This little polynomial should keep the computer so busy it doesn't even know we're here.

—Chris Knight (Val Kilmer), *Real Genius* (1985)

Chapter 1

Introduction

This thesis presents new algorithms for computations with polynomials. Such computations form the basis of some of the most important problems in computational mathematics, from factorization to solving nonlinear systems. We seek algorithms that give improvements both in theoretical efficiency as well as practical performance. As the operations we consider are important subroutines in a range of applications, our low-level performance gains will produce improvements in a variety of problems.

A primary focus of this work is the issue of how polynomials are represented in memory. Polynomials have traditionally been stored in the *dense representation* as an array of coefficients, allowing efficient computations with polynomials such as

$$f = x^5 - 2x^4 + 8x^3 + 3x^2 - x + 9.$$

By contrast, the *sparse representation* of a polynomial is a list of nonzero coefficient-exponent tuples, a much more efficient representation when most of the coefficients are zero. This allows compact storage of a much larger set of polynomials, for instance

$$f = x^{5000} - 3x^{4483} - 10x^{2853} + 4x^{21}.$$

The types of polynomial computations we will examine fall into three general categories: basic arithmetic operations, algebraic problems, and inverse symbolic computations.

Basic arithmetic includes operations such as addition, subtraction, multiplication, and division. Optimal, linear-time algorithms for addition and subtraction of polynomials (in any representation) are easily derived. Division, modular reduction, and a great many other operations on densely-represented polynomials have been reduced to the cost of multiplication (sometimes with extra logarithmic factors). In fact, almost any nontrivial computation with polynomials in any representation uses multiplication as a subroutine. Hence multiplication emerges as the most crucial low-level arithmetic operation to consider.

At a slightly higher level, understanding the basic algebraic structure of polynomials is important for many applications in symbolic computation as well as cryptography. The fast algorithms that have been developed for polynomial factorization have been identified as some of the greatest successes of computer algebra. However, these algorithms are only efficient when the polynomials under consideration are represented densely. Many polynomials that arise in practice can only be feasibly stored using the sparse representation, but this more compact representation demands more sophisticated algorithms. Understanding the effects of sparsity on computational efficiency is fascinating theoretical work with important practical consequences.

The arithmetic and algebraic computations above involve manipulating and computing with symbolic representations of mathematical objects. Said mathematical objects are often not known explicitly, but rather given implicitly by a function or program that can be evaluated at any chosen point. Inverse symbolic problems involve computing a symbolic formula for a sampled function. Sparse polynomial interpolation in particular has a rich history as well as important applications in factorization and nonlinear system solving.

Kaltofen (2010) has recently proposed a taxonomy of the most significant mid-level and high-performance computational tasks for exact mathematical computing, called the "seven dwarfs" of symbolic computation. In terms of this classification, the current work falls under the categories of the second and third dwarfs, exact polynomial algebra and inverse symbolic problems. We will also briefly touch on the fifth dwarf, hybrid symbolic-numeric computation.

1.1 Overview

The remaining sections of this chapter introduce the basic concepts and definitions that will be used for the remainder of the thesis. Our computational model has a carefully defined memory layout and counts machine word operations as well as arithmetic operations over an arbitrary algebraic domain. We show the implications of this model for the important basic operation of integer multiplication.

Chapter 2 follows by discussing the architecture and design of a high-performance C++ library containing all the algorithm implementations that we will present. This library is used to benchmark our algorithms against previous approaches and show how the theoretical improvements correspond to concrete practical gains.

The first algorithmic problem we turn to is also the most fundamental: multiplication of dense univariate polynomials. Chapter 3 examines the problem of computing the so-called Truncated Fourier Transform (TFT), which among other applications is used as a subroutine in fast polynomial multiplication. While the normal radix-2 Fast Fourier Transform (FFT) can be computed completely in-place, i.e., overwriting the input with the output, the TFT as originally developed by van der Hoeven (2004) requires linear extra space in the general case. Our new algorithm for an in-place TFT overcomes this shortcoming without sacrificing time cost.

The in-place TFT algorithm is used as a subroutine in one of two new algorithms for dense polynomial multiplication presented in Chapter 4. These new algorithms have the same time

complexity as the two most commonly used sub-quadratic “fast” algorithms for multiplication, but improve on the linear amount of extra space previously required. Our first algorithm works over any ring, matches the $O(n^{1.59})$ time complexity of Karatsuba’s algorithm, and only uses $O(\log n)$ extra space. The second algorithm works in any ring that admits radix-2 FFTs and matches the $O(n \log n)$ time cost of usual FFT-based multiplication over such rings, but uses only $O(1)$ extra storage space. Under our model of space cost, these are the first algorithms to achieve sub-quadratic time $\times$ space complexity for multiplication.

Next we turn to a broader view of polynomial multiplication, encompassing both dense and sparse polynomial multiplication. Two new approaches to this problem are presented in Chapter 5 which effectively provide a gradient between existing sparse and dense methods. Specifically, the algorithms adapt to the case of dense “chunks” of nonzero coefficients in an otherwise sparse polynomial, and to the opposite case of sparse polynomials in which the nonzero terms are mostly evenly spaced. These algorithms provide advantages over previous techniques by being more adaptive to the structure of the input polynomials, giving significant improvement in many cases while never being more than a constant factor more costly than any existing approach. We also show how to combine the two approaches into one algorithm that simultaneously achieves both benefits.

Chapter 6 examines an important algebraic problem for sparse polynomials. Given a sparse polynomial as input, we investigate how to determine whether it is a perfect power of another (unknown) polynomial, and if so, to compute this unknown polynomial root. Perfect powers are a special case of both polynomial factorization and decomposition, and fast algorithms for the problem are known when the input is given in the dense representation. We give the first polynomial-time algorithms for detecting *sparse* polynomial perfect powers, which are randomized of the Monte Carlo type, and work when the coefficients are rational numbers or elements of a finite field with sufficiently large characteristic. We then turn to the problem of actually computing the unknown polynomial root, and give two approaches to that problem. The first is output-sensitive and provably polynomial-time, while the second is much more efficient in practice but relies on a conjecture regarding the sparsity of intermediate results.

The third broad topic of this thesis is sparse polynomial interpolation. Chapter 7 gives a brief overview of existing algorithms for sparse interpolation over various domains. We show how to improve the polynomial-time complexity of the recent algorithm of Garg and Schost (2009) over sufficiently large finite fields. We also present a related algorithm for approximate sparse interpolation, where the coefficients are approximations to complex numbers, and show that our algorithm improves the numerical stability over existing approaches.

Building on these sparse interpolation algorithms, in Chapter 8 we extend their domain by presenting the first polynomial-time algorithm for the problem of sparsest-shift interpolation. This problem seeks a representation of the unknown sampled polynomial in the shifted power basis $1, (x-\alpha), (x-\alpha)^2, \ldots$, with α chosen to minimize the size of the representation, rather than the usual power basis $1, x, x^2, \ldots$. By first computing this sparsest shift α and then performing interpolation in that basis, we greatly expand the class of polynomials that can be efficiently interpolated.

1.2 Polynomials and representations

The issue of polynomial representations is of central importance to much of this thesis. We therefore pause to consider what a polynomial is and how one might be stored in a digital computer.

First, we define some standard notation that we will use throughout. Big-O notation, which has become standard in algorithmic analysis, is a convenient way to compare the asymptotic behavior of functions. Unfortunately, the notation itself can be somewhat misleading in certain situations, so we give a formal definition.

Given two k-variate functions $f, g : \mathbb{N}^k \to \mathbb{R}_{\geq 0}$, we say that $f(n_1,\ldots,n_k) \in O(g(n_1,\ldots,n_k))$ if there exist constants N and c such that, whenever $n_i \geq N$ for every $1 \leq i \leq k$, we have that $f(n_1,\ldots,n_k) \leq c\,g(n_1,\ldots,n_k)$. This is really defining a partial order on functions, but we can also think of $O(g(n_1,\ldots,n_k))$ as defining a set of all k-variate functions f which satisfy the above conditions, and our notation follows the latter convention.

In most cases, the k variables $n_1,\ldots,n_k$ will appear explicitly in $g(n_1,\ldots,n_k)$, but when they do not, or when other symbols appear that are *not* variables, these distinctions must unfortunately be inferred from the context. An important example is $O(1)$, which in the current context represents all k-variate functions bounded above by a constant.

The notations Ω, Θ, o, and ω can all be derived from the definition of O in the standard ways, and we will not list them here. Another common notation which we will use occasionally is *soft-O notation*: $f(n_1,\ldots,n_k) \in \tilde{O}(g(n_1,\ldots,n_k))$ if and only if

$$f(n_1,\ldots,n_k) \in O\left(g(n_1,\ldots,n_k) \cdot (\log g(n_1,\ldots,n_k))^c\right),$$

for some positive constant c. Informally, soft-O notation means ignoring logarithmic factors of polynomial terms. Note for example that the univariate function $\log n \cdot \log\log n$ is in $\tilde{O}(\log n)$ but *not* $\tilde{O}(1)$.

1.2.1 Functional representation

Consider a ring R (commutative, with identity) with binary operations $\times, +, -$, and a positive integer n. If only these three operations are allowed, say on a computer, then the set $\mathsf{R}[x_1, x_2, \ldots, x_n]$ of n-variate polynomials is exactly the set of all computable functions on n inputs. For instance, in a very simple view of modern digital computers operating directly (and exclusively) on bits, every possible computation corresponds to a multivariate polynomial over $\mathbb{F}_2$, the finite field with two elements.

With this fundamental connection between polynomials and computation in mind, we begin with a functional definition of the set $\mathsf{R}[x_1, x_2, \ldots, x_n]$ of n-variate polynomials, and will get to the more common algebraic definition later. Functionally, an n-variate polynomial f is simply a finite sequence of ring operations on a set of n indeterminates $x_1, x_2, \ldots, x_n$. This function can be represented by a so-called straight line program or, somewhat more expressively, by a division-free algebraic circuit.

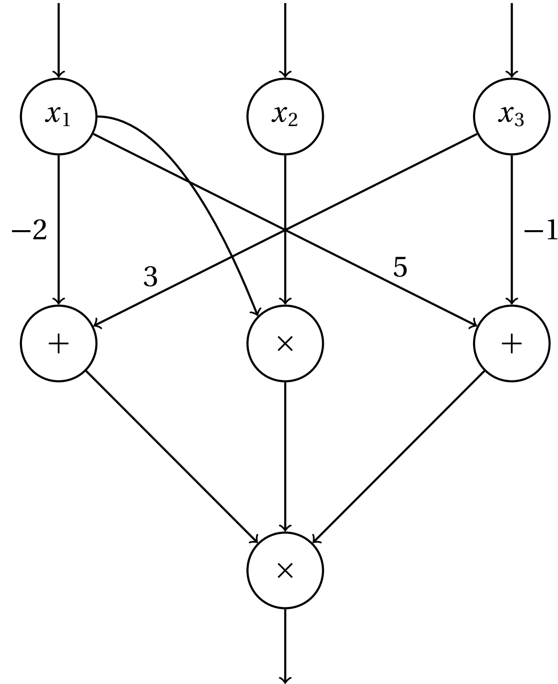

Figure 1.1: Algebraic circuit for $(-2x_1+3x_3)\cdot(x_1\cdot x_2)\cdot(5x_1-x_3)$

For our purposes, such a circuit will be a directed acyclic graph with n source nodes labeled x_1 through x_n and a single sink node for the output. Each node is labeled with $+$ or $\times$. Incoming edges to each $+$ node are also labeled with constant elements from R, and the value computed at that node is the R-linear combination of the values at its direct predecessors, times the values on the incoming edges. The value at each $\times$ node is simply the product of the values of its direct predecessor nodes. For instance, the algebraic circuit in Figure 1.1 represents the function

$$f(x_1,x_2,x_3)=(-2x_1+3x_3)\cdot(x_1\cdot x_2)\cdot(5x_1-x_3).$$

Computations on polynomials represented by algebraic circuits or straight-line programs have been studied extensively (Freeman, Imirzian, Kaltofen, and Yagati, 1988; Kaltofen, 1989; Kaltofen and Trager, 1990; Encarnación, 1997), and the main advantages of this representation are its potential for conciseness and connection to evaluation. Unfortunately, with the important exception of computing derivatives (Baur and Strassen, 1983), very few operations can be efficiently computed in polynomial-time in this representation. Even the most basic operation, determining whether two circuits represent the same polynomial, is notoriously difficult to solve, at least without randomization (Saxena, 2009). Because of these difficulties, many algorithms parameterize their cost based on some algebraic property of the polynomial(s) computed by the circuit. For instance, the best-known factorization algorithm by Kaltofen requires polynomial time in the degree, which could be exponentially larger than the circuit size.

1.2.2 Algebraic representations

A more typical way to represent polynomials is algebraically, as an R-linear combination of monomials in some chosen basis. The most common choice is the standard power basis, wherein each monomial is a product of the indeterminates, each raised to a nonnegative integer power. (Chapter 8 will consider an important alternate basis, namely the sparsest-shifted power basis.)

We now pause briefly to define some useful terminology. Each product of a monomial and a coefficient is called a *term*. We will typically focus on *nonzero terms*, i.e., terms with a nonzero coefficient, and we will implicitly assume that each monomial appears at most once. The highest exponent of a given indeterminate x_i in any nonzero term is called the *partial degree* in x_i. The greatest partial degree is called the *max degree*. The greatest sum of exponents in any nonzero term is called the *total degree*. The number of (distinct) nonzero terms is called the *sparsity* of the polynomial, and the monomials with nonzero coefficients make up its *support*.

For instance, the polynomial represented by the circuit in Figure 1.1 can be written algebraically, in the standard power basis, as $f = -3x_1x_2x_3^2 + 17x_1^2x_2x_3 - 10x_1^3x_2$. It has three nonzero terms, max degree 3, and total degree 4.

The question remains how to actually represent such expressions in a computer. We will briefly discuss the three most common representations, and refer to (Stoutemyer, 1984; Fateman, 2002) for a more in-depth comparison. The most straightforward choice is a multidimensional $d_1 \times d_2 \times \cdots \times d_n$ array of coefficients, where each d_i is greater than the partial degree in x_i. This we term the *dense* representation; its size is bounded by $O(d_1d_2\cdots d_n)$ ring elements, or more simply as $O(d^n)$, where d is the max degree. The dense representation typically admits very fast algorithms, but can be inefficient when the number of nonzero terms in the polynomial is very small compared to d^n.

More compact storage is afforded by the so-called *recursive dense* representation. The zero polynomial is represented as a null pointer; otherwise, terms are collected by the last variable and the representation is by a single length-d_n array of recursive dense polynomials in $n-1$ variables $x_1, x_2, \ldots, x_{n-1}$. This representation has proven to be practically useful in a number of situations, as it admits the use of dense univariate algorithms while being somewhat sensitive to sparsity. A major drawback is its high sensitivity to the variable ordering. Despite this, it is easy to see the size in the worst case is at most $O(dnt)$ ring elements, where t is the sparsity, i.e., number of nonzero terms. Since t could be at most $(d+1)^n$, this would seem to be potentially worse than the dense representation in cases where almost every term is nonzero, but this is not actually true; a better bound on the number of ring elements required for the recursive dense representation of $O(\min(d^n, dnt))$ is also easy to derive.

The most compact algebraic representation is the *sparse* representation (also called the *distributed sparse* representation). In this representation, a polynomial is stored as a list of coefficient-exponent tuples, wherein nonzero terms never need to be explicitly stored. This also corresponds exactly to the way we naturally write polynomials when doing mathematics, and for this reason it has become the standard and default representation in general-purpose computer algebra systems such as Maple, Mathematica, and Sage. This representation always

uses exactly t ring elements for storage. However, in this case we also must take into account storage space for the potentially large exponents. This exponent storage is bounded above by $O(nt \log d)$ bits.

Observe that there is potentially an exponential-size gap between the size of each of these representations. For instance, assuming elements 0 and 1 in R are stored with a constant number of bits, the size of the polynomial $f = x_1^d x_2^d \cdots x_n^d$ in each of the dense, recursive dense, and sparse representations is $\Theta((d+1)^n)$, $\Theta((d+1)n)$, and $\Theta(n \log d)$ bits, respectively.

1.2.3 Algorithms

Most published algorithms for polynomial computation do not explicitly discuss what representation they have in mind, but nonetheless we can roughly categorize them based on those representations mentioned above.

Generally, algorithms in the literature that make no mention of sparsity and whose cost does not explicitly depend on t we will term *dense algorithms*. These algorithms are said to be polynomial-time if their cost is $(n^d)^{O(1)}$, i.e., polynomial in the size of the dense representation. Many important classical results in computer algebra fall into this class, including fast multiplication algorithms (Karatsuba and Ofman, 1963; Schönhage and Strassen, 1971; Cantor and Kaltofen, 1991) and polynomial factorization (Berlekamp, 1967; Cantor and Zassenhaus, 1981).

Some algorithms for so-called sparse polynomial computation have complexity $(ndt)^{O(1)}$. Observe that this is polynomial-time in the size of the recursive dense representation. Richard Zippel's sparse interpolation algorithms (1979; 1990) are good examples of algorithms with this sort of cost. Algorithms for straight-line programs whose cost depends partially on the degree (e.g., Kaltofen and Trager, 1990) could also be considered of this type, as one can easily construct a small circuit for evaluating a sparse polynomial.

The most difficult algorithms to derive are those whose cost is polynomial in the size of the sparse representation defined above. These go by many names in the literature: "sparse" (e.g., Johnson, 1974), "lacunary" (e.g., Lenstra, 1999), and "supersparse" (e.g., Kaltofen and Koiran, 2005). To avoid any ambiguity, we will call any algorithm whose cost is polynomial in the size of the sparse representation — that is, polynomial in n, t, and $\log d$ — a *lacunary algorithm*. Observe that efficient lacunary algorithms automatically give efficient algorithms for all three algebraic representations mentioned above; the blow-up in complexity from a lacunary algorithm to a dense one is at most $O(n \log d)$. Lacunary algorithms also have advantages related to the univariate case, as we will discuss shortly. However, despite the ubiquity of the sparse representation in computer algebra systems, the development of lacunary algorithms has been much less rapid than dense algorithms; see Davenport and Carette (2009) for an overview of some of the specific algorithmic challenges that accompany the sparse representation.

Another class of algorithms we will occasionally consider are those whose cost depends even more strongly on one of the parameters, such as the sparsity t or the size of coefficients. These algorithms are polynomial-time in special cases, for instance when the number

of terms or size of coefficients is constant, as in the recent examples by Filaseta, Granville, and Schinzel (2008) and Pébay, Rojas, and Thompson (2010) on computing gcds and extrema, respectively. However, observe that these are exponential-time algorithms no matter the choice of representation.

1.2.4 Univariate polynomials

Univariate polynomials are an important special case for consideration. Algorithms are of course easier to develop, describe, and implement in this case. But univariate polynomials also arise frequently in practice and have a strong relationship with multiple-precision integers. For instance, algorithms for fast multiplication of dense univariate polynomials have typically followed those for multiple-precision integers with the same complexity.

Observe that in the univariate case the recursive dense representation degenerates to the dense representation and hence becomes useless. So for univariate problems we need only consider two types of algebraic representations.

Fast algorithms for univariate polynomial computation can be applied to multivariate polynomials via the well-known Kronecker substitution:

Fact 1.1. *(Kronecker, 1882) Let* R *be a ring and* $n, d_1, d_2, \ldots, d_n \in \mathbb{N}$, *For any polynomial* $f \in \mathsf{R}[x]$ *with degree less than* $d_1 d_2 \cdots d_n$, *there exists a unique polynomial* $\hat{f} \in \mathsf{R}[x_1, x_2, \ldots, x_n]$, *with partial degrees less than* $d_1, d_2, \ldots, d_n$ *respectively, such that*

$$f(x) = \hat{f}(x, x^{d_1}, x^{d_1 d_2}, \ldots, x^{d_1 d_2 \cdots d_{n-1}}).$$

For many problems with multivariate polynomials, making this substitution (evaluating at high powers of a single variable) often allows the use of univariate algorithms. For instance, given the bivariate polynomial

$$\hat{f}(x, y) = 5 + 6x^2 y - 3xy^{10},$$

we could compute $\hat{f}^2$ by using the Kronecker substitution with bounds on the degrees in $\hat{f}^2$:

$$f(x) = \hat{f}(x, x^5) = 5 + 6x^7 - 3x^{51}.$$

We see that

$$f^2 = 25 + 60x^7 + 36x^{14} - 30x^{51} - 36x^{58} + 9x^{102},$$

and because we know that $\deg_x \hat{f}^2 < 5$, we can then apply Fact 1.1 to write down the coefficients of $\hat{f}^2$, by taking a quotient with remainder when each exponent in f^2 is divided by 5:

$$\hat{f}^2 = 25 + 60x^2 y + 36x^4 y^2 - 30xy^{10} - 36x^3 y^{11} + 9x^2 y^{20}.$$

More generally, the Kronecker substitution corresponds to writing each exponent of the univariate polynomial f in the mixed-radix representation with basis $(d_1, d_2, \ldots, d_n)$, and using the fact that every nonnegative integer less than the product $d_1 d_2 \cdots d_n$ has a unique representation as an n-tuple in the mixed-radix representation to that basis (see Knuth, 1981, §4.1).

A similar approach is used to convert a polynomial with coefficients in a finite field to an integer. For a polynomial with coefficients in $\mathbb{Z}/m\mathbb{Z}$, for example, we can write each coefficient as its unique integer residue in the range $0, 1, \ldots, m-1$, then convert this to an integer (in the binary representation) by evaluating the integer polynomial at a power of 2. If the power of 2 is large enough, then this is an invertible map, as used for example by Harvey (2009) for the problem of multiplication.

This conversion between integers and univariate polynomials over finite fields can also be used in the other direction, as we will see in Section 1.4. However, observe that not every problem works as nicely as multiplication. For instance, to factor a bivariate polynomial $\hat{f} \in \mathsf{R}[x, y]$, we could again use the Kronecker substitution to write $f = \hat{f}(x, x^d)$ for $d > \deg_x \hat{f}$, and then factor the univariate polynomial $f \in \mathsf{R}[x]$. However, even though each factor of $\hat{f}$ will correspond to a factor of f, observe that the converse is not true, so that recovering the bivariate factors from the univariate ones is a non-trivial process. Reducing integer factorization to polynomial factorization is even more problematic because of carries.

The Kronecker map is especially useful when polynomials are stored in the sparse representation. If $f \in \mathsf{R}[x]$ and $\hat{f} \in \mathsf{R}[x_1, x_2, \ldots, x_n]$ corresponding to the Kronecker substitution as above, first notice that f and $\hat{f}$ will have the same number of nonzero terms, and in fact the same coefficients. Furthermore, if written in the sparse representation, these two polynomials have essentially the same size. For simplicity, assume the partial degrees are all less than d. Then each exponent in f is a single integer less than d^n, and each exponent in $\hat{f}$ is a vector of integers each less than d. Writing these in binary, their bit-length in both cases is $n \log_2 d$, and so the representation size in the sparse model is identical, at least asymptotically.

This relationship will be useful for our purposes, as it means that algorithms for univariate polynomials will remain polynomial-time under the Kronecker map. Roughly speaking, the exponential increase in the degree only corresponds to a polynomial increase in the logarithm of the degree, which corresponds to the size of the sparse representation. Hence algorithms for univariate polynomials with very few terms and very high degree have an important application. We will consider exactly this type of algorithm at various points throughout this thesis.

1.3 Computational model

It is always important to carefully define the computational framework that will be used to describe and analyse algorithms. In our case, this is more than an academic exercise, as the choice of model will be crucially important for understanding our results in the first few chapters.

1.3.1 Background

Countless descriptions of different abstract machines have been proposed over the last seventy years as general models of mathematical computation. These models have a wide variety of motivations and goals; some are defined according to the functions they can compute,

some according to the structure of the computation, and some according to properties of a class of actual digital computers. In the present work, we will prefer this final category for its (hopeful) connection between theoretical results and practical performance. However, we will discuss some other models as well for comparison.

Turing (1937) developed the model of the first abstract computing machine, the so-called *Turing machine*, which we still use today as the basis for computability and basic complexity classes such as **P** and **NP**. The greatest strength of this model is its simplicity and computational equivalence to other fundamental models such as Church's lambda calculus and pushdown automata.

The slight refinement of *multi-tape Turing machines* are of particular interest here. The simplest such machine has three tapes, a read-only input tape, a write-only output tape, and a single working tape. More sophisticated models such as the seven-tape Turing machine of Schönhage, Grotefeld, and Vetter (1994) have shown some evidence of practicality for mathematical computations, despite bearing little resemblance to actual machines.

However, Turing machines are known to over-estimate the cost of certain operations on modern digital computers, the difference owing primarily to the way memory is accessed. In Turing machine models, each tape has a single head for reading and/or writing which is only allowed to move one position at each step of the computation. By contrast, the structure of internal memory in any modern computer allows random access to any location with essentially the same cost.

This motivated the definition of the *random access machine* (or RAM) by Cook and Reckhow (1973). Their model looks similar to a 3-tape Turing machine, except that the work memory is not a sequential-access tape but rather an array that allows random access and indirect addressing. Another important difference is that every memory location in input, output, or working space can hold an integer of unbounded size, as opposed to the finite alphabet restriction in Turing machines. In addition to the usual operations such as LOAD, STORE, and BRANCH, the instruction set of a RAM also contains operations for basic arithmetic such as addition and multiplication of integers.

While the RAM model has seen much success and seems to be very commonly used in algorithm design and analysis, it still has significant differences from actual digital computers. The most obvious is the RAM's ability to manipulate and store unbounded-length integers with unit cost. This is somewhat mitigated by the *log-cost RAM*, in which the cost of an operation on integers with values less than n is charged according to their bit-length: $O(\log n)$ for all operations except MUL and QUO, which cost $O(\log^2 n)$. A slightly more refined approach is presented by Bach and Shallit (1996, §3.6), where an interesting storage scheme is also mentioned which would allow for sub-linear time algorithms. However, as pointed out for instance by Grandjean and Robson (1991), these models still underestimate the cost of certain kinds of memory accesses and (at least for the log-cost RAM) define the cost of arithmetic too bluntly.

Motivated by these shortcomings is the primary model we will rely on in this thesis, the *Random Access Computer* (or RAC) of Angluin and Valiant (1979). This model bears some similarity to pointer machines (Kolmogorov and Uspenskiĭ, 1958; Schönhage, 1980; Schönhage et al., 1994) in that each memory location may hold a pointer to another memory loca-

tion. Reconciling this with the RAM model, each "pointer" is actually an integer of bounded size, and this "word size" is dependent upon the size of the input. The model supports modular arithmetic on word-size integers as well as indirect addressing. This seems to be most closely aligned to machine instructions in modern (sequential) computing architectures, and the model we propose in the next subsection is essentially an elaboration on the RAC.

Having described the evolution and motivation of our model of choice, we briefly mention some other computational models that will be of interest. First, on the practical side, a shortcoming still of the RAC is that every random memory access is assumed to have the same cost. On any modern computer, the multi-level memory hierarchy of registers, cache, main memory, and disk grossly violates this assumption. Especially in algorithms with close to linear complexity, the dominant cost often becomes that of memory access (these are called *memory-bounded* operations). Models which seeks to address this are the I/O model or "DAM model" of Aggarwal and Vitter (1988) and its cache-oblivious extension by Frigo, Leiserson, Prokop, and Ramachandran (1999).

Backing away from practicality, most of the models mentioned so far are notoriously resistant to any lower bounds for interesting problems. Simpler models such as circuits (equivalently, straight-line programs), described in the previous section, are more useful for these purposes, while still being sufficiently general to describe many algorithms in practice. A special type of circuit of note here is the *bounded coefficients* model, in which every constant appearing on an edge in the circuit must be of constant size (Chazelle, 1998).

The most significant shortcoming of circuits is that they do not allow branching; this is overcome by *branching programs*, an extension of binary decision trees to general domains (Borodin and Cook, 1980). In such models, each node represents a "state", of the computation, which proceeds according to the values computed. The size complexity in such models is defined as the number of bits required to store this state, which is the logarithm of the number of nodes.

1.3.2 In-Memory Machine

As mentioned before, the model we will use in this thesis is the Random Access Computer (RAC) of Angluin and Valiant (1979). The primary motivation for this model was to get a closer connection between practical performance and theoretical complexity for *low-level* computations. While high-level computations or entire programs may often read a large input from a file (similar to the read-once input tape in a RAM), low-level computations are often subroutines within a program, and therefore their input, as well as space for the output, presumably resides in main (random-access) memory at the beginning of the computation.

In the RAC model, this in-memory property is handled by making a special additional instruction to those of a standard RAM that loads the entire input into working memory in a single step. Hence the memory structure of a RAC looks identical to that of a RAM, with the primary difference being that the integers stored in each memory location have bit-length — that is, the size of machine words — bounded (in some way) by the size of the input.

The model we define here gives a more detailed memory layout that will allow a richer discussion of space complexity in the algorithms we develop. Our variation will also give

greater ease in composing algorithms, i.e., calling one algorithm as a subroutine of another. It is computationally equivalent to the RAC, but for clarity, we will use the distinct name of *In-Memory Machine*, or IMM for short.

Storage in the IMM is divided into four parts: constants (C), input (I), temporary working space (T), and output (O). Each of these is represented by a sequential array, which we will refer to respectively as M_C, M_I, M_T, and M_O. All four arrays allow random read access at unit cost, and the work space and output also allow for random write access at unit cost.

Instruction	Description
COPY(A, i, B, j)	Copy the value of $M_A[M_T[i]]$ to $M_B[M_T[j]]$.
	A must be one of C, I, T, or O.
	B must be one of T or O.
BRANCH(i, L)	Go to instruction labeled L iff $M_T[i] = 0$.
ADD(i, j, k)	$M_T[i] \leftarrow M_T[j] + M_T[k] \mod 2^w$
SUB(i, j, k)	$M_T[i] \leftarrow M_T[j] - M_T[k] \mod 2^w$
MUL(i, j, k)	$M_T[i] \leftarrow M_T[j] \cdot M_T[k] \mod 2^w$
QUO(i, j, k)	$M_T[i] \leftarrow \left\lfloor \frac{M_T[j]}{M_T[k]} \right\rfloor$
HALT	Stop execution

Table 1.1: Instruction set for IMM model

The instruction set for the IMM is given in Table 1.1. The parameter w gives the word size, so that $2^w - 1$ is the largest integer that can be stored in a single word of memory. Observe that only the COPY instruction involves indirect addressing. We must also specify that $M_T[i] = 0$ initially for any $i \geq 0$. For any instruction COPY(A, i, B, j) with $B = O$, i.e., a copy to the output space, we also require that $M_T[j]$ is less than the size of the output. More precisely, using the notation of the following paragraph, $0 \leq M_T[j] < \max(\#O, \#O')$.

A *problem* for IMM computation is described by a set $\mathscr{S} \subseteq (\mathbb{Z}^*)^3$ of tuples (I, O, O') indicating the initial configurations of the input and output space, and the corresponding final configuration of the output space. Most commonly, I will simply contain the input and O will be empty, but the model allows for the output space to be wholly or partly initialized, for reasons we will see. As O and O' will be stored in the same part of memory, the size of the output space is defined to be $\max(\#O, \#O')$, and this also bounds the size of this part of memory at any intermediate step in the computation, as detailed above. For the problem to be well-defined, we of course require that any initial input and output configuration appears at most once in $\mathscr{S}$. That is, for any $I, O, O'_1, O'_2 \in \mathbb{Z}^*$, if $(I, O, O'_1) \in \mathscr{S}$ and $(I, O, O'_2) \in \mathscr{S}$, then $O'_1 = O'_2$.

The *size* n of a given instance $(I, O, O') \in \mathscr{S}$ is defined as the size of the input and output, $n = \#I + \max(\#O, \#O')$. Every integer stored in memory must fit in a single word, but as in the RAC model, this word size varies according to the size of the instance, as we will discuss below.

An *algorithm* for an IMM problem consists of three components: a labeled list of instructions from the set in Table 1.1, a function $\mathscr{W} : \mathbb{N} \to \mathbb{N}$ to define the word size, and a function $\mathscr{C} : \mathbb{N} \to \mathbb{Z}^*$ to define the values stored in the array C of constants.

The word-size function $\mathscr{W}(n)$ gives the number of bits w of a single word in memory for any instance of size n. Consider the restrictions on the behaviour of $\mathscr{W}(n)$. First, machine words must be large enough to hold every integer that appears in the input or output. For a given problem $\mathscr{S}$, define $\mathscr{M}_{\mathscr{S}}(n)$ to be the largest integer appearing in any valid size-n instance in $\mathscr{S}$. Then we must have $\mathscr{W}(n) \geq \log_2 \mathscr{M}_{\mathscr{S}}(n)$.

Furthermore, machine words must be large enough to address memory. To address the input and output, this immediately implies that we must have $\mathscr{W}(n) \geq \log_2 n$. But the algorithm may use more than n cells of memory in temporary storage. This is why the word-size function $\mathscr{W}(n)$ must be defined by the algorithm and not by the problem. However, we should be careful in allowing the algorithm to set the word size, as setting $\mathscr{W}(n)$ to be very large could make many computations trivial.

This "cheat" is avoided by requiring that algorithms may only increase the word-size by a constant factor. Specifically, we require

$$\mathscr{W}(n) \in O\left(\log_2 n + \log_2 \mathscr{M}_{\mathscr{S}}(n)\right).$$

A consequence is that the IMM model is sufficiently rich to describe the computation of any function in **PSPACE**.

There is one more comment to be made on the word size. The bound $\mathscr{M}_{\mathscr{S}}(n)$ comes from the problem itself, but for some problems this is also flexible in a certain sense. For instance, consider algorithms whose input or output contains arbitrary-precision integers. Encoding every integer into a single word clearly violates the principle of the model, and eliminates from discussion any algorithm for multiple-precision arithmetic. But how should a long integer be encoded into multiple words?

To address this issue, we say a problem is *scalable* if the size of integers in a length-n instance are proportional to $\log_2 n$. More precisely, a problem $\mathscr{S}$ is scalable if $\mathscr{M}_{\mathscr{S}}(n) \in O(\log n)$. Roughly speaking, this means that if the problem can be encoded into any actual computer with fixed machine precision, then the word-size and word operations in any algorithm for that problem correspond to a constant number of words and word operations on that real computer.

The third component of an algorithm solving a problem on an IMM is the constants computation function $\mathscr{C}(n)$. This gives the values stored in the array of constants C for any instance size n. To avoid gratuitous cheats such as computing all possible answers as constants, we also require that the number of constants, $\#C$, be fixed for the algorithm independently of the instance size. That is, $\#C$ does not vary according to the instance size, although the *values* in C may. The $\mathscr{C}(n)$ function must be computable, and in most practical situations will be trivial.

What it means to say that a given IMM algorithm correctly solves a given IMM problem should be clear. We now turn to the cost analysis of an algorithm A. The time complexity of A, written $T(n)$, is the maximum over all instances of size n in $\mathscr{S}$ of the number of steps taken by A to produce O' in the output space and halt. Importantly, this does *not* include the time to compute $\mathscr{W}(n)$ and $\mathscr{C}(n)$. In the case that $\mathscr{W}(n)$ and $\mathscr{C}(n)$ can be computed in $O(T(n))$ time by an IMM with word size exactly $\log_2 n$ and absolutely fixed constants, we say that the algorithm A is *universal*.

The *space complexity* of algorithm A, written $S(n)$, is based only on the size of M_T, the temporary working space. It is defined as the maximum i such that $M_T[i]$ is accessed in any instruction during the execution of any valid instance of size n. Note that this differs crucially from most notions of space complexity, in models where the input and output exist in read-once and write-once tapes, respectively. We will see how the ability to read from *and* write to the output space breaks at least some lower bounds.

Some portions of this thesis examine *in-place* algorithms. Intuitively, this means that the output is overwritten with the input. More precisely, we say a problem given by $\mathscr{S}$ is *in-output* if I is empty for every $(I, O, O') \in \mathscr{S}$. That is, the data used as input for the algorithm is stored in the initial configuration of the read-write output space. An algorithm for such a problem is said to be *in-place* if it uses only $O(1)$ temporary space in M_T. To our knowledge, ours is the first abstract computational model that allows a formal description of such problems and algorithms.

The reader is referred to Angluin and Valiant (1979) for connections between time complexity in the IMM (equivalently RAC) model and the more common RAM model. In particular, an algorithm with time complexity $T(n)$ on a log-cost RAM can be simulated by an IMM with the same cost, and an IMM algorithm with time complexity $T(n)$ can be simulated on a log-cost RAM with at most $O(T(n)\log^2 n + n\log n)$ operations.

Finally, some algorithms will make use of randomization to improve performance. To this end, define a *randomized IMM* to be equivalent to the normal IMM with an additional instruction RAND(i) that chooses an integer uniformly and randomly between 0 and $2^{\mathscr{W}(n)} - 1$ and stores its value in $M_T[i]$.

1.3.3 Storage specification for elements in common rings

The subject of this thesis is computations with polynomials in $\mathsf{R}[x_1, \ldots, x_n]$, where the sorts of domains we have in mind for the ring R are the integers $\mathbb{Z}$, rationals $\mathbb{Q}$, finite fields $\mathbb{F}_q$ where q is a prime power, and floating-point approximations to real and complex numbers in $\mathbb{R}$ and $\mathbb{C}$. For completeness, we discuss briefly how elements in these rings can be stored in the memory of an IMM.

A natural number $a \in \mathbb{N}$ is stored in the 2^w-adic representation as a list $(a_0, a_1, \ldots, a_{n-1})$ with $0 \le a_i < 2^w$ for all i, and $a = a_0 + a_1 2^w + a_2 2^{2w} + \cdots + a_{n-1} 2^{(n-1)w}$. So each a_i fits in a single word of memory, and the entire representation of a may be stored either contiguously or (with twice as much space) in a linked list. To extend this to all integers, we add an additional word of storage for the sign. In any case, the size of the representation in memory is $O(\frac{1}{w}\log a)$ words of storage.

In the case of scalable problems as defined in the previous subsection, we conclude that $O(n)$ words in IMM (or RAC) memory can be used to represent any $(n\log_2 n)$-bit integer, and conversely a single n-bit integer can always be represented in $O(n/(\log n))$ words. (To be more pedantic, we might additionally specify that the representation of a is preceded by a single word indicating the length of the representation.)

A rational number in $\mathbb{Q}$ can always be written n/d with $n, d \in \mathbb{Z}$ relatively prime and $d > 0$. Such a number is simply represented as the pair (n, d).

Elements in a modular ring $\mathbb{Z}/m\mathbb{Z}$ are stored as integers in the range $\{0, 1, \ldots, m-1\}$. This handles the case of finite fields $\mathbb{F}_p$ with prime order. For extension fields of size p^e, we will work in the isomorphic field $\mathbb{F}_p/\langle\Gamma\rangle$ for $\Gamma \in \mathbb{F}_p[x]$ irreducible of degree e. Then we represent elements in $\mathbb{F}_{p^e}$ by polynomials in $\mathbb{F}_p[x]$ with degree less than e using the dense representation, as an array of e elements from $\mathbb{F}_p$.

Consider a real number $\alpha \in \mathbb{R}$ in the range $(0, 1)$, and any chosen small $\epsilon > 0$. To store an ϵ-approximate representation of α, we use an integer a stored in $k = \left\lceil \frac{1}{w} \log_2 \frac{1}{\epsilon} \right\rceil$ words, satisfying $\left|\alpha - 2^{-kw} a\right| < \epsilon$. This can be extended to any real number β by adding an integer exponent b such that $\left|\beta - 2^{b-kw} a\right| < \epsilon \cdot 2^b$. An approximation to a complex number $\gamma \in \mathbb{C}$ is stored as a pair of approximate real numbers, representing the rectangular coordinates for γ.

For any of these rings, and for an element $u \in \mathsf{R}$ in the ring, we write size(u) for the number of machine words required to represent u (where the parameter w for the word-size is understood). So for instance if $u \in \mathbb{N}$ is a nonnegative integer, size$(u) = \lceil \log_w(u) \rceil$.

Observe that addition in any of these rings is easily accomplished in linear time in the size of their representation. We will discuss the cost of multiplication in the next subsection. An important subroutine is modular multiplication with word-sized operands: given $a, b, c \in \mathbb{N}$ with $0 \le a, b, c < 2^w$, computing $r \in \mathbb{N}$ with $0 \le r < c$ such that $a \cdot b \equiv r \bmod c$. First the multiplication $a \cdot b$ is performed to double-word precision, by splitting each a, b in half via a division with remainder by $2^{\lfloor w/2 \rfloor}$, then performing four word-size multiplications and some additions. The division with remainder by c can then be accomplished using similar techniques. In summary, the modular multiplication subroutine can be performed in a constant number of word operations in the IMM.

1.3.4 Algebraic IMM

The results in the previous subsection handle storage and manipulation of algebraic objects of various types within the usual IMM model. However, it is often convenient to describe and analyse an algorithm independently of the particular domain of computation. For instance, the same algorithm might work over an arbitrary field, whether it be a prime field, an extension field, the rational numbers, or any number of other fields. To present the algorithm concisely, it would be useful to perform basic arithmetic in an arbitrary algebraic domain.

The concept of an algebraic RAM was developed to handle mathematical computations involving integers as well as elements from an arbitrary domain. The idea is a usual RAM with a single program but duplicated storage: two input tapes, two output tapes, and two memory banks. (Often the input and output are only on one side.) The *arithmetic* side of the RAM is in the usual model and computes with arbitrary-length integers, while the *algebraic* side computes with elements from the chosen algebraic domain. This seems to be the dominant model (whether or not explicitly stated) in algorithms for symbolic computation.

Along these lines, we extend our concept of IMM to an algebraic IMM. The four memory banks are duplicated on the arithmetic and algebraic sides, albeit not necessarily of the

same size. All instructions can be performed on either side of the IMM, but the two may never be mixed. That is, it is impossible to (explicitly) copy values between the algebraic and arithmetic sides. Furthermore, the specific arithmetic operations for the algebraic side might differ from the ADD, SUB, MUL, and QUO of the arithmetic side, depending on the domain and the specific problem.

An algorithm for an algebraic IMM consists of a single list of instructions, the two functions $\mathscr{W}(n)$ and $\mathscr{C}(n)$, and a third function $\mathscr{A}(n)$ to generate the constants on the algebraic side. This function may produce different values for different domains *and* different instance sizes n, but the size of the constant array on the algebraic side must be fixed for any valid domain and any instance size.

A universal algorithm for an algebraic IMM is similar to before, with the additional requirement that $\mathscr{A}(n)$ can be computed in the same time and space complexity.

We could define an additional instruction for choosing random algebraic elements, but this is problematic to define in a sensible generic way. Instead, a randomized algebraic algorithm can define a subset of algebraic elements, stored in any part of the memory (input, constants, output, or temporary working space), and then use a randomly-chosen integer to index into the set. For all applications we are aware of, this will be sufficient to guarantee performance, and also sufficiently general to be applicable in any domain.

Now consider the problem of representing a polynomial $f \in \mathsf{R}[x_1, \ldots, x_n]$ in an algebraic IMM. In the dense representation, the entire polynomial can be stored in the algebraic side, perhaps with some word-sized integers for the size on the arithmetic side. The recursive dense representation is actually the most intricate of the three, and will be stored mostly on the arithmetic side, with pointers to ring elements on the algebraic side at the bottom level of recursion only. Finally, the sparse representation of a t-sparse polynomial will consist of a length-t array of nonzero coefficients on the algebraic side, coupled with a length-nt array of exponents on the arithmetic side.

1.4 Warm-up: $O(n \log n)$ multiplication

We now show the implications of the IMM model with an example: multiplication of multiple-precision integers in $\mathbb{Z}$. This is undoubtedly one of the most well-studied problems in mathematical computation, and dense multiplication algorithms also hold a central importance in this thesis. It is therefore not only instructive but prudent for us to examine the problem in our chosen model of computation. Furthermore, some of the number-theoretic tools we use here will come up again later in this thesis.

1.4.1 Summary of previous algorithms

The traditional model of study for integer multiplication algorithms is that of bit complexity. So consider the problem of multiplying two n-bit integers, that is, $a, b \in \mathbb{N}$ such that $0 \leq a, b < 2^n$.

The naïve algorithm for multiplication uses $O(n^2)$ word operations. Karatsuba's algorithm (Karatsuba and Ofman, 1963) uses a divide-and-conquer approach to improve the complexity to $O(n^{\log_2 3})$, or $O(n^{1.59})$. A family of divide-and-conquer algorithms due to Toom (1963) and Cook (1966) improves this to $O(n^{1+\epsilon})$, for any positive constant ϵ. Schönhage and Strassen (1971) make use of the Fast Fourier Transform algorithm to improve the bit complexity to $O(n \log n \log\log n)$. Recently, this technique has been carefully refined by Fürer (2007) to achieve $O(n \log n 2^{\log^* n})$ bit complexity for multiplication [1].

Knuth (1981, §4.3.3) discusses most of these algorithms from both a practical and a theoretical viewpoint. Of note here is an idea presented by Knuth but attributed to Schönhage to achieve $O(n \log n)$ complexity in a log-cost RAM for multiplication. The idea is to use complex number arithmetic and use the so-called Four Russians trick (Arlazarov, Dinic, Kronrod, and Faradžev, 1970) to precompute all products under a certain size. In the storage modification machine (SMM) model, Schönhage (1980, §6) shows how to achieve $O(n)$ time for n-bit integer multiplication, which is also similar to our result here. (By way of comparison, Schönhage explicitly states that the goal of his model is not to correspond to any "physical realization", but rather to develop a flexible and general model for its own sake. Our IMM model has the opposite motivation.)

1.4.2 Integer multiplication on an IMM

The algorithm presented here achieves a similar result to Schönhage's $O(n \log n)$ algorithm for a RAM, but without the need for extensive precomputation, and using only modular arithmetic. The idea is similar to the "Three primes FFT integer multiplication" algorithm of von zur Gathen and Gerhard (2003, §8.3), extended to arbitrary word sizes. We now proceed to describe the problem and our algorithm for the IMM model.

As is standard, for simplicity we restrict our attention to the case that both multiplication operands are of roughly the same size. A valid instance in $\mathscr{S}$ will consist of input I containing two integers a and b, each n words long, output O initially empty, and O' — the desired output — consisting of the product ab written in $2n$ words of memory. The total instance size is therefore $4n$, and this completes the formal description of the problem.

Write $m = \log_2 \max(4n, \mathscr{M}_{\mathscr{S}}(n))$, the lower bound on the word size implied by the problem definition. We assume the input integers a and b are written in the 2^m-adic representation as

$$a = a_0 + a_1 \cdot 2^m + a_2 \cdot 2^{2m} + \cdots + a_{n-1} \cdot 2^{(n-1)m}$$
$$b = b_0 + b_1 \cdot 2^m + b_2 \cdot 2^{2m} + \cdots + b_{n-1} \cdot 2^{(n-1)m}.$$

Now define the polynomial $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} \in \mathbb{Z}[x]$ and $B(x) \in \mathbb{Z}[x]$ similarly. Our approach is to compute the product $A(x) \cdot B(x) = C(x) \in \mathbb{Z}[x]$, then evaluate $C(2^m)$ to compute the final result and write it again in the 2^m-adic representation.

[1]The iterated logarithm, denoted $\log^* n$, is the extremely slowly-growing function defined as the number of times logarithm must be taken to reduce n to 1. So, for instance, if $n > 1$, $\log^* n = \log^*(\log n) + 1$.

Let k be the least power of 2 greater than $2n$, i.e., $k = 2^{\lceil \log_2 n \rceil + 1}$. Our algorithm works by first choosing a set of primes $\mathscr{P}$ such that for each $p \in \mathscr{P}$, $(p-1)$ is divisible by k, and the product of the primes in $\mathscr{P}$ is at least 2^{3m}.

From the fact that $\deg A, \deg B < n$ and every coefficient of A and B is at less than 2^m, we can see that every coefficient in C is less than $n \cdot 2^{2m}$, which is less than 2^{3m} from the definition of m. Therefore computing the coefficients of $C \bmod p$ for each $p \in \mathscr{P}$, followed by Chinese remaindering, will give the actual integer coefficients of C.

Furthermore, since the multiplicative group of integers modulo each prime $p \in \mathscr{P}$ is divisible by a power of 2 greater than $\deg C$, the coefficients of $C \bmod p$ can be efficiently computed using the Fast Fourier Transform, using $O(n \log n)$ operations in $\mathbb{F}_p$, assuming a k'th primitive root of unity modulo p (which must exist) is known in advance. Denote such a root of unity ω_p for each $p \in \mathscr{P}$.

The crucial question that remains is how large each p must be to satisfy these conditions. For this, we turn to analytic number theory, and in particular Linnik's theorem (which we somewhat simplify for our particular purposes):

Fact 1.2. *(Linnik, 1944) There exist absolute constants q_L, c_L, L such that, for all $q \geq q_L$, the least prime p such that $p \equiv 1 \bmod q$ satisfies $p \leq c_L q^L$.*

Unfortunately, despite the large body of work devoted to finding concrete values for the exponent L, it appears that no proofs give explicit values for both q_L and c_L. Although the exponent L has been improved recently by Xylouris (2009), the most useful result for us is from Heath-Brown (1992), where it is proven that the inequality above holds for $q_L = 1$, $L = 5.5$, and c_L is an effectively computable absolute constant.

From this, we have the following.

Lemma 1.3. *There exists an absolute constant c such that, for any $k, m \in \mathbb{N}$ as above, there exists a set of at most three primes $\mathscr{P}$ with $\prod_{p \in \mathscr{P}} p \geq 2^{3m}$ and, for each $p \in \mathscr{P}$,*

1. *k divides $(p-1)$*
2. *$p \leq c 2^{16.5m}$.*

Proof. Let c_L be the (effectively computable) constant from the version of Linnik's theorem by Heath-Brown (1992).

Let p_1 be the least prime congruent to 1 modulo 2^m. We have that $p_1 < c_L (2^m)^{5.5} = c_L \cdot 2^{5.5m}$.

If $p_1 \geq 2^{3m}$, then we simply set $\mathscr{P} = \{p_1\}$. Otherwise, let p_2 be the least prime congruent to 1 modulo $2^{\lceil \log_2 p_1 \rceil}$. Since p_2 has a divisor greater than p_1, clearly $p_2 \neq p_1$, and furthermore we see that 2^m divides $(p_2 - 1)$. Finally, $2^{\lceil \log_2 p_1 \rceil} < 2^{3m+1}$, and therefore $p_2 \leq c_L (2^{3m+1})^{5.5} < 64 c_L \cdot 2^{16.5}$.

If $p_1 p_2 \geq 2^{3m}$, then set $\mathscr{P} = \{p_1, p_2\}$. Otherwise, let p_3 be the least prime congruent to 1 modulo $2^{\lceil \log_2 p_2 \rceil}$ as previously and observe that $p_3 < 64 c_L \cdot 2^{16.5}$ as well.

At this point, since each of p_1, p_2, p_3 is greater than 2^m, their product is at least 2^{3m}. So set $\mathscr{P} = \{p_1, p_2, p_3\}$ in this case.

From the definition of m, $4n < 2^m$. Recalling also that k is the least power of two greater than $2n$, it must be the case that $k \mid 2^m$, and therefore k divides each $p_i - 1$. Letting $c = 64c_L$, we get the stated result. □

For our IMM integer multiplication algorithm, then, we set $\mathscr{W}(n)$ — the word size for instance size n — to be $\mathscr{W}(n) = \lceil \log_2 c \rceil + 16.5m$. This means that each $p \in \mathscr{P}$ will fit into a single word in memory. Clearly $\mathscr{W}(n)$ is bounded by

$$\mathscr{W}(n) \in O(\log n + \log \mathscr{M}_{\mathscr{S}}(n)) = O(m),$$

so this is allowable in the IMM model.

The constants defined by $\mathscr{C}(n)$ for our algorithm will consist of the (at most three) primes in $\mathscr{P}$, along with the k'th primitive roots of unity modulo each prime in $\mathscr{P}$. For any n, this consists of at most six words of memory, so the constants are also valid for the IMM model.

This completes the description of our algorithm for multiplication in the IMM model. Because arithmetic modulo a word-sized integer can be performed in a constant number of steps, the cost of arithmetic operations in $\mathbb{F}_p$ is constant for each $p \in \mathscr{P}$. Therefore the total cost of the algorithm is simply $O(n \log n)$ word operations, the cost of the modular FFT computations.

This result is summarized in Theorem 1.4 below. Unfortunately, as the constant c_L is not known explicitly, we have only shown the *existence* of an algorithm. Some more careful steps and repeated doubling in the computation of $\mathscr{W}(n)$ and $\mathscr{C}(n)$ could avoid this issue, but we will not go into those details here.

Theorem 1.4. *There exists an IMM algorithm for multiplying n-word integers that runs in time $O(n \log n)$.*

Another unfortunate fact is that the algorithm is not universal, i.e., the word-size and constants cannot be computed in the same time as the algorithm itself. This is because of the cost of searching for each prime $p \in \mathscr{P}$, which could be avoided by either (1) using randomization to randomly choose primes in the progression, or (2) assuming a more realistic bound on the least prime in an arithmetic progression. The latter possibility will be discussed at length in Chapter 8.

1.4.3 Implications

Observe that the problem as defined for integer multiplication in the previous subsection is not *scalable*. That is, the size m of words in the input could be much larger than $\log_2 n$. This actually means that our $O(n \log n)$ algorithm is actually a stronger result, as it applies no matter what the word size is.

To compare the result of Theorem 1.4 against previous work in the bit complexity model, we should restrict the problem to be scalable. The word size for a size-n input is then restricted to $O(\log n)$, meaning that an n-bit integer requires exactly $\Theta(n/(\log n))$ words of storage. Therefore the algorithm described can be used to multiply n-bit integers using $O(n)$ word

operations in the IMM model. This matches the algorithm of Schönhage (1980) in the SMM model but (like that result) can be misleading, as it counts the input in bits but the cost in word operations.

Simulating the algorithm described above on a log-cost RAM is a fairer comparison, and gives bit complexity $O(n \log^2 n)$ for n-bit integer multiplication. This could be improved at least to some extent with a more careful simulation, but we will not attempt that exercise.

The fast integer multiplication algorithm can be applied to other rings represented in words in the IMM, as well as to polynomials with coefficients in such rings and stored in the dense representation, using the Kronecker substitution.

For instance, consider the multiplication of two univariate integer polynomials $A, B \in \mathbb{Z}[x]$ with degrees less than d and all coefficients stored in at most k words. Observe that the instance size n is $\Theta(dk)$. To compute $A \cdot B \in \mathbb{Z}[x]$, simply write down the integers $a = A(2^{(2k+1)w})$ and $b = B(2^{(2k+1)w})$, and multiply them. Since each integer is stored in $O(dk) = O(n)$ words, the cost of this multiplication is $O(n \log n)$. Furthermore, each coefficient in $A \cdot B$ is at most $d \cdot (2^{wk})^2 \le 2^{(2k+1)w}$, so the coefficients of $A \cdot B$ can simply be read off from the integer product ab.

This same idea can be extended to any $\mathsf{R}[x_1, \ldots, x_m]$, with R any of the rings mentioned in subsection 1.3.3 to perform dense polynomial multiplication in time $O(n \log n)$, where n is the size of a single instance in the IMM. However, importantly, we do *not* have a $O(n \log n)$ algorithm for multiplication of dense polynomials over an arbitrary ring R in the algebraic IMM model. This is because there is no way to encode the elements of an arbitrary ring into integers. In this case, the best result is still that of Cantor and Kaltofen (1991), giving an algorithm for polynomial multiplication in an algebraic IMM using $O(n \log n \operatorname{loglog} n)$ ring operations.

In summary, we see that the IMM model can allow very slight improvements over the best known bit complexity results. However, these differences are quite minor, and furthermore we would argue that the IMM model gives a more accurate measure of the actual cost on actual physical machines. In the remainder, the IMM model will form the basis of our discussions and analysis, although the algorithms will not be presented so formally as they have been here. Furthermore, most of our results will apply equally well to more common models such as (algebraic) RAMs.

Bibliography

Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, September 1988. ISSN 0001-0782. doi: 10.1145/48529.48535. Referenced on page 11.

D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155 – 193, 1979. ISSN 0022-0000. doi: 10.1016/0022-0000(79)90045-X. Referenced on pages 10, 11 and 14.

V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev. The economical construction of the transitive closure of an oriented graph. *Dokl. Akad. Nauk SSSR*, 194:487–488, 1970. ISSN 0002-3264. Referenced on page 17.

Eric Bach and Jeffrey Shallit. *Algorithmic number theory. Vol. 1*. Foundations of Computing Series. MIT Press, Cambridge, MA, 1996. ISBN 0-262-02405-5. Referenced on page 10.

Walter Baur and Volker Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22(3):317 – 330, 1983. ISSN 0304-3975. doi: 10.1016/0304-3975(83)90110-X. Referenced on page 5.

E. R. Berlekamp. Factoring polynomials over finite fields. *Bell System Tech. J.*, 46:1853–1859, 1967. ISSN 0005-8580. Referenced on page 7.

A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, STOC '80, pages 294–301, New York, NY, USA, 1980. ACM. ISBN 0-89791-017-6. doi: 10.1145/800141.804677. Referenced on page 11.

David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991. ISSN 0001-5903. doi: 10.1007/BF01178683. Referenced on pages 7 and 20.

David G. Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Math. Comp.*, 36(154):587–592, 1981. ISSN 0025-5718. doi: 10.2307/2007663. Referenced on page 7.

Bernard Chazelle. A spectral approach to lower bounds with applications to geometric searching. *SIAM Journal on Computing*, 27(2):545–556, 1998. doi: 10.1137/S0097539794275665. Referenced on page 11.

Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973. ISSN 0022-0000.
doi: 10.1016/S0022-0000(73)80029-7. Referenced on page 10.

Stephen Arthur Cook. *On the minimum computation time of functions*. PhD thesis, Harvard University, 1966. Referenced on page 17.

James H. Davenport and Jacques Carette. The sparsity challenges. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2009 11th International Symposium on*, pages 3 –7, September 2009.
doi: 10.1109/SYNASC.2009.62. Referenced on page 7.

Mark J. Encarnación. Black-box polynomial resultants. *Information Processing Letters*, 61(4): 201–204, 1997. ISSN 0020-0190.
doi: 10.1016/S0020-0190(97)00016-1. Referenced on page 5.

Richard Fateman. Draft: Comparing the speed of programs for sparse polynomial multiplication. Online, July 2002.
URL http://www.cs.berkeley.edu/~fateman/algebra.html. Referenced on page 6.

Michael Filaseta, Andrew Granville, and Andrzej Schinzel. Irreducibility and greatest common divisor algorithms for sparse polynomials. In *Number theory and polynomials*, volume 352 of *London Math. Soc. Lecture Note Ser.*, pages 155–176. Cambridge Univ. Press, Cambridge, 2008.
doi: 10.1017/CBO9780511721274.012. Referenced on page 8.

Timothy S. Freeman, Gregory M. Imirzian, Erich Kaltofen, and Lakshman Yagati. Dagwood: A system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Softw.*, 14:218–240, September 1988. ISSN 0098-3500.
doi: 10.1145/44128.214376. Referenced on page 5.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297, 1999.
doi: 10.1109/SFFCS.1999.814600. Referenced on page 11.

Martin Fürer. Faster integer multiplication. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, STOC '07, pages 57–66, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-631-8.
doi: 10.1145/1250790.1250800. Referenced on page 17.

Sanchit Garg and Éric Schost. Interpolation of polynomials given by straight-line programs. *Theoretical Computer Science*, 410(27-29):2659–2662, 2009. ISSN 0304-3975.
doi: 10.1016/j.tcs.2009.03.030. Referenced on page 3.

Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, Cambridge, second edition, 2003. ISBN 0521826462. Referenced on page 17.

Etienne Grandjean and J. Robson. RAM with compact memory: a realistic and robust model of computation. In E. Börger, H. Büning, M. Richter, and W. Schönfeld, editors, *Computer Science Logic*, volume 533 of *Lecture Notes in Computer Science*, pages 195–233. Springer Berlin / Heidelberg, 1991.
doi: 10.1007/3-540-54487-9_60. Referenced on page 10.

David Harvey. Faster polynomial multiplication via multipoint Kronecker substitution. *Journal of Symbolic Computation*, 44(10):1502–1510, 2009. ISSN 0747-7171.
doi: 10.1016/j.jsc.2009.05.004. Referenced on page 9.

D. R. Heath-Brown. Zero-free regions for Dirichlet L-functions, and the least prime in an arithmetic progression. *Proc. London Math. Soc.*, s3-64(2):265–338, March 1992.
doi: 10.1112/plms/s3-64.2.265. Referenced on page 18.

Joris van der Hoeven. The truncated Fourier transform and applications. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, ISSAC '04, pages 290–296, New York, NY, USA, 2004. ACM. ISBN 1-58113-827-X.
doi: 10.1145/1005285.1005327. Referenced on page 2.

Stephen C. Johnson. Sparse polynomial arithmetic. *SIGSAM Bull.*, 8:63–71, August 1974. ISSN 0163-5824.
doi: 10.1145/1086837.1086847. Referenced on page 7.

Erich Kaltofen. Factorization of polynomials given by straight-line programs. In *Randomness and Computation*, pages 375–412. JAI Press, 1989. Referenced on page 5.

Erich Kaltofen and Pascal Koiran. On the complexity of factoring bivariate supersparse (lacunary) polynomials. In *ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, pages 208–215, New York, NY, USA, 2005. ACM. ISBN 1-59593-095-7.
doi: 10.1145/1073884.1073914. Referenced on page 7.

Erich Kaltofen and Barry M. Trager. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *Journal of Symbolic Computation*, 9(3):301–320, 1990. ISSN 0747-7171.
doi: 10.1016/S0747-7171(08)80015-6. Computational algebraic complexity editorial. Referenced on pages 5 and 7.

Erich L. Kaltofen. The "seven dwarfs" of symbolic computation. Manuscript prepared for the final report of the 1998-2008 Austrian research project SFB F013 "Numerical and Symbolic Scientific Computing," Peter Paule, director, April 2010.
URL http://www.math.ncsu.edu/~kaltofen/bibliography/10/Ka10_7dwarfs.pdf. Referenced on page 2.

A. A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 7:595–596, 1963. Referenced on pages 7 and 17.

Donald E. Knuth. *The art of computer programming, Volume 2: seminumerical algorithms.* Addison-Wesley, Boston, MA, 1981. ISBN 0-201-89684-2. Referenced on pages 8 and 17.

A. N. Kolmogorov and V. A. Uspenskiĭ. On the definition of an algorithm. *Uspehi Mat. Nauk*, 13(4(82)):3–28, 1958. ISSN 0042-1316.
URL http://mi.mathnet.ru/eng/umn7453. Referenced on page 10.

Leopold Kronecker. Grundzüge einer arithmetischen Theorie der algebraischen Grössen. *Journal Für die reine und angewandte Mathematik*, 92:1–122, 1882. Referenced on page 8.

H. W. Lenstra, Jr. Finding small degree factors of lacunary polynomials. In *Number theory in progress, Vol. 1 (Zakopane-Kościelisko, 1997)*, pages 267–276. de Gruyter, Berlin, 1999. Referenced on page 7.

U. V. Linnik. On the least prime in an arithmetic progression. II. The Deuring-Heilbronn phenomenon. *Rec. Math. [Mat. Sbornik] N.S.*, 15(57):347–368, 1944. Referenced on page 18.

Philippe Pébay, J. Maurice Rojas, and David C. Thompson. Optimizing n-variate $(n+k)$-nomials for small k. *Theoretical Computer Science*, In Press, Corrected Proof, 2010. ISSN 0304-3975.
doi: 10.1016/j.tcs.2010.11.053. Referenced on page 8.

Nitin Saxena. Progress on polynomial identity testing. *Bull. EATCS*, 99:49–79, 2009. Referenced on page 5.

A. Schönhage. Storage modification machines. *SIAM Journal on Computing*, 9(3):490–508, 1980.
doi: 10.1137/0209036. Referenced on pages 10, 17 and 20.

A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971. ISSN 0010-485X.
doi: 10.1007/BF02242355. Referenced on pages 7 and 17.

Arnold Schönhage, Andreas F. W. Grotefeld, and Ekkehart Vetter. *Fast algorithms.* Bibliographisches Institut, Mannheim, 1994. ISBN 3-411-16891-9. A multitape Turing machine implementation. Referenced on page 10.

David R. Stoutemyer. Which polynomial representation is best? Surprises abound! In *Proc. 1984 Macsyma users' conference*, pages 221–243, Schenectady, New York, 1984. Referenced on page 6.

A. L. Toom. The complexity of a scheme of functional elements simulating the multiplication of integers. *Doklady Akademii Nauk SSSR*, 150:496–498, 1963. ISSN 0002-3264. Referenced on page 17.

A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.
doi: 10.1112/plms/s2-42.1.230. Referenced on page 10.

Triantafyllos Xylouris. On Linnik's constant. Technical report, arXiv:0906.2749v1 [math.NT], 2009.
URL http://arxiv.org/abs/0906.2749. Referenced on page 18.

Richard Zippel. Probabilistic algorithms for sparse polynomials. In Edward Ng, editor, *Symbolic and Algebraic Computation*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer Berlin / Heidelberg, 1979.
doi: 10.1007/3-540-09519-5_73. Referenced on page 7.

Richard Zippel. Interpolating polynomials from their values. *Journal of Symbolic Computation*, 9(3):375–403, 1990. ISSN 0747-7171.
doi: 10.1016/S0747-7171(08)80018-1. Computational algebraic complexity editorial. Referenced on page 7.