multiplication) If $V = \mathbb{C}$, we may compute an approximation $\tilde{w} \in \tilde{\mathbb{C}}_\circ$ for $w = uv$ such that $\epsilon(\tilde{w}) < 2$ and in time $\mathcal{O}(p^{1+\delta})$.

4. (E.4 $\mathscr{R}$ Multiplication) If $V = \mathscr{R}$ we may compute an approximation $\tilde{w} \in \tilde{\mathscr{R}}_\circ$ for $w = uv/r$ such that $\varepsilon(\tilde{w}) < 2$ and in time $4M(3rp) + \mathcal{O}(rp)$.

Let $\mathcal{F}_{t_1,\ldots,t_d}$ denote the DFT of $\mathbb{C}^{t_1} \otimes \cdots \otimes \mathbb{C}^{t_d}$ and let $\mathcal{G}_{t_1,\ldots,t_d}$ denote the DFT of $\mathscr{R}^{t_1} \otimes \cdots \otimes \mathscr{R}^{t_d}$ (also known as a *synthetic DFT*). We let $\mathcal{F}^{-1}_{t_1,\ldots,t_d}$ and $\mathcal{G}^{-1}_{t_1,\ldots,t_d}$ denote their inverses. We also denote the convolution functions in $\mathbb{C}$ and $\mathscr{R}$ as $\mathcal{M}_{\mathbb{C}}$ and $\mathcal{M}_{\mathscr{R}}$ respectively.

Let $\mathcal{A} : V \to W$ be a $\mathbb{C}$-linear map between finite-dimensional vector spaces V and W. We use the operator norm

$$\|A\| := \sup_{v \in V_\circ} \|\mathcal{A}v\|,$$

and define the associated error of a linear map as

$$\varepsilon(\tilde{\mathcal{A}}) := 2^p \max_{v \in \tilde{V}_0} \|\tilde{\mathcal{A}}v - \mathcal{A}v\|.$$

Proposition 6. Let $\mathcal{A} : V \to W$ and $\mathcal{B} : V \to W$ be $\mathbb{C}$-linear maps such that $\|\mathcal{A}\|, \|\mathcal{B}\| \leq 1$ and let $v \in V_\circ$. Let $\tilde{\mathcal{A}} : \tilde{V}_\circ \to \tilde{W}_\circ$, $\tilde{\mathcal{B}} : \tilde{V}_\circ \to \tilde{W}_\circ$ be numerical approximations for $\mathcal{A}$ and $\mathcal{B}$. Then

1. (E.5 Linear map) For $w \in \mathcal{A}v \in W_\circ$ we may construct a numerical approximation $\tilde{w} = \tilde{\mathcal{A}}v$ with $\varepsilon(\tilde{w}) \leq \epsilon(\tilde{\mathcal{A}}) + \varepsilon(\tilde{v})$.

2. (E.6 Composition) For $\mathcal{C} := \mathcal{B}\mathcal{A}$ we may construct a numerical approximation $\tilde{\mathcal{C}} = \tilde{\mathcal{A}}\tilde{\mathcal{B}}$ such that $\varepsilon(\tilde{\mathcal{C}}) \leq \varepsilon(\tilde{\mathcal{A}}) + \varepsilon(\tilde{\mathcal{B}})$.

Lemma 5 (Error of Bilinear Maps)**.** A similar result holds for bilinear maps, define $\|\mathcal{A}\| = \sup_{u \in U_\circ, v \in V_\circ} \|\mathcal{A}(u,v)\|$. Then if $\mathcal{A} : U \times V \to W$ is a $\mathbb{C}$-bilinear map with $\|\mathcal{A}\| \leq 1$ and $u \in U_\circ$ and $v \in V_\circ$, then we may construct an approximation $\tilde{w} = \tilde{\mathcal{A}}(\tilde{u}, \tilde{v}) \in \tilde{W}_\circ$ for $w = \mathcal{A}(u,v) \in W_\circ$ such that $\varepsilon(\tilde{w}) \leq \varepsilon(\tilde{\mathcal{A}}) + \varepsilon(\tilde{u}) + \varepsilon(\tilde{v})$.

Lemma 6. Let $k \geq 1$ and j be integers such that $0 \leq j < k$.

1. (E.7 Complex Exponential) If $w = e^{2\pi i j/k} \in \mathbb{C}_0$ then we may compute an approximation $\tilde{w} \in \tilde{C}_\circ$ such that $\varepsilon(\tilde{w}) < 2$ in time $\mathcal{O}(\max(p, \log k)^{1+\delta})$

2. (E.8 Real Exponential) If $w = e^{-\pi j/k} \in \mathbb{C}_\circ$ and $j \geq 0$, then we may compute $\varepsilon(\tilde{w}) < 2$ in time $\mathcal{O}(\max(p, \log(|j|+1), \log k)^{1+\delta})$. Similarly, if $w = 2^{-\sigma} e^{\pi j/k} \in \mathbb{C}_0$, then we may compute $\varepsilon(\tilde{w}) < 2$ in time $\mathcal{O}(\max(p, \log k)^{1+\delta})$.

Lemma 7 (E.9 Tensor Products)**.** Let R be a coefficient ring of dimension r, and consider the maps $\mathcal{A}_i : R^{m_i} \to R^{n_i}$ to be an R-linear map with $\|\mathcal{A}_i\| \leq 1$ and let $\tilde{\mathcal{A}}_i$ be a numerical approximation. Let $\mathcal{A} := \otimes_i \mathcal{A}_i : \otimes_i R^{m_i} \to \otimes_i R^{n_i}$. Then we can construct a numerical approximation $\tilde{\mathcal{A}}$ such that $\varepsilon(\tilde{\mathcal{A}}) \leq \sum_i \varepsilon(\tilde{\mathcal{A}}_i)$ and

$$C(\tilde{\mathcal{A}}) \leq M \sum_i \frac{C(\tilde{A}_i)}{m_i} + \mathcal{O}(Mrp \log M).$$

6.2 Transforms for Powers of Two

In this section we show how to efficiently evaluate the convolution of $\mathbb{C}^{t_1} \otimes \cdots \otimes \mathbb{C}^{t_d}$ for $t_1, \ldots, t_d$ powers of two, such that the error is sufficiently minimised.

Throughout this chapter let $r = t_d$ and $\mathscr{R} = \mathbb{C}[y]/(y^r + 1)$.

Theorem 8 (Theorem 3.1 in [27])**.** Let $d \geq 2$ and $t_1, \ldots, t_d$ be powers of two, let $t_d \geq \cdots \geq t_1 \geq 2$ and write $T = t_1 \ldots t_d$. Choose a precision p such that $T < 2^p$. Then we may construct a numerical approximation

$$\tilde{\mathcal{F}}_{t_1,\ldots,t_d} : \otimes_i \tilde{\mathbb{C}}_\circ^{t_i} \to \otimes_i \tilde{\mathbb{C}}_\circ^{t_i}$$

for $\mathcal{F}_{t_1,\ldots,t_d}$ such that $\epsilon(\tilde{\mathcal{F}}_{t_1,\ldots,t_d}) < 8T \log T$ and

$$C(\tilde{\mathcal{F}}_{t_1,\ldots,t_d}) < \frac{4T}{t_d} M(3t_d p) + \mathcal{O}(Tp \log T + Tp^{1+\delta}).$$

The first term comes from the pointwise multiplications in $\tilde{\mathbb{C}}_\circ$, which are handled recursively. The $Tp \log T$ term comes from applying the FFT after applying Bluestein's trick. The remaining $Tp^{1+\delta}$ term comes from an additional scaling step we must perform at the end of evaluating the DFTs in order to normalise the result.

The construction of the approximation in Theorem 8 will be construct via a set of five proposition, structured as shown.

1. Construct an approximation for a uni-dimensional FFT $\mathcal{G}_i$.
2. Generalise to a multi-dimensional FFT in $\otimes_{i=1}^{d-1} \mathscr{R}_i$.
3. Evaluate the convolution in $\otimes_{i=1}^{d-1} \mathscr{R}^{s_i}$.
4. Evaluate the convolution in $\otimes_{i=1}^{d-1} \mathbb{C}^{s_i}$.
5. Evaluate the FFT: $\mathcal{F}_{t_1,\ldots,t_d}$.

At first, it may seem counter intuitive to first calculate a DFT, use it to calculate a convolution, then use the convolution to evaluate a DFT. The reason for all these steps is so that we can perform the majority of the computations inside the coefficient field $\mathscr{R}$ to take advantage of the synthetic roots of unity that admit efficient arithmetic and incur no loss in precision.

Lemma 8 (Lemma 3.2 [27])**.** For $t \in \{2, 4, \dots, 2r\}$, we may construct a numerical approximation $\tilde{\mathcal{G}}_t : \mathscr{R}^t \to \mathscr{R}^t$ for $\mathcal{G}_t$ such that $\epsilon(\tilde{\mathcal{G}}_t) \leq \log t$ and $\mathcal{C}(\tilde{\mathcal{G}}_t) = \mathcal{O}(trp \log 2t)$.

Proof. Let $t \in \{2, 4, \dots, 2r\}$ and write $\omega_t = y^{2r/t}$ for the primitive root of unity of order $t \in \mathscr{R}$.

First we will need to evaluate the DFT in a way that is normalised. Rather that partitioning the sum into even and odd sample, we split the sum into its top and bottom half[1], this gives

$$\begin{aligned}(\mathcal{G}_t u)_k &= \frac{1}{t}\sum_{j=0}^{t-1} a_j \omega_t^{-jk} + \frac{1}{t}\sum_{j=0}^{t/2-1} a_{j+t/2}\omega^{-j(k+t/2} \\ &= \frac{1}{t/2}\sum_{j=0}^{t-1}\frac{1}{2}(a_j + (-1)^j a_{j+t/2})\omega_t^{-jk}\end{aligned}$$

Then for $0 \leq p < t/2$ we have

$$(\mathcal{G}_t u)_{2p} = \frac{1}{t/2}\sum_{j=0}^{t-1}\frac{1}{2}(a_j + a_{j+t/2})\omega_{t/2}^{-pk}, \qquad (\mathcal{G}_t u)_{2p+1} = \frac{1}{t/2}\sum_{j=0}^{t-1}(\frac{1}{2}(a_j - a_{j+t/2})\omega_t^j)\omega_{t/2}^{pk}$$

Thus we broken the problem into two sub-DFTs of the vectors $f, g \in \mathbb{C}^{t/2}$ given by $f_j = a_j + a_{j+t/2}$ and $g_j = \frac{1}{2}(a_j - a_{j+t/2})\omega_t^j$.

We can then apply (E.1 Addition) to obtain an approximation $\tilde{f} \in \tilde{\mathbb{C}}_\circ^{t/2}$ for f such that $\varepsilon(\tilde{f}) < 1$ in $\mathcal{O}(trp)$ time. The same can be done to compute an approximation $\tilde{g} \in \tilde{\mathbb{C}}_\circ^{t/2}$ though we must also multiply by roots of unity, we obtain an addition cost of $\mathcal{O}(trp)$ (but no further loss in precision). Since we perform this operation at $\log_2 t$ recursion levels, we can accumulate at most 1 bit of error each time. Therefore $\varepsilon(\tilde{\mathcal{G}}_t) \leq \log_2 t$.

We obtain the exact same recursive formula as in our original formulation of the FFT, hence $\mathcal{C}(\tilde{\mathcal{G}}_t) = \mathcal{O}(trp \log t)$. □

[1]This is know as the *decimation in frequency* FFT, our previous formulation in Chapter 4 is know as the *decimation in time* FFT

Proposition 7 (Proposition 3.3 [27])**.** Let $t_1, \dots, t_d$ and T be as in Theorem 3.1. We may construct a numerical approximation $\mathcal{G}_{t_1,\dots,t_{d-1}} : \otimes_i \tilde{\mathscr{R}}^{t_i} \to \otimes_i \tilde{\mathscr{R}}^{t_i}$ for $\mathcal{G}_{t_1,\dots,t_{d-1}}$ such that $\epsilon(\tilde{\mathcal{G}}_{t_1,\dots,t_{d-1}}) < \log_2 T$ and $\mathcal{C}(\tilde{\mathcal{G}}_{t_1,\dots,t_{d-1}}) \in \mathcal{O}(Tp \log T)$.

Proof. We can view $\mathcal{G}_{t_1,\dots,t_{d-1}}$ as a multi-linear map. Using Lemma 7 with $M = t_1 \cdots t_{d-1} = T/r$, we have

$$\mathcal{C}(\tilde{G}_{t_1,\cdots,t_{d-1}}) \leq \frac{T}{r} \sum_{i=1}^{d-1} \frac{\mathcal{C}(\tilde{\mathcal{G}}_{t_i})}{t_i} + \mathcal{O}(\left(\tfrac{T}{r}\right) rp \log\left(\tfrac{T}{r}\right))$$

Applying Lemma 8 we may evaluate the cost of the individual DFTs as

$$\begin{aligned}
\mathcal{C}(\tilde{G}_{t_1,\cdots,t_{d-1}}) &= \frac{T}{r} \sum_{i=1}^{d-1} \mathcal{O}(rp \log 2t_i) + \mathcal{O}(Tp \log T) \\
&= \mathcal{O}(Tp \sum_{i}^{d-1} \log 2t_i) + \mathcal{O}(Tp \log T) \\
&= \mathcal{O}(Tp \log T).
\end{aligned}$$

Lemma 7 also gives the bound the error associated with $\varepsilon(\tilde{\mathcal{G}}_{t_1,\dots,t_{d-1}}) \leq \sum_{i=1}^{d-1} \tilde{\mathcal{G}}_{t_i} = \sum_{i=1}^{d-1} \log_2 t_i$. □

Proposition 8 (Proposition 3.4 [27])**.** Let $t_1, \dots, t_d$ and T be as in Theorem 3.1. Then we can construct a numerical approximation $\tilde{\mathcal{M}}_{\mathscr{R}} : \otimes_i \tilde{\mathscr{R}}^{t_i} \otimes_i \tilde{\mathscr{R}}^{t_i} \to \otimes_i \tilde{\mathscr{R}}^{t_i}$ for the convolution function $\mathcal{M}_{\mathscr{R}}$ such that $\epsilon(\tilde{\mathcal{M}}_{\mathscr{R}}) < 3T \log_2 T + 2T + 3$ and

$$\mathcal{C}(\tilde{\mathcal{M}}_{\mathscr{R}}) < \frac{4T}{r} M(3rp) + \mathcal{O}(Tp \log T + Tp^{1+\delta}).$$

Proof. Using the formula for our normalised convolution (6.2)

$$\frac{1}{t_1 \cdots t_{d-1}} u * v = (t_1 \cdots t_{d-1})\, \mathcal{G}^{-1}_{t_1,\dots,t_{d-1}}((\mathcal{G}_{t_1,\dots,t_{d-1}} u) \cdot (\mathcal{G}_{t_1,\dots,t_{d-1}} v))).$$

We need to ensure that the intermediate calculations have norm at most 1 to fit inside the p bits of precision. However, if computed as in previous chapters, the result of the pointwise multiplications of the two forward transforms may have norm $r > 1$. To remedy this we divide both sides by r to obtain $w = (T/r)w'$, where

$$w' = \mathcal{G}^{-1}_{t_1,\dots,t_{d-1}}(\tfrac{1}{r}(\mathcal{G}_{t_1,\dots,t_{d-1}} u) \cdot (\mathcal{G}_{t_1,\dots,t_{d-1}} v)). \tag{6.3}$$

We use Proposition 7 to compute approximations $u' = \tilde{\mathcal{G}}_{t_1,\dots,t_{d-1}} u \in \otimes_i \tilde{\mathscr{R}}^{t_i}$ and $v' = \tilde{\mathcal{G}}_{t_1,\dots,t_{d-1}} v \in \otimes_i \mathscr{R}^{t_i}$. The cost of this step is $\mathcal{O}(Tp \log T)$.

Next we must handle the pointwise multiplications between the two transforms. The multiplication map $\mathcal{A} : \mathscr{R} \times \mathscr{R} \to \mathscr{R}$ given by $\mathcal{A}(a, b) = ab/r$ is a bilinear map with $\|\mathcal{A}\| \leq 1$ and so we may compute an approximation $\tilde{\mathcal{A}}$ by Lemma 5 such that $\varepsilon(\tilde{\mathcal{A}}) < 2$ and $\tilde{v} = \tilde{\mathcal{A}}(\tilde{u}, \tilde{v})$ has $\varepsilon(\tilde{v}) = \varepsilon(\tilde{\mathcal{A}}) + \varepsilon(\tilde{u}) + \varepsilon(\tilde{v}) = 2\log_2 T + 2$. The combined complexity of these steps (the combined complexity of the approximation lemmas) is then

$$\frac{T}{r}(4M(3rp) + \mathcal{O}(rp)) = \frac{4T}{r}M(3rp) + O(Tp).$$

The inverse transform follows in the same way with the same complexity and error bounds as the forward transforms.

Finally, we must scale by T/r, since we know that $\|w\| \leq 1$ we can apply the approximation Lemma (E.2 Scaling) to obtain $w = (T/r)w'$ such that $\varepsilon(\tilde{w}) < 2(T/r)\varepsilon(\tilde{w}') + 3 \leq T(3\log_2 T + 2) + 3 = 3T\log_2 T + 5$ in time $\mathcal{O}(Tp^{1+\delta})$. □

Proposition 9 (Proposition 3.5 [27])**.** Let $t_1, \ldots, t_d$ be as in Theorem 3.1. We may construct a numerical approximation $\tilde{\mathcal{M}}_{\mathbb{C}} : \otimes_i \tilde{\mathbb{C}}_\circ^{t_i} \times \tilde{\mathbb{C}}_\circ^{t_i} \to \tilde{\mathbb{C}}_\circ^{t_i}$ for $\mathcal{M}_{\mathbb{C}}$ such that $\varepsilon(\tilde{\mathcal{M}}_{\mathbb{C}}) < 3T\log_2 T + 2T + 15$ and

$$C(\tilde{\mathcal{M}}_{\mathbb{C}}) < \frac{4T}{r}M(3rp) + \mathcal{O}(Tp\log T + Tp^{1+\delta}).$$

Proof. Note that since $\mathscr{R} = \mathbb{C}[y]/(y^r - 1)$ and $r = t_d$, we have the isomorphism

$$\mathcal{T} : \frac{\mathbb{C}[x_1, \ldots, x_d]}{x_1^{t_1} - 1, \ldots, x_d^{t_d} - 1} \to \frac{\mathscr{R}[x_1, \ldots, x_{d-1}]}{x_1^{t_1} - 1, \ldots, x_d^{t_{d-1}} - 1}$$

given by the map $\mathcal{T}u((x_1, \ldots, x_d)) \to u(x_1, \ldots, x_{d-1}, \omega y)$ where ω is a $2r$-root of unity in $\mathbb{C}$. This induces the isomorphism

$$\mathcal{M}_{\mathbb{C}}(u, v) = \mathcal{T}^{-1}(\mathcal{M}_{\mathscr{R}}(\mathcal{T}u, \mathcal{T}v)) \qquad u, v \in \otimes_i \mathbb{C}^{t_i}.$$

Now we construct numerical approximations for $\mathcal{T}$ and $\mathcal{T}^{-1}$. Let $\mathcal{S} : \mathbb{C}_\circ^r \to \mathscr{R}_\circ$ denote the map that sends $x_d \mapsto \omega y$. Then we may construct an approximation $\tilde{\omega}_j$ for each $\omega_j = e^{\pi i j/r} \in \mathbb{C}_\circ$ and then using Lemma E.2 we compute an approximation $\tilde{v}_j$ for $v_j = \omega_j u_j$. We obtained $\varepsilon(\tilde{\omega}_j) < 2$ and so $\varepsilon(\tilde{v}_j) < \varepsilon(\tilde{\omega}_j) + 2 < 4$. Hence $\varepsilon(\mathcal{S}) < 4$ and $C(\tilde{\mathcal{S}}) = O(rp^{1+\delta})$. So by applying $\tilde{\mathcal{S}}$ separately to the coefficient of each $x_1^{j_1} \cdots x_{d-1}^{j_{d-1}}$ we obtain an approximation $\tilde{\mathcal{T}} : \otimes_{i=1}^d \tilde{\mathbb{C}}_\circ^{t_i} \to \otimes_{i=1}^{d-1} \tilde{\mathscr{R}}_\circ^{t_i}$ such that $\varepsilon(\tilde{\mathcal{T}}) < 4$ and $C(\tilde{\mathcal{T}}) = \mathcal{O}(Tp^{1+\delta})$. The inverse is computed similarly with the same complexity and error bound.

Thus we have

$$\varepsilon(\tilde{\mathcal{M}}_{\mathbb{C}}) \leq \varepsilon(\tilde{\mathcal{M}}_{\mathscr{R}}) + \varepsilon(\tilde{\mathcal{U}}) + \varepsilon(\tilde{\mathcal{T}}) + \varepsilon(\tilde{\mathcal{T}}) < 3T\log_2 T + 2T + 15$$

□

Lemma 9. Let $\phi : \mathbb{Z}^d \to \mathbb{C}$ be the function given by

$$\phi(j_1, \ldots, j_d) = r\left(\frac{j_1^2}{t_1} + \cdots + \frac{j_d^2}{t_d}\right) \mod 2r$$

Then computing the sequence $(\phi_r(j_1, \ldots, j_d))_{j_1<t_1,\ldots,j_d<t_d}$ requires $\mathcal{O}(Tp^{1+\delta})$ time.

Proof. First $j_1 = j_2 = \ldots = j_d = 0$. Then we follow this procedure. We compute the $\lceil \log_2 d \rceil$ partial sums.

$$\frac{r}{2}(j_1^2/t_1 + \cdots + j_{\lfloor d/2 \rfloor}^2/t_{\lfloor d/2 \rfloor}), \frac{r}{4}(j_{\lfloor d/2 \rfloor+1}/t_{\lfloor d/2 \rfloor+1}^2 + \cdots + j_{\lfloor d/4 \rfloor+1}^2/t_{\lfloor d/4 \rfloor+1}) + \cdot \left(\frac{j_{d-1}^2}{t_{d-1}}\right)$$

Then we compute j_d^2/t_d for all $1 \leq j_d \leq t_d$ and use the partial sums to evaluate the entire sum. We must operate on the partial sums since our approximation lemma (E.1 Addition) is only formulated to work of two values at a time. This will take $\mathcal{O}(\log r)$. Once we have computed all of the j_d values, we must recompute the partial sum with the next value of $j_1, \ldots, j_{d-1}$. Recomputing the partial sums will take $O(r)$ each and there are $\mathcal{O}(T/r)$ of them. Hence they will take $\mathcal{O}(T)$ time in total to compute. Thus computing all the sums in the table will take $\mathcal{O}(T \log r)$ time. □

Finally, we may prove the main theorem of this section

Proof of Theorem 8. Let $u \in \otimes_i \tilde{\mathbb{C}}_\circ^{t_i}$. Here we will apply Bluestein's trick for converting DFTs into convolutions. Adapting Bluestein's algorithm for normalised DFTs gives

$$v = \bar{a} \cdot (\tfrac{1}{T} a * (\bar{a} \cdot u))$$

where $a_{j_1,\ldots,j_d} := e^{\pi i(j_1^2/t_1 + \cdots + j_d^2/t_d)} \in \mathbb{C}_\circ$.

First we must compute a. We can write a as $a_{j_1,\ldots,j_d} = e^{2\pi i \eta_{j_1,\ldots,j_d}/2r}$ where

$$\eta_{j_1,\ldots,j_d} = r\left(\frac{j_1^2}{t_1} + \cdots + \frac{j_d^2}{t_d}\right) \mod 2r.$$

We evaluate $\eta_{j_1,\ldots,j_d}$ separately so as to keep the intermediate expressions normalised. Lemma 9 tells us that we may compute η in $\mathcal{O}(T \log r) = \mathcal{O}(Tp)$ time.

We then apply the approximation lemma (E.8 Real Exponential) to compute an approximation $\tilde{a}_{j_1,\ldots,j_d} \in \tilde{\mathbb{C}}_\circ$ such that $\varepsilon(\tilde{a}_{j_1,\ldots,j_d}) < 2$ in time $\mathcal{O}(p^{1+\delta})$.

Now we compute the approximation $\tilde{b} \in \otimes_i \tilde{\mathbb{C}}_\circ^{t_i}$ for $b = \bar{a} \cdot u$ with $\varepsilon(\tilde{b}) < \varepsilon(\tilde{a}) + 2 < 4$ in time $\mathcal{O}(Tp^{1+\delta})$.

Apply Proposition 9 to compute an approximation $\tilde{c} \in \otimes_i \tilde{\mathbb{C}}_\circ^{t_i}$ for $c = \frac{1}{T} a * b$. This requires time $(4T/r)M(3rp)+\mathcal{O}(Tp\log T+Tp^{1+\delta})$ and by the approximation lemma for bilinear functions (Lemma 5), we have

$$\varepsilon(\tilde{c}) \leq \varepsilon(\tilde{\mathcal{M}}_{\mathbb{C}}) + \varepsilon(\tilde{a}) + \varepsilon(\tilde{b}) < 3T\log_2 T + 2T + 21.$$

Finally we multiply $\tilde{v} = \tilde{a} \cdot \tilde{c}$ to obtain the final result. From Lemma 5 for bilinear functions we obtain the error bound

$$\varepsilon(\tilde{v}) \leq \varepsilon(\tilde{a}) + \varepsilon(\tilde{c}) + 2 < 3T\log_2 T + 2T + 25$$

in time $\mathcal{O}(Tp^{1+\delta})$. Since $T = t_1 \cdots t_d \geq 4$ we have we have $2T + 25 < 5T\log_2 T$, and thus $\varepsilon(\tilde{v}) < 8T\log_2 T$. □

We note that the contents of this section do not make any major changes from the techniques outlined in Chapter 4 for using the FFT to evaluate convolutions. The only changes are those made to normalise the result, and use synthetic DFTs to control the error.

6.3 Proof of Main Theorem

In order to prove Theorem 7, we first need to present the main result of Harvey and van der Hoeven's *Gaussian Resampling* technique to convert a DFT of order $s_1 \times \cdots \times s_d$ where s_i are prime, to one of order $t_1 \times \cdots \times t_d$, where $t_i > s_i$ is the next power of two greater than s_i, to allow us to apply Theorem 8. This is the main point of divergence between the conditional and unconditional algorithms. Where the unconditional algorithm using Gaussian Resampling to make this conversion, the conditional algorithm uses Rader's transform. Since we are not proving the theorem here, we will simplify it for the sake of clarity.

Theorem 9 (Theorem 4.1 in nlogn)**.** Let $d \geq 1$, let $s_1, \ldots, s_d$ and $t_1, \ldots, t_d$ be integers such that $2 \leq s_i < t_i < 2^p$ and $\gcd(s_i, t_i) = 1$ and let $T = t_1 \ldots t_d$. Suppose $s_i/t_i - 1 > p^{-1/6}$ for all i. Then there exists a $\gamma < 2dp^{7/24}$ and linear maps $\mathcal{A} : \otimes_i \mathbb{C}^{s_i} \to \otimes_i \mathbb{C}^{t_i}$ and $\mathcal{B} : \otimes_i \mathbb{C}^{t_i} \to \otimes_i \mathbb{C}^{s_i}$ with $\|\mathcal{A}\|, \|\mathcal{B}\| \leq 1$ such that

$$\mathcal{F}_{s_1,\ldots,s_d} = 2^\gamma \mathcal{B}\mathcal{F}_{t_1,\ldots,t_d}\mathcal{A}$$

Moreover we can construct numerical numerical approximations $\tilde{\mathcal{A}} : \otimes_i \tilde{\mathbb{C}}_0^{s_i} \to \otimes_i \tilde{\mathbb{C}}_0^{t_i}$ and $\tilde{\mathcal{B}} : \otimes_i \tilde{\mathbb{C}}_0^{t_i} \to \otimes_i \tilde{\mathbb{C}}_0^{s_i}$ such that $\epsilon(\tilde{\mathcal{A}}), \epsilon(\tilde{\mathcal{B}}) < dp^2$ and

$$C(\tilde{\mathcal{A}}), C(\tilde{\mathcal{B}}) = \mathcal{O}(dTp^{\frac{43}{24}+\delta} + Tp\log T).$$

Now we will set up the parameters to prove the theorem.

Let $b = \lceil \log_2 n \rceil \geq d^{12} \geq 4096$ be the size of the coefficients after applying Kronecker substitution to convert the integers into polynomials in $\mathbb{Z}[x]$. Fix the working precision to be $p = 6b$.

Let T be the unique power of two in the interval

$$4n/b \leq T < 8n/b \tag{6.4}$$

In other words, T is the smallest power of two that is greater than the degree of the result of the polynomial multiplication in $\mathbb{Z}[x]$; it follows that $T < 2^p$.

Let r be the unique power of two in the interval,

$$T^{1/d} \leq r < 2T^{1/d}.$$

Now we construct a factorisation $T = t_1 \cdots t_d$ that satisfies the hypotheses of Theorem 3.1. Let $d' := \log_2(r^d/T)$. As $T \leq r^d < 2^d T$ we have $1 \leq r^d/T < 2^d$ and hence $0 \leq d' < d$. Define

$$t_1, \ldots, t_{d'} := \frac{r}{2}, \qquad t_{d'+1}, \ldots, t_d := r$$

Then $t_d \geq \cdots \geq t_1 \geq 2$ and $t_1 \cdots t_d = T$.

We will now present a number theoretic fact that ensure that from our choice of $t_1, \ldots t_d$ before, we may find primes $s_1, \ldots, s_d$ that satisfy the conditions of Theorem 9.

Lemma 10 (Lemma 5.1 [27])**.** Let $x \geq e^{8d}$, then there are at least $\frac{1}{8d} x \log x$ primes s in the interval $(1 - \frac{1}{2d})x < s \leq (1 - \frac{1}{4d})x$.

We note that from our choice of r we have $r \geq (4n/b)^{\frac{1}{d}} \geq n^{\frac{1}{2d}}$ where the last inequality holds for any $d, n \geq 2$. Then using the fact that $\lceil \log_2 n \rceil \geq d^{12}$ and $\log_2 n \geq \lceil \log_2 n \rceil - 1$ we obtain

$$r \geq n^{\frac{1}{2d}} \geq 2^{\log(n^{\frac{1}{2d}})} \geq 2^{\frac{1}{2d}\log n} \geq 2^{\frac{1}{2d}(\lceil \log_2 n \rceil - 1)} \geq 2^{\frac{1}{2}(d^{11}-1)} \geq 2^{d^{10}}$$

where the last line follows from the fact that $d \geq 2$. Thus $r/2 \geq 2^{d^{10}-1} \geq e^{8d}$, and so Lemma 10 tells us that there exist

$$\frac{1}{8d} \cdot \frac{r/2}{\log(r/2)} \geq \frac{1}{16d} \cdot \frac{r}{\log r} \geq \frac{1}{16r} \cdot \frac{2^{d^{10}}}{\log(2^{d^{10}})} = \frac{2^{d^{10}}}{16d^{11}\log 2} \geq d \geq d'$$

Therefore we may find the first d' primes $s_1, \ldots, s_{d'}$ in this interval.

Similarly, we may apply the lemma again but with $x = r$ to find $d \geq d - d'$ primes in the interval $(1 - \frac{2}{d})r < q \leq (1 - \frac{1}{4d})r$. We may find these primes in $\mathcal{O}(n)$ time [27].

Proposition 10 (Proposition 5.2 in [27])**.** We may construct a numerical approximation $\tilde{\mathcal{F}}_{s_1,\ldots,s_d} : \otimes_i \tilde{\mathbb{C}}_\circ^{s_i} \to \otimes_i \tilde{\mathbb{C}}_\circ^{s_i}$ for $\mathcal{F}_{s_1,\ldots,s_d}$ such that $\epsilon(\tilde{F}_{s_1,\ldots,s_d}) < 2^{\gamma+5} T \log_2 T$ and

$$\mathcal{C}(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) < \frac{4T}{r} M(3rp) + \mathcal{O}(n \log n).$$

Proof. First we verify the conditions of Theorem 9. For our choice of $t_1, \ldots, t_d$ and $s_1, \ldots, s_d$ above, we have

$$\frac{t_i}{s_i} - 1 \geq \frac{1}{1 - \frac{1}{4d}} - 1 = \frac{1}{4d - 1} > \frac{1}{4d}$$

and since $d \leq \lceil \log_2 n \rceil^{\frac{1}{d}}$ and $p = 6 \lceil \log_2 n \rceil$ we have $\frac{1}{4d} > p^{-1}$. Thus the conditions of the theorem hold.

Hence there exists a $\gamma < 2dp$ and maps $\mathcal{A} : \otimes_i \mathbb{C}^{s_i} \to \otimes_i \mathbb{C}^{t_i}$ and $\mathcal{B} : \otimes_i \mathbb{C}^{t_i} \to \otimes \mathbb{C}^{s_i}$ such that $\mathcal{F}_{s_1,\ldots,s_d} = 2^\gamma \mathcal{A} \mathcal{F}_{t_1,\ldots,t_d} \mathcal{B}$ and approximations $\tilde{\mathcal{A}} : \otimes_i \tilde{\mathbb{C}}^{s_i} \to \otimes_i \tilde{\mathbb{C}}^{t_i}$ and $\tilde{\mathcal{B}} : \otimes_i \tilde{\mathbb{C}}^{s_i} \to \otimes_i \tilde{\mathbb{C}}^{t_i}$. We then apply Theorem 7 to obtain an approximation $\tilde{\mathcal{F}}_{t_1,\ldots,t_d}$ for $\mathcal{F}_{t_1,\ldots,t_d}$.

Now consider the scaled transform

$$\mathcal{F}'_{s_1,\ldots,s_d} = 2^{-\gamma} \mathcal{F}_{s_1,\ldots,s_d} = \mathcal{A} \mathcal{F}_{t_1,\ldots,t_d} \mathcal{B}.$$

We then may compute an approximation $\tilde{\mathcal{F}}'_{s_1,\ldots,s_d} = \tilde{\mathcal{A}} \tilde{\mathcal{F}}_{t_1,\ldots,t_d} \tilde{\mathcal{B}}$. By Lemma (E.6 Function Composition) we have

$$\varepsilon(\tilde{\mathcal{F}}'_{s_1,\ldots,s_d}) \leq \varepsilon(\tilde{\mathcal{A}}) + \varepsilon(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) + \varepsilon(\tilde{B}) < 2dp^2 + 8T \log_2 T.$$

Since $\|\mathcal{F}_{s_1,\ldots,s_d}\| \leq 1$, we can obtain the approximation by applying the approximate scaling lemma with $c = 2^\gamma$ to obtain the approximation $\tilde{\mathcal{F}}'_{s_1,\ldots,s_d}$. We thus obtain via the approximate multiplication lemma that,

$$\begin{aligned}
\varepsilon(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) &< 2^{\gamma+1} \varepsilon(\tilde{\mathcal{F}}'_{s_1,\ldots,s_d}) + 3 \\
&< 2^{\gamma+2}(2dp^2 + 8T \log_2 T) + 3 \\
&< 2^{\gamma+1}(3dp^2 + 8T \log_2 T).
\end{aligned}$$

Since $d \leq \lceil \log_2 n \rceil^{\frac{1}{12}}$ and $p = 6 \lceil \log_2 n \rceil$ it follows that $3dp^2 < n / \lceil \log_2 n \rceil$, and thus $3dp^2 < n / \lceil \log_2 n \rceil = T < T \log_2 T$.

So we may conclude $\varepsilon(\tilde{\mathcal{F}}'_{s_1,\ldots,s_d}) < 2^{\gamma+5} T \log_2 T$.

The complexity is straightforward to compute

$$\begin{aligned}
\mathcal{C}(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) &= \mathcal{C}(\tilde{\mathcal{A}}) + \mathcal{C}(\tilde{\mathcal{F}}_{t_1,\ldots,t_d}) + \mathcal{C}(\tilde{\mathcal{B}}) + O(Tp^{1+\delta}) \\
&= \frac{4T}{r} M(3rp) + O(dTp^{3/2+\delta} p^{7/24} + Tp \log T + Tp^{1+\delta}).
\end{aligned}$$

Since we have $d \leq \lceil \log_2 n \rceil^{\frac{1}{d}} = (\frac{1}{6}p)^{\frac{1}{d}}$, and $\delta < 1/8$ (this could be guaranteed with the Schönage Strassen integer multiplication algorithm) we have

$$dp^{3/2+\delta}p^{7/23} = O(p^2).$$

Since $p = \mathcal{O}(\log n)$ by its definition, we can further simplify the complexity to

$$C(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) = (\frac{4T}{r}M(3rp) + O(n \log n).$$

□

Proposition 11 (Proposition 5.3 in nlogn)**.** We may construct a numerical approximation $\tilde{\mathcal{M}} : \otimes_i \tilde{\mathbb{C}}_\circ^{s_i} \times \otimes_i \tilde{\mathbb{C}}_\circ^{s_i} \to \otimes_i \tilde{\mathbb{C}}_\circ^{s_i}$ for $\mathcal{M}(u, v) := \frac{1}{S}u * v$ such that $\epsilon(\tilde{M}) < 2^{\gamma+8}T^2 \log_2 T$ and

$$\mathcal{C}(\tilde{M}) < \frac{12T}{r}M(3rp) + \mathcal{O}(n \log n).$$

Proof. Now we will apply the convolution formula to evaluate the convolution here. First use the previous proposition to handle the forward $\tilde{\mathcal{F}}_{s_1,\ldots,s_d}$ and inverse transforms $\tilde{\mathcal{F}}^{-1}_{s_1,\ldots,s_d}$, and one of the approximation lemmas (E.3 Multiplication) to handle the pointwise multiplications. Applying the approximate function composition lemma we obtain

$$\begin{aligned}\varepsilon(\tilde{w}') &\leq \varepsilon(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) + \varepsilon(\tilde{\mathcal{F}}_{s_1,\ldots,s_d}) + \varepsilon(\tilde{\mathcal{F}}^{-1}_{s_1,\ldots,s_d}) + 2 \\ &< 3 \cdot 2^{\gamma+5}T \log_2 T + 2 < \frac{7}{2} \cdot 2^{\gamma+5}T \log_2 T.\end{aligned}$$

Then we need to scale the result by S. Which can be accomplished in time $\mathcal{O}(Sp^{1+\delta}) = \mathcal{O}(n \log n)$ such that

$$\varepsilon(\tilde{w}) < 2S\varepsilon(\tilde{w}') + 3 < 7S \cdot 2^{\gamma+5}T \log_2 T + 3 < 2^{\gamma+8}T^2 \log_2 T.$$

So altogether, the three transforms and the scaling gives us a complexity of $\mathcal{C}(\tilde{M}) < \frac{12T}{r}M(3rp) + \mathcal{O}(n \log n)$. □

All that remains now is to convert the integer multiplication problem into a convolution problem in $\otimes_{i=1}^d \mathbb{C}^{s_i}$ which we showed could be done. We now apply the previous proposition to evaluate the convolution and coerce the result back into $\mathbb{Z}[x]$ by rounding to the nearest integer.

Now we will show that we can recover the coefficients of the result correctly. Let $f, g \in \mathbb{Z}[x]$ and $h = fg \in \mathbb{Z}[x]$ be the polynomials in $\mathbb{Z}[x]$, and let $f', g', h' \in \otimes_i \mathbb{C}^{s_i}$ be the corresponding elements in $\otimes_i \mathbb{C}^{s_i}$ given by the coefficient vectors,

so $h' = f' * g'$. Let $u = 2^{-b} f'$, $v = 2^{-b} g'$ and $w = \mathcal{M}(u, v) = \frac{1}{S} u * v$. Then $h' = 2^{2b} S w$. Since we set our working precision p to be $p = 6b$, we may use the previous proposition to compute an approximation $\tilde{w} = \tilde{\mathcal{M}}(u, v)$ such that $\varepsilon(\tilde{w}) < 2^{\gamma+8} T^2 \log_2 T$ in time $(12T/r)M(3rp) + O(n \log n)$.

Now notice that

$$\|h' - 2^{2b} S\tilde{w}\| = 2^{2b} S \|w - \tilde{w}\| < 2^{2b+\gamma+8-p} T^3 \log_2 T.$$

Since $T < n \leq 2^b$ and $T \log_2 T \leq T \log_2 n \leq Tb < 8n \leq 2^{b+3}$ this yields

$$\|h' - 2^{2b} S\tilde{w}\| < 2^{5b+\gamma+11-p} = 2^{-b+\gamma+11}.$$

However since $\gamma < 8b^{2/3} < b - 13$, because $b \geq 4096$ we have

$$\|h' - 2^{2b} S\tilde{w}\| < \frac{1}{4}$$

thus by rounding the coefficients in h' to the nearest integer, we are guanteed to recover the correct result. Additionally we may perform this operation in $\mathcal{O}(Sp^{1+\delta}) = \mathcal{O}(n \log n)$ by multiply each coefficient of $\tilde{w}$ by $2^{2b} S$ and then truncating the result to p bits.

Thus we obtain the recursive formula

$$\mathcal{C}(n) < \frac{12T}{r} \mathcal{C}(3rp) + \mathcal{O}(n \log n),$$

To find a closed expression for this formula, we expand the first recursive step to show that the cost of the algorithm is geometrically decreasing at each step

$$\begin{aligned} \frac{12T}{r} \mathcal{C}(3rp) < c \frac{12T}{r} \times (3rp) \log(3rp) & \\ &= c36Tp \log(3rp) \\ &< 1728cn \log(3rp) \\ &< 1728cn \log(36b) + 1728cn \frac{1}{d} \log(n). \end{aligned}$$

The first term is negligible for large n, and the second term experiences a geometric size reduction for $d > 1728$.

Thus when fully expanded, $\mathcal{C}(n)$ is bounded above by a decreasing geometric sequence multiplied by $\mathcal{O}(n \log n)$. Since a decreasing geometric sequence is always bounded above by a constant, we have $\mathbb{C} \in \mathcal{O}(n \log n)$.

Chapter 7

Implementation Details

In previous chapters we look at asymptotically fast algorithms for polynomial multiplication in a variety of different coefficient algebras. To conclude this thesis we will introduce a technique for sparse multiplication and an analysis of the performance of some of the algorithms covered so far in our nPoly library[35].

Sparse multiplication techniques are used when the number of non-zero terms in the polynomial proportion to the degree of the polynomial is low. The complexity of the algorithms we have covered so far have all been solely dependent on the degree of the polynomial and so these algorithms will perform poorly for sparse polynomials. Now we will look at a technique for sparse polynomial multiplication, this is also useful as it may be used as a base case for any of the algorithms we have covered here.

For example if we were to use Karatsuba's algorithm on $1 + x^2 + x^{60} + x^{100}$, then the algorithm would need to recurse 8 times to reach a base case of one. However, if we organise our polynomial into a sparse vector and set a base case at $\#f \leq 2$, then we only need to recurse once. Thus by formulating our base case in terms of the number of non-zero term in conduction with this algorithm, we may achieve a nice speedup. This algorithm also has the advantage for working equally over multivariate polynomials as well, which tend to be sparse in practice.

Note that this doesn't change the current complexity analysis from previous chapters as they were formulated more towards dense coefficient vector representations since we are interesting in proving upper bounds for the algorithms.

7.1 Sparse Polynomial multiplication

In Chapter 3 we showed that the schoolbook multiplication method has complexity $\mathcal{O}(nm)$ for input polynomials of degree n and m. However if our polynomials are reasonably sparse, it is natural to ask if we can design and algorithm with a complexity that is only dependent on the number of non-zero terms in the polynomial, rather than the maximum degree.

It would seem reasonable to try to use a variation of the schoolbook method to achieve this, however a problem arises when coordinating the access of elements in memory. Once we have multiplied two terms together, we need to include this result into the collection of terms we have calculated so far. Since we typically want the terms in our sparse polynomial vectors to be in sorted order[1], we will assume that whatever method we use, we must obtain a sorted list of monomials in the end.

There are several methods we might look at using to achieve this:

- a coefficient vector. This requires potentially $\mathcal{O}(n)$ time to shift data to allow the new term to be inserted,
- a binary tree would provide $\log n$ insertion times, but uses expensive memory operations and is not memory efficient,
- a hashmap would provide $\mathcal{O}(1)$ insertion time, but incurs additional costs when hashing the elements. Furthermore, since elements in a hashmap are not store in any particular order, the elements would need to be sorted in the end. Hence the total complexity of this method would be $m \log m$ where m is the number of elements in the hashmap. This is more suitable in the case of multiplying many polynomials together as we then only need to sort the array once at the end.

The last two are valid options and achieve the same complexity, though they incur a high runtime. Our method avoids such costs by recognising a partial ordering on the resulting terms and resolving this partial order into a total order using a heap.

To see this partial order, let $f, g \in K[x_1, \ldots, x_n]$ whose monomials have a lexicographical ordering[2], then we may general our formula for the product of fg

[1] Otherwise this would make other operations such as addition and finding the lead term more difficult to compute

[2] This holds for any monomial ordering however

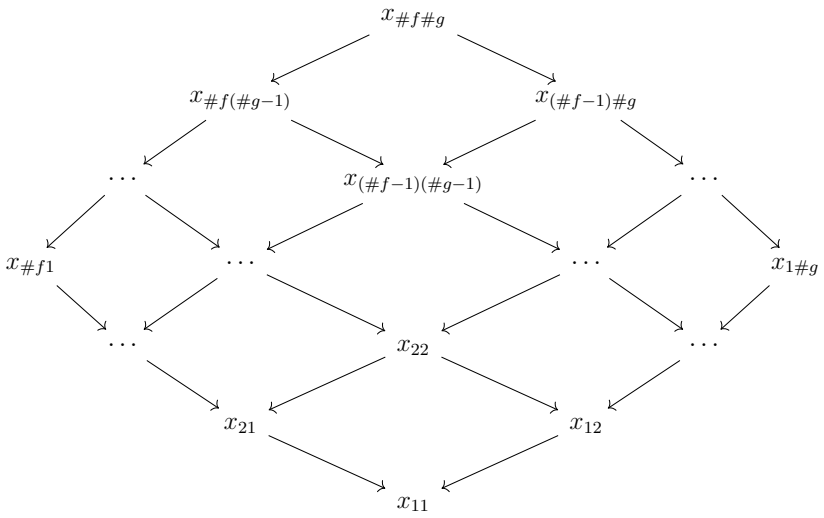

Figure 7.1: The partial order on the monomials of the product fg

as

$$fg = \sum_{\alpha,\beta} \left(\sum_{\alpha+\beta=\gamma} f_\alpha g_\beta \right) x^\gamma.$$

Consider the term with monomial $x^{\alpha+\beta}$. Then for all $\alpha' \leq \alpha$ and $\beta' \leq \beta$ we have $x^{\alpha'+\beta'} \leq x^{\alpha+\beta}$. Thus this forms a partial ordering which can be visualised in Figure 7.1.

A *min-heap* is a data structure which has two operations: insert an element, extract the minimum element in the heap. We will use a min-heap to resolve the partial ordering that we know *a priori* into a total ordering.

Theorem 10. Let $f, g \in K[x_1, \ldots, x_n]$, with f and g stored in a sparse vector representation and sorted lexicographically. Then we may calculate the product fg using $\mathcal{O}(\#f\#g)$ ring operations, and $\mathcal{O}(\#f\#g \log(\min\{\#f, \#g\}))$ monomial additions or comparisons. Thus when $K = \mathbb{Z}$, then the total complexity is $\mathcal{O}(\#f\#g \log(\min\{\#f, \#g\})\mathcal{C}(mon)$ where $\mathcal{C}(mon)$ denotes the complexity of monomial additions and comparisons.

Later research showed that this technique is similar to Johnson's sparse multiplication algorithm [7].

Proof. We will evaluate the graph in Figure 7.1 by traversing the graph from the bottom up, computing each layer as we go. First multiply the smallest two terms of f and g and add it to the result vector. We know that there can not be any other terms smaller than this. In general, whenever $x^{\alpha_{i+1}} x^{\beta_j}$ and $x^{\alpha_i} x^{\beta_{j+1}}$

has been put into the resulting vector, compute $x^{\alpha_{i+1}}x^{\alpha_{j+1}}$ and insert it into the heap. After this, we then extract the minimum element from the heap and insert it into the result.

This continues until there are no more elements in the heap.

Since we calculate each of the resulting monomials once, this requires $\mathcal{O}(\#f\#g)$ ring operations and $\mathcal{O}(\#f\#g)$ monomial additions. Then note that in this partial order, there can only be a maximum of $\min(\#f, \#g)$ elements that can simultaneously not have an order relation on them. Therefore the heap can contain $\min(\#f, \#g)$ elements at any time. Therefore each of the heap operations requires at most $\log(\min(\#f, \#g))$ monomial comparisons.

Therefore, this algorithm requires at most $\mathcal{O}(\#f\#g)$ ring operations, and $\mathcal{O}(\#f\#g \log(\min\{\#f, \#g\}))$ monomial additions or comparisons.

□

As mentioned previously the algorithms in previous chapter are solely dependent on the degree of the polynomials for their run time, and often it is not clear how to generalise them for sparse polynomials. Since the algorithms are recursively formulated, on way of generalising them would be to set the base case to occur based on the number of non-zero terms, rather than a degree. Then in the base case we may apply this algorithm to efficiently evaluate the solution.

7.2 nPoly Multiplication Benchmarks

In the Rust programming language, we have implemented several of the algorithms presented in this thesis as part of the nPoly polynomial arithmetic library. Rust is a modern systems programming language that has become increasingly popular in recent years for its ability to make highly-performant, highly generic code.

One of the advantages of Rust is its support for generic programming which has enabled nPoly to implement its algorithms over all algebras the algorithms support. Currently, nPoly supports: polynomial addition/subtraction/evaluation, Schoolbook multiplication, Karatsuba multiplication, FFT multiplication, Kronecker substitution and limited support for Gröbner bases.

Here we will look at several algorithms, namely: Schoolbook multiplication (with hash-maps), Karatsuba's algorithm, and the standard FFT-based multiplication algorithm. One of the goals of this thesis was to develop a heuristic for selecting the most appropriate algorithm for a given set of input polynomials,

however we note that the times obtained here are not highly optimised implementations as we have implemented all parts of the algorithms (including the FFT) ourselves. To improve the runtimes we may look at using existing software libraries to perform such parts of the code such as FFTW for the Fourier transform. Thus we can only explain the heuristic that nPoly uses for selecting its algorithms. All measurements in this section were obtained with a MacBook Air 2016 1.6 GHz Dual-Core Intel Core i5, with 8 GB of RAM.

Beginning with the schoolbook method for multiplication, we have tested the algorithm by multiplying polynomials together with varying sparsity. We have also tested the runtime cost of increasing the number of indeterminates. From Figure 7.2 we can see that the number of indeterminates has a (roughly)-linear affect on the runtime performance, as we would expect since the only thing that would change is the hashing function used to hash the terms into the hash table. This hashing function is a source for potential optimisation as we have used the default function given by Rust, though since our monomials are vectors is $\mathbb{Z}^d$ (where d is the number of indeterminates), it may be easy find more efficient hashing functions.

We also included a test for an optimised version of the univariate case. This involved a simpler monomial that the monomial data type we used for the other versions and is a bout two as faster when compared to the univariate case where we use the monomial for multi-indices type.

Karatsuba's algorithm performs reasonably well and is able to calculate polynomials with degree 10^5 in around 2 seconds. Notice in Figure 7.3 that the runtime for Karatsuba's algorithm has several plateaus, this is due to the recursive nature of the algorithm that all achieve a similar runtime due to the fact that the algorithm recurses the same number of times, and currently there is a very small base case of $n = 1$. This indicates that the algorithm performs particularly bad for some groups of polynomials that are only slightly greater than a power of two. One could experiment with large bases cases, or the method from the previous section to reduce this effect.

As we can see in Figure 7.4, the FFT algorithm exhibits an $\mathcal{O}(n \log n)$ complexity. Note however that it is measured in terms of the degrees of the input polynomials rather than the number of non-zero terms as the algorithm will behave the same regardless of the sparsity. As with Karatsuba's algorithm, the recursive nature of the FFT creates plateaus in the run time which can be seen in the graph.

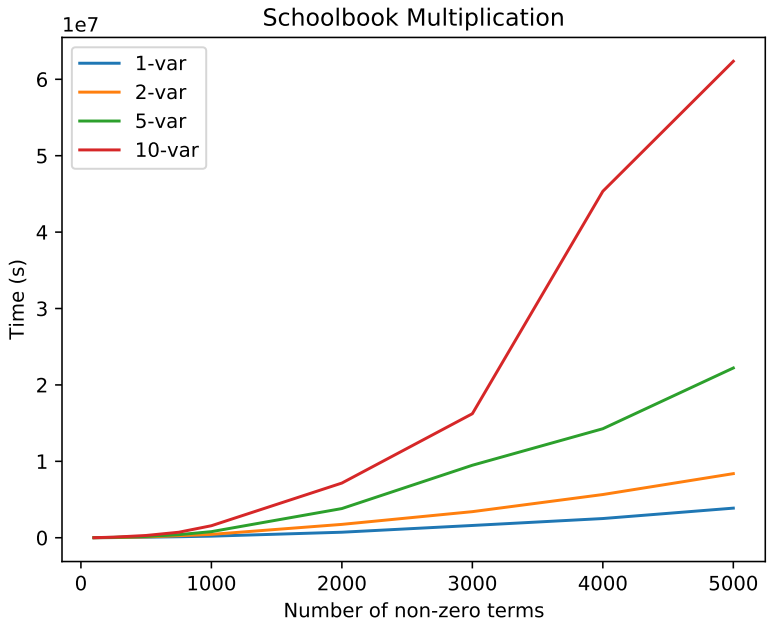

Figure 7.2: Schoolbook Multiplication (hash-maps) on Sparse multivariate polynomials

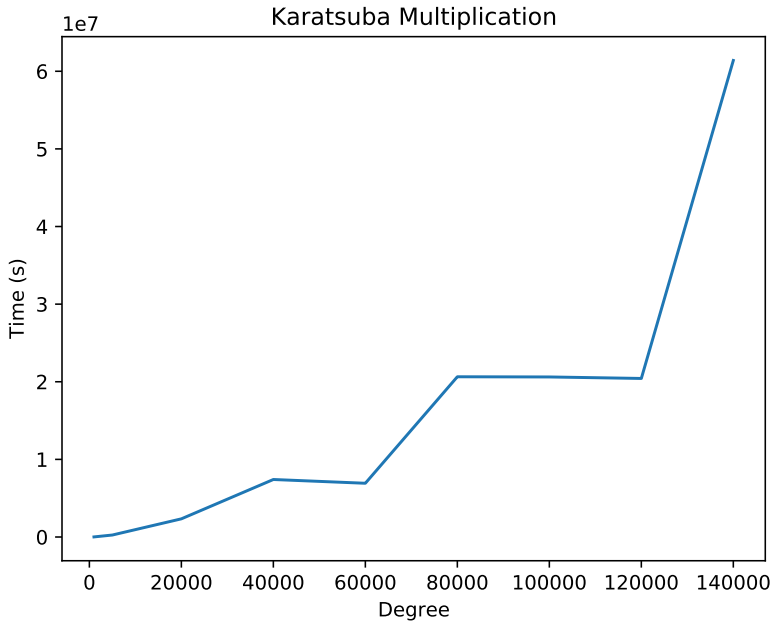

Figure 7.3: Karatsuba's Algorithm for polynomial multiplication

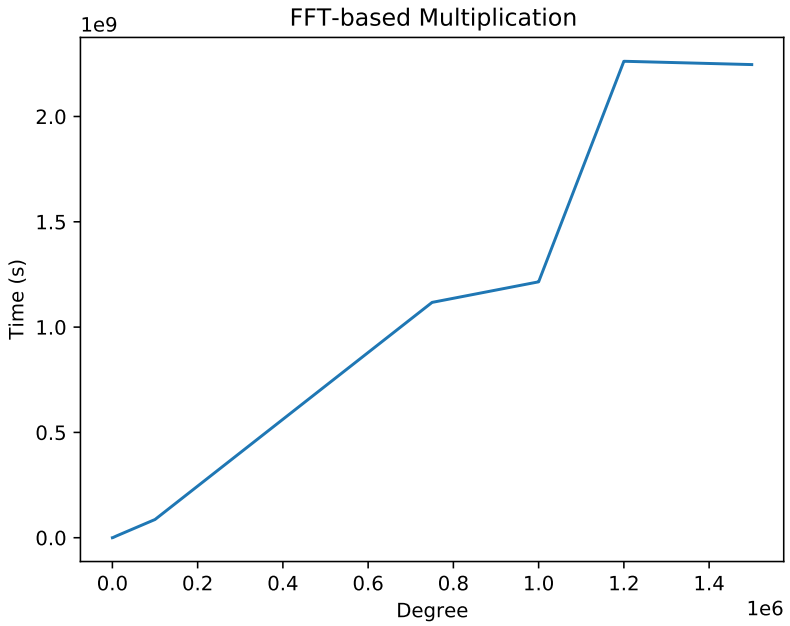

Figure 7.4: FFT-based polynomial multiplication algorithm on dense polynomials

Figure 7.5 show the run time for a naïve implementation of Kronecker substitution for converting multivariate polynomials to their univariate equivalent and back again. Note that this implementation creates new vectors each time and so it needs to dynamically allocate memory which may account for a significant proportion of the cost.

As we can see, Kronecker substitution performs quite poorly and has a similar complexity as the Schoolbook method. Thus we propose the optimisation as suggested in Section 3.4.2 where we simply take a different algebraic meaning to the multi-indices to see them as univariate indices in a mixed-radix representation. This will incur no runtime costs if the polynomials are sorted in the lex order, however, we may need to reorganise the array of terms otherwise. This will be $\mathcal{O}(n \log n)$ in the number of terms and so is insignificant compared to the cost of multiplication.

From our tests we found that the FFT algorithm dominates schoolbook and Karatsuba's algorithm for almost all degree sizes. By inspecting the tabulated results in the Appendix, we can see that the FFT algorithm begins to dominate Karatsuba's when the degree of the polynomial is around 20. This is unexpectedly low, and is likely to be due to unnecessary overhead from function call, as our implementation of Karatsuba's algorithm uses a top-down approach whereas our

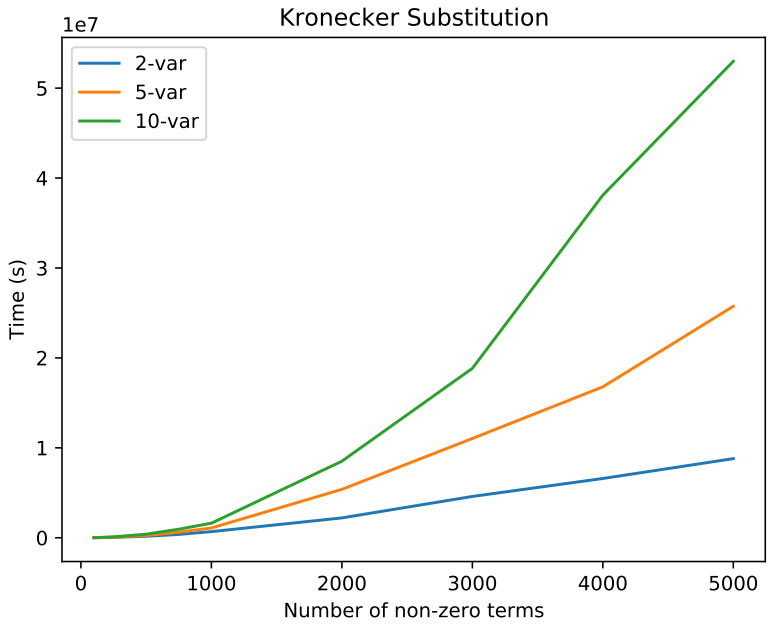

Figure 7.5: Naïve Kronecker substitution on multivariate polynomials

implementations of the FFT uses a bottom-up approach.

The schoolbook algorithm doesn't out-perform Karatsuba's algorithm at any reasonable sizes which is likely due to the expensive hashing operations we uses. Note however that our tests for the schoolbook method were a function of the number of non-zero terms whereas Karatsuba's and the FFT were a function of the degrees of the polynomials. Hence, when selecting the algorithm to use to multiply polynomials, we can compare the cost of the schoolbook method using the number of non-zero terms in the polynomial and the cost of Karatsuba's algorithm (if the degree is less than 20) or the FFT based algorithm as a function of the degree of the polynomials. We then select the algorithm that returns the smallest result.

Appendix A

nPoly Benchmarking

Rows corresponds to benchmarks at a certain degree sizes or number of variables in the polynomial ring. All measurements are in microseconds.

Degree	time(μs)
5	6
10	16
20	69
30	68
50	131
100	353
100	358
500	7412
1000	12516
10000	772279
100000	20829654
200000	64446486

Table A.1: Karatsuba's Algorithm

FFT	time(μs)
5	20
10	59
20	65
50	78
100	92
1000	544
10000	7824
100000	82067
1000000	172875

Table A.2: The FFT algorithm

Schoolbook	1	2	5	10
5	14	17	27	72
10	34	32	77	170
50	652	931	1154	2106
100	2744	3280	4804	9731
250	16879	32055	53999	75831
500	74315	126403	168269	295479
750	117782	268272	431288	730663
1000	184860	428177	780056	1437264
2500	1064871	2650782	5490774	10380380
5000	4551621	8260184	22343481	66544669

Table A.3: Schoolbook method with a hashmap, with different number of indeterminates

Kronecker	2	5	10
100	4153	5552	8891
250	28008	44412	68947
500	108576	177699	276068
750	257015	414361	668346
1000	419838	709191	1117284
2500	2284089	5064354	9442733
5000	6444083	19040990	65630087

Table A.4: Naïve Kronecker substitution, with different number of indeterminates

Bibliography

[1] Anatolii Karatsuba and Yu Ofman. "Multiplication of Multidigit Numbers on Automata". In: *Soviet Physics Doklady* 7 (Dec. 1962), p. 595.

[2] James W. Cooley and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series". In: *Mathematics of Computation* 19.90 (1965), pp. 297–301. ISSN: 00255718, 10886842. URL: http://www.jstor.org/stable/2003354.

[3] L. Bluestein. "A linear filtering approach to the computation of discrete Fourier transform". In: *IEEE Transactions on Audio and Electroacoustics* 18.4 (1970), pp. 451–455. DOI: 10.1109/TAU.1970.1162132.

[4] A. Schönhage and V. Strassen. "Schnelle Multiplikation großer Zahlen". In: (1971). DOI: 10.1007/BF02242355.

[5] Stephen A. Cook and Robert A. Reckhow. "Time-Bounded Random Access Machines". In: *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*. STOC '72. Denver, Colorado, USA: Association for Computing Machinery, 1972, pp. 73–80. ISBN: 9781450374576. DOI: 10.1145/800152.804898. URL: https://doi.org/10.1145/800152.804898.

[6] R. Agarwal and C. Burrus. "Fast Convolution using fermat number transforms with applications to digital filtering". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 22.2 (1974), pp. 87–97.

[7] Stephen C. Johnson. "Sparse polynomial arithmetic". In: *SIGSAM Bull* (1974).

[8] Sean C. Martin. http://etheses.dur.ac.uk/7615/1/7615_4680.PDF?UkUDh:CyT. Accessed: 23-9-2020. 1980.

[9] Don H Heideman Michael T.; Johnson. *Gauss and the History of the Fast Fourier Transform*. 1984. DOI: 10.1109/MASSP.1984.1162257.

[10] Wan-Chi Siu and A.G.Constantinides. "Fast mersenne number transforms for the computation of discrete fourier transforms". In: (1984). DOI: 10.1016/0165-1684(85)90035-0.

[11] Paul Barrett. "Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor". In: *Advances in Cryptology — CRYPTO' 86*. Ed. by Andrew M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 311–323. ISBN: 978-3-540-47721-1.

[12] David G. Cantor and Erich Kaltofen. "On fast multiplication of polynomials over arbitrary algebras". In: *Acta Informatica Vol 28 Issue 7* (1991). DOI: 10.1007/BF01178683.

[13] A. Dutt and V. Rokhlin. "Fast Fourier Transforms for Nonequispaced Data, II". In: *Applied and Computational Harmonic Analysis* 2.1 (1995), pp. 85–100. ISSN: 1063-5203. DOI: https://doi.org/10.1006/acha.1995.1007. URL: http://www.sciencedirect.com/science/article/pii/S106352038571007X.

[14] Anatolii Karatsuba. "The Complexity of Computations". In: *Proceedings of the Steklov Institute of Mathematics* 211 (Jan. 1995), pp. 169–.

[15] Wieb Bosma, John Cannon, and Catherine Playoust. "The Magma algebra system. I. The user language". In: (1997), pp. 235–265.

[16] Peter Bürgisser, Michael Clausen, and M. Amin Shokrollahi. *Algebraic Complexity Theory*. 1997.

[17] Donald Knuth. *The Art of Computer Programming Volume 2: Seminumerical Algorithms*. 1997. ISBN: 0-201-89684-2.

[18] Daniel Bernstein. "Multidigit Multiplication For Mathematicians". In: (Sept. 2001).

[19] Pavel Emeliyanenko. "Efficient Multiplication of Polynomials on Graphics Hardware". In: *Advanced Parallel Processing Technologies*. Ed. by Yong Dou, Ralf Gruber, and Josef M. Joller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 134–149. ISBN: 978-3-642-03644-6.

[20] Pavel Emeliyanenko. "Efficient Multiplication of Polynomials on Graphics Hardware". In: *Advanced Parallel Processing Technologies*. Ed. by Yong Dou, Ralf Gruber, and Josef M. Joller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 134–149. ISBN: 978-3-642-03644-6.

[21] Andrzej Chmielowiec. "Fast, Parallel Algorithm for Multiplying Polynomials with Integer Coefficients". In: *Proceedings of the World Congress on Engineering*. 2012.

[22] Joachim von vur Gathen and Jürgen Gerhard. *Modern Computer Algebra. Third Edition*. 2013. ISBN: 1107039037.

[23] Mounir T. Hamood and Said Boussakta. "Efficient algorithms for computing the new Mersenne number transform". In: *Digital Signal Processing* 25 (2014), pp. 280–288. ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2013.10.018. URL: http://www.sciencedirect.com/science/article/pii/S1051200413002388.

[24] Michael Monagan and Roman Pearce. "POLY: A New Polynomial Data Structure for Maple 17". In: *Computer Mathematics*. Ed. by Ruyong Feng, Wen-shin Lee, and Yosuke Sato. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 325–348. ISBN: 978-3-662-43799-5.

[25] B. Chang et al. "Accelerating Multiple Precision Multiplication in GPU with Kepler Architecture". In: *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2016, pp. 844–851. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0122.

[26] A. Fog. *Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. http://www.agner.org/optimize/. Accessed: 20-10-2020. 2017.

[27] David Harvey and Joris Van Der Hoeven. "Integer multiplication in time O(n log n)". working paper or preprint. Mar. 2019. URL: https://hal.archives-ouvertes.fr/hal-02070778.

[28] David Harvey and Joris Van Der Hoeven. "Polynomial multiplication over finite fields in time O(n log n)". working paper or preprint. Mar. 2019. URL: https://hal.archives-ouvertes.fr/hal-02070816.

[29] *Archived page of Magma website*. https://web.archive.org/web/20060820053803/http://magma.maths.usyd.edu.au/magma/Features/node86.html. Accessed: 20-10-2020. Shows when the Schonage and Strassen integer multiplication algorithm outperforms Karatsuba's in the Magma computer algebra system.

[30] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*.

[31] Steven G. Frigo Matteo; Johnson. "The Design and Implementation of FFTW3". In: (). URL: http://fftw.org/fftw-paper-ieee.pdf.

[32] *GNU Multiple Precision Arithmetic library*. https://gmplib.org/manual/FFT-Multiplication#FFT-Multiplication. Accessed: 20-10-2020.

[33] *Macaulay 2 Source code*. https://github.com/Macaulay2/M2/blob/master/M2/Macaulay2/e/poly.cpp. Accessed: 21-10-2020. Commit 0219847, Line 797 - 806 we see it uses essentially the schoolbook method.

[34] *Maxima Polynomial Documentation*. https://math.tntech.edu/machida/1911/maxima/help/maxima_11.html. Accessed: 20-10-2020. The "fasttimes" function has the same complexity as Karatsuba's algorithm.

[35] *nPoly, A Polynomial library for Rust*. https://github.com/willcsm/nPoly. A polynomials library for the Rust programming language.

[36] *NTL: A library for doing number theory*. http://www.shoup.net/ntl/. Accessed: 20-10-2020. Used to reduce multiplication in finite fields of prime powers, to fields of prime characteristic.

[37] Colin Percival. "Rapid multiplication modulo the sum and difference of highly composite numbers". In: *Centre, Simon Fraser University* (), pp. 387–395.