

# Discrete Logarithms in Finite Fields

Michele Mosca

Wolfson College

M.Sc. in Mathematics and the Foundations of Computer Science  
Faculty of Mathematical Sciences  
University of Oxford

September 1996

## Acknowledgements

I would like to thank Peter Neumann and Dominic Welsh for their very helpful co-supervision of this M.Sc. dissertation, and for their teaching and encouragement throughout the course.

I am also indebted to Scott Vanstone and Ron Mullin for the teaching and support they have provided me over the years and for introducing me to this problem while I was an undergraduate.

Many thanks to Alfred Menezes for the countless references, to Carl Pomerance for telling me about the existence of [BP96] and [SWD96], and to Ian Blake for other references. I am also very grateful to Carl Pomerance, Renet Lovorn Bender and Oliver Schirokauer for providing me with copies of their work. Thanks to Artur Ekert, Adriano Barenco, Richard Cleve, and Chiara Macchiavello for their *quantum* help. Thanks to Robert Zuccherato, Simon Blackburn, Shuhong Gao, and to Bryan Birch for the helpful conversions and e-mails.

I am also very grateful to the U. K. Commonwealth Commission, and the Natural Sciences and Engineering Research Council of Canada for their financial support.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Basic Notation . . . . .	7
1.2	Factoring . . . . .	7
<b>2</b>	<b>General Techniques</b>	<b>9</b>
2.1	Baby Step Giant Step algorithm . . . . .	9
2.2	Pollard's method . . . . .	10
2.3	Subgroup technique . . . . .	12
2.4	Subgroup techniques applied to finite fields . . . . .	14
<b>3</b>	<b>Index Calculus Techniques</b>	<b>16</b>
3.1	Basic Method . . . . .	16
3.2	Linear Algebra . . . . .	18
3.2.1	Wiedemann's algorithm . . . . .	19
3.2.2	Linear Algebra Issues for Index Calculus . . . . .	24
3.3	Polynomial ring techniques . . . . .	25
3.3.1	Practical Improvements . . . . .	27
3.3.2	Some new improvements to the above techniques . . . . .	31
3.4	Techniques for $\mathbf{GF}(p)$ . . . . .	34
3.5	A More General Look at Index Calculus . . . . .	35
3.6	Other implementations . . . . .	36
<b>4</b>	<b>Quantum Computers</b>	<b>37</b>
4.1	What is a quantum computer? . . . . .	37
4.2	Quantum Computation Tools . . . . .	39
4.3	Quantum discrete logarithm algorithm . . . . .	40
4.3.1	Analysis . . . . .	40
4.4	Concluding remarks . . . . .	42

<b>5</b>	<b>Conclusions</b>	<b>44</b>
<b>A</b>		<b>46</b>
A.1	The expected value of $\gcd(n, N)$ . . . . .	46
A.2	Hensel lifting in non-singular matrices . . . . .	46
A.3	Selecting Parameters for Coppersmith's algorithm . . . . .	46
A.4	Chebyshev correction . . . . .	47
A.5	Toeplitz matrices . . . . .	48
A.6	Finding the order of an element using discrete logarithms . . . . .	49
A.7	The Number of Relatively Prime $(v_1(x), v_2(x))$ Pairs . . . . .	49

# Chapter 1

## Introduction

The logarithm is a very familiar function in mathematics, defined in various settings.

We define the *Discrete Logarithm Problem (DLP)* in a finite cyclic group  $G$  as follows.

**Definition 1** *Given a generator  $\alpha$  of  $G$ , and any  $\beta \in G$ , find the smallest positive integer,  $k$ , such that  $\alpha^k = \beta$ . This  $k$  is called the discrete logarithm of  $\beta$  to the base  $\alpha$ , denoted  $\log_\alpha(\beta)$ .*

Let  $N$  be the order of  $G$ . The DLP in a finite field,  $\mathbf{F}$ , refers to the DLP in the multiplicative group,  $\mathbf{F}^*$ . Throughout this dissertation, an algorithm is *efficient* if its running time in terms of basic operations is bounded by a polynomial in the size of the input. Assume the size of an element of  $G$  is at least  $\log(N)$ . The size might be larger if, for example,  $G$  is a tiny subgroup of a much larger group  $H$ , and elements are represented as elements of  $H$ . This is why I chose to describe the running time of these algorithms in terms of *group operations* and their space requirements in terms of group elements. Since we assume that the group elements have size at least  $\log(N)$ , then calling  $O(\log(N))$ -bit integer multiplication and addition a group operation seems justified, as does referring to a requirement for  $O(\log(N))$  bits of space as requiring space for one group element. *Group operations* will include not only the group multiplication, but a small family of operations whose running time is bounded by a polynomial in the running time of the group multiplication. Parallel or distributed computing can often be used to reduce the actual time it takes to carry out the operations in these algorithms.

Discrete logarithms in finite fields are a useful tool, for example, in analysing linear feedback shift registers for use as event counters (see [CW94]). They are also used to find the position of a subarray in a pseudo-random array, which can be useful in measuring the absolute position of automated guided vehicles (see [LB92]). These are cases where we want fast algorithms for solving the DLP. The cardinality,  $q$ , of the fields does not typically get very large. In the first case, a field of size  $2^n$  should suffice if the counter is not going to exceed  $2^n - 1$ . In the second case, we keep track of positions in two dimensions, and as long as we never walk more than  $n_1$  positions in one direction, or  $n_2$  positions in the other, a field of size not much bigger than  $n_1 n_2$  should suffice.

There does not seem to be an efficient algorithm to solve the DLP in finite fields, whereas it is well-known that the inverse operation, exponentiation, can be performed efficiently. Exponentiation in finite fields is thus believed to be a *one-way* function, and suitable for use in *public-key*

cryptographic protocols (see [MvV96] for a thorough description). In this case, the cryptographer counts on there not existing efficient algorithms for the DLP, and selects fields with parameters such that discrete logarithm computations are infeasible for an adversary.

I present several algorithms for solving this problem, and discuss their running times and space requirements. Most of the algorithms are randomised algorithms. By that I mean that they require random bits as input. Randomness and the existence of random bits are philosophical questions I will not address. For theoretical purposes, I assume the existence of a random bit generator. For practical purposes, I assume that the pseudo-random bit generators in existence are effective. We assume that one call to the random bit generator is one operation. Randomisation can be very useful for solving problems for which deterministic algorithms have a reasonable average running time but have terrible running times for certain inputs. Instead we might produce randomised algorithms which will solve the problem for *any* input with a reasonable expected running time, the average now with respect to the random bit inputs. This idea is discussed in [Joh84] and [MR95]. Some of the randomised algorithms are *probabilistic* algorithms. More details about what we mean by a probabilistic algorithm appear in section 12 of [LP92], and in [Joh84]. The paper [LP92] by Lenstra and Pomerance also includes a useful discussion on analysing the running time of probabilistic algorithms. The expected running time given is an upper bound for the expected running time of the algorithm taken over the random bit inputs. So this upper bound on the expected running time is valid for *any* input. Given a probabilistic algorithm with expected running time  $T$ , for any  $k \geq 1$  Markov's inequality implies that the algorithm will succeed within time  $kT$  with probability at least  $(k - 1)/k$ . By repeating such an algorithm  $n$  times, each for time  $kT$ , the probability of never succeeding is at most  $(1/k)^n$ . Other randomised algorithms we present have an expected running time which is valid for any input, provided certain assumptions are true. I call these *heuristic probabilistic algorithms*. Only two of the algorithms presented, including exhaustive search, are deterministic and guaranteed to work on all inputs. The running times given are the worst case running times, and happen to be within a constant factor of the expected running times.

The fact that we define the discrete logarithm to be the smallest positive integer  $k$  such that  $\alpha^k = \beta$  is not a big restriction. Suppose we could solve for *some* positive integer  $k$  such that  $\alpha^k = \beta$ . Then we have a probabilistic algorithm for finding the order of  $\alpha$  in  $O(\log(N))$  group operations (see appendix A.6 for details). We can thus find the smallest positive integer  $k$  such that  $\alpha^k = \beta$ .

It is well known that all finite fields of the same order are isomorphic and have prime power order. Hence we can refer to *the* field of order  $q$  for any prime power  $q$ . We denote the field of order  $q$  by  $\mathbf{GF}(q)$ , the Galois Field of order  $q$ , after the French mathematician Evariste Galois who started the study of finite fields. It is also often denoted  $\mathbf{F}_q$ . For any integer  $n$ , we denote the integers modulo  $n$  by  $\mathbf{Z}_n$ . For prime  $p$ ,  $\mathbf{Z}_p$  is a field of order  $p$ , and thus also  $\mathbf{GF}(p)$ . One way to represent  $\mathbf{GF}(p^n)$ , for a prime  $p$  and positive integer  $n$ , that will be useful for solving discrete logarithms, is as  $\mathbf{GF}(p)[x]/(f(x))$ , the  $\mathbf{GF}(p)$ -ring of polynomials in  $x$  modulo  $f(x)$ , for an irreducible  $f(x)$  of degree  $n$ .

I assume throughout that we know the size of the fields and the representations of the fields lend themselves to efficient (with respect to the group multiplication) basic arithmetic operations, including addition, comparisons, and inverse-taking. I assume that the prime subfield  $\mathbf{GF}(p)$  is easily identified with  $\mathbf{Z}_p$ , and that given any basis for  $\mathbf{GF}(q)$  as a vector space over  $\mathbf{Z}_p$ , we can

efficiently find the coordinates of any element with respect to this basis.

Algorithms for finding logarithms in  $\mathbf{GF}(q)$  often take advantage of the way we represent the field. In the case of a polynomial ring modulo an irreducible polynomial, we sometimes want the irreducible to have a specific form. We note that it suffices to find one representation in which we can solve the DLP. We know that all finite fields of the same order are isomorphic and I show here how to construct these isomorphisms efficiently. This is discussed in [Zie74] and [Odl85] in the case that we have two representations of  $\mathbf{GF}(p^n)$  of the form  $\mathbf{GF}(p)[x]/(f(x))$  for two different irreducibles  $f(x)$ . Deterministic algorithms are discussed in [Len91]. Assume we have identified the subfield  $\mathbf{GF}(q_1)$  of  $\mathbf{GF}(q)$  in both representations, and we can efficiently represent elements with respect to a basis over  $\mathbf{GF}(q_1)$  (we could let  $q_1$  be the characteristic, for example). Also assume that we have an isomorphism between the two representations of  $\mathbf{GF}(q_1)$ . Suppose  $q = q_1^{n_1}$ . Find an irreducible,  $f(X)$ , of degree  $n_1$  over  $\mathbf{GF}(q_1)$ . Find a root of  $f(X)$  in each representation of  $\mathbf{GF}(q)$ ,  $\alpha$  and  $\bar{\alpha}$ . There exist efficient randomised algorithms for finding irreducible polynomials (see [Nie91]) and for factoring polynomials (see, for example, [Ber70], [CZ81]). An isomorphism is then defined by  $\Psi(\alpha) = \bar{\alpha}$ . Linear algebra over  $\mathbf{GF}(q_1)$  allows us to efficiently represent arbitrary elements as a  $\mathbf{GF}(q_1)$ -linear combination of powers of  $\alpha$  or  $\bar{\alpha}$ , and thus we can efficiently evaluate  $\Psi$  or  $\Psi^{-1}$  at any element. We may also select a different generator,  $\alpha_1$ , instead of  $\alpha$ , since we know  $\log_\alpha(\beta) \log_{\alpha_1}(\alpha) \equiv \log_{\alpha_1}(\beta)$  modulo  $N$ .

To solve logarithms in a given field representation with respect to a given base, we can proceed as follows. Find an algorithm for solving logarithms to the base  $\gamma$  in our representation of choice. Find an isomorphism  $\Psi$  between the given representation and our representation. Solve for  $\log_\gamma(\Psi(\beta))$  and  $\log_\gamma(\Psi(\alpha))$  in our representation. Use these two results to solve for  $\log_{\Psi(\alpha)}(\Psi(\beta))$ , which is by the isomorphism equal to  $\log_\alpha(\beta)$ .

Besides exhaustive search, which is a  $O(e^{\log(q)})$  group operation algorithm for a field of order  $q$ , there are two general classes of algorithms for solving the DLP on a classical computer. The first class contains algorithms which bring the running time down to  $O(q^{\frac{1}{2}}) = O(e^{\frac{\log q}{2}})$  group operations or less if  $q - 1$  has no large factors. These are discussed in Chapter 2, and work in any finite cyclic group. Chapter 3 covers the second class, referred to as *Index Calculus* techniques, which for *most* finite fields give rigorous probabilistic algorithms with expected running time  $e^{O((\log q \log \log q)^{\frac{1}{2}})}$ . Heuristic probabilistic index calculus algorithms exist with the same running time for all finite fields. For many finite fields, there are heuristic probabilistic algorithms with expected running time  $e^{O((\log q)^{\frac{1}{3}} (\log \log q)^{\frac{2}{3}})}$ . It seems possible therefore, to construct finite fields for which finding discrete logarithms is intractable, at least on a classical computer.

However a *quantum computer* would reduce the expected running time down to a polynomial number of quantum operations. The possibility of ever creating a device to perform such calculations is an open problem and currently an area of active research. I briefly discuss quantum computers and a discrete logarithm algorithm in Chapter 4.

Most of the information contained in this thesis is a survey of known techniques. Some of the results are original, including sections 2.4 and 3.3.2, appendices A.7, A.1, and A.4, Algorithm 2.2.2, as well as parts of other sections where I generalise some results and point out what seem to be problems or errors in the literature. I have made every effort to cite the ideas I have taken from others, or ideas I have come up with independently but later discovered to already exist in the literature.

I conclude the introduction with a section to introduce some basic notation used in the subsequent chapters, followed by a brief summary of known integer factoring methods.

## 1.1 Basic Notation

One notion that is used in many of the algorithms is that of *smoothness*. Intuitively, an element is *smooth* if it can be *easily* factored into a product of other elements of *small size*; an element is  $k$ -smooth if it can be factored into a product of elements of size at most  $k$ . I give two specific examples that are used in later sections. Let  $k$  be a non-negative integer.

**Definition 2** *An integer  $N$  is  $k$ -smooth if each of the prime factors of  $N$  is at most  $k$ .*

**Definition 3** *A polynomial  $f(x) \in \mathbf{R}[x]$  is  $k$ -smooth if all the irreducible factors of  $f(x)$  have degree at most  $k$ .*

For two integers  $a$  and  $N$ ,  $N > 0$ , by  $a \bmod N$  I mean the least positive element congruent to  $a$  modulo  $N$ . For two polynomials  $a(x)$  and  $f(x)$ , by  $a(x) \bmod f(x)$  I mean the unique polynomial of degree less than the degree of  $f(x)$  that is congruent to  $a(x)$  modulo  $f(x)$ .

I denote tuples or vectors with a bold variable, and their corresponding entries are denoted in italics by the same variable with subscripts, for example,  $\mathbf{y}$ , with entries  $y_0, y_1, \dots$

## 1.2 Factoring

There exist efficient probabilistic algorithms for testing primality, however there seem to be no efficient randomised algorithms for factoring any integer  $N$  into its prime factors. It suffices to find an algorithm for factoring any composite into two non-trivial factors. Factoring is an important part of many discrete logarithm algorithms that will be discussed later, so I give a brief overview of the current state of the art. Sometimes we want to factor the number completely, while other times we just want to find its small factors, or test if its factors are smaller than some value.

Pollard's *rho* factoring algorithm has a heuristic expected running time of  $O(\sqrt{p})$  modular multiplications to find a factor  $p$  of  $N$ . This is good for finding small factors of  $N$ , and it only requires space for  $O(1)$  group elements. A rigorously proven deterministic algorithm that will find the least prime factor of  $N$  less than  $m$  in time  $m^{1/2+o(1)}$  is the *Pollard-Strassen method*, described in [Pom82]. This method seems to require  $m^{1/2+o(1)}$  space as well. I am not aware of any rigorous techniques that require the same time but less space.

Pollard's  $(p-1)$ -algorithm will usually find a prime factor  $p$  in  $O(B \log(N) / \log(B))$  modular multiplications if every factor of  $p-1$  is at most  $B$ . The elliptic curve factoring algorithm is a generalisation of Pollard's  $(p-1)$ -algorithm with heuristic expected running time  $e^{(\sqrt{2}+o(1))(\log(p) \log \log(p))^{1/2}}$  for uncovering a factor  $p$ . Thus it should factor a general integer  $N$  in time  $e^{(1+o(1))(\log(N) \log \log(N))^{1/2}}$ .

Pomerance [Pom87] rigorously shows that elliptic curve methods can be applied to uncover the prime factors of  $N$  in the set  $S(m)$  with probability at least  $1 - \log(N)/N$  in expected time

$O(\log(N)^4 e^{2(\log(m)^{6/7})})$  where  $S(x)$  is a certain set of primes less than  $x$ . He shows that  $\pi(x) - |S(x)| = O(x/e^{\frac{1}{2}\log(x)^{1/6}})$ , that is, *most* primes less than  $m$  lie in  $S(m)$ .

We can efficiently remove all factors of 2 from an integer, and detect and factor prime powers. Thus we only need to worry about odd integers with more than one distinct prime factor. Given an odd integer  $N$ , with  $\omega(N) = k > 1$  distinct prime factors, there are  $2^k$  positive integers,  $x < N$ , with  $x^2 \equiv 1 \pmod{N}$ . If we find any non-trivial one, then  $\gcd(x - 1, N)$  will be a non-trivial factor of  $N$ . Several algorithms are based on this principle, and these are called *random squares* algorithms. Many attempt to find such  $x$  using index calculus methods we describe in chapter 3. The quantum computer algorithm given by Shor [Sho94b] is a random square algorithm. The quadratic sieve factoring algorithm is a random square index calculus algorithm and has a heuristic expected running time of  $e^{(1+o(1))(\log(n) \log \log(n))^{1/2}}$ . The general number field sieve is another with a heuristic expected running time of  $e^{((8/3)^{2/3} + o(1)) \log(n)^{1/3} \log \log(n)^{2/3}}$ . Pomerance [Pom87] describes a rigorous algorithm with expected running time  $e^{(\sqrt{2} + o(1))(\log(n) \log \log(n))^{1/2}}$ .

Lenstra and Pomerance [LP92] devised the currently fastest known rigorously analysed algorithm with an expected running time of  $e^{((1+o(1))(\log(n) \log \log(n))^{1/2})}$ .

Much of this brief summary was obtained from chapter 3 of [MvV96] where more details and relevant references can be found.

Note that a discrete logarithm algorithm for subgroups of  $Z_N^*$  would give a randomised algorithm for factoring odd  $N$ . This is because a discrete logarithm algorithm would allow us to compute the order of elements in the group  $Z_N^*$ . This can be used to factor  $N$  as is done with Shor's quantum computer factoring algorithm [Sho94b], since if we have an element  $a$  with even order  $2k$ , and  $a^k \neq -1$ , we can find a non-trivial factor of  $N$ .

## Chapter 2

# General Techniques

I describe three techniques for finding logarithms in any cyclic group. The first algorithm does not require knowing the order of the cyclic group, and in fact McCurley [McC90] points out that it was developed in order to find the order of elements (see appendix A.6). The second requires the order of the group as input. The methods of sections 2.1 and 2.2 are called *square root* techniques, because their running times are roughly proportional to the square root of the order of the group. Section 2.3 outlines a method which also requires the order and exploits any known factorisation of  $N$ , using any applicable discrete logarithm algorithm as a subroutine, including the square root methods. In section 2.4 I describe the applicability of the last method to finite fields.

### 2.1 Baby Step Giant Step algorithm

For this algorithm we will need an efficiently testable order relation on the elements of  $G$ . This is discussed at the end of the section. Consider comparisons to be group operations. The first step is to build up a sorted database. Let  $m$  be the size of the database; the database will contain  $m$  ordered pairs, each containing an integer from 1 to  $m$ , and an element from  $G$ . Algorithm 2.1.1 outlines the main steps.

Since  $k = i + mj$  for some integers  $j$  and  $i$  satisfying  $0 \leq i < m$ ,  $0 \leq j < N/m$ , we know this algorithm will find the answer within  $N/m$  iterations of Step 2. For a chosen  $m$ , the precomputation takes  $O(m \log(m))$  group operations. Finding a particular logarithm then costs  $O(\frac{N}{m} \log(m))$  group operations. If we want to minimise worst case or expected running time, we should select  $m$  to be roughly  $\sqrt{N}$ . We could increase the speed of finding individual logarithms by spending more time on the precomputation. Likewise, we could decrease the memory requirements and the precomputation time at the cost of increasing the time spent on finding individual logarithms.

There are several alternatives for what we compute for the database, and what we search for. We could, for example, precompute  $\alpha^{mi}$ ,  $\beta\alpha^i$  or  $\beta\alpha^{mi}$  and then search for matches with  $\beta\alpha^{-j}$ ,  $\alpha^{-mj}$ , or  $\alpha^{-j}$  respectively. The database we describe does not involve  $m$  or  $\beta$ , and thus can easily be enlarged to allow for faster postcomputation of logarithms, and applied to find the logarithm of any element in  $G$ .

---

**Algorithm 2.1.1 Baby-Step Giant-Step Algorithm**

---

1. Select a database size  $m$ .
  2. Compute and sort the set of ordered pairs  $T = \{(\alpha^i, i), i = 0, 1, \dots, m - 1\}$ . This requires  $m$  group multiplications, by the same element  $\alpha$ . To produce the sorted list thus costs  $O(m \log(m))$  moves and comparisons, and  $O(m)$  multiplications.
  3. Precompute  $\alpha^{-m}$ . Since we just computed up to  $\alpha^{m-1}$ , this costs one multiplication and one inverse operation. Set  $j = 0$ .
  4. Compute  $\beta\alpha^{-mj}$ , and search for it in the database. This costs one multiplication and  $O(\log(m))$  comparisons each time.
  5. If no match is found, increment  $j$  and go to step 4; otherwise go to step 6.
  6. When we get a match we know that  $\beta = \alpha^{i+mj}$ .
- 

With regards to the order on  $G$ , if each element has an efficiently computable canonical representative with respect to some alphabet, then lexicographic order will suffice. Any efficient injective mapping  $f : G \rightarrow \mathbf{Z}$ , where the integers are represented in standard binary notation, will work. If we count this mapping as a group operation, it does not affect the running time given. It suffices that the preimage of  $f$  for any element is small. If the largest preimage size is  $r$ , then Step 3 will only be slowed down by the cost of testing up to  $r$  possible matches. In practice one might use a hash function or a hash table, and reduce memory requirements, time spent searching, or both.

The baby-step giant-step method is described in the solution to exercise 17 on page 575 of [Knu73]. It is described for the case that  $G$  is  $\mathbf{GF}(p)^*$ , but it applies to any group. Knuth attributes it to an idea of D. Shanks. Heiman [Hei92] gives a general formulation of this algorithm, which allows us to target exponents with specific structure. Suppose we know the solution lies in a subset  $X$  of the integers modulo  $N$ . Find two sets  $A$  and  $B$  such that  $X = A + B$  modulo  $N$ . By  $A + B$  I mean  $\{a + b | a \in A, b \in B\}$ . By  $X = Y$  modulo  $N$ , I mean  $\{x \bmod N | x \in X\} = \{y \bmod N | y \in Y\}$ . Store a sorted list of the values  $(\alpha^a, a)$  for all  $a \in A$ . Then search for  $\beta\alpha^{-b}$  with  $b \in B$  until a match is found. This requires  $O(|A| \log |A|)$  group operations for the first phase, and  $O(|B| \log |A|)$  group operations for the second phase. The space complexity is  $O(|A|)$  group elements. Clearly  $|X| \leq |A||B|$ , so in the best case, this technique can reduce the method to roughly  $\sqrt{|X|}$  group operations. Some examples of restricted sets of exponents are exponents with bounded or fixed hamming weight, or exponents with some digits fixed in some base representation. See [Hei92] and [MvV96] for more details.

## 2.2 Pollard's method

Pollard [Pol78] presents a method that has about the same running time as the baby-step giant-step algorithm, but only requires space for  $O(1)$  group elements.

Partition  $G$  into three sets of roughly the same size,  $S_1, S_2$ , and  $S_3$ . Recursively define a

sequence  $x_i, i = 0, 1, \dots$  as follows. Let  $x_0 = 1$ , and for all  $i \geq 0$ , let

$$x_{i+1} = \begin{cases} \alpha x_i & \text{if } x_i \in S_1 \\ x_i^2 & \text{if } x_i \in S_2 \\ \beta x_i & \text{if } x_i \in S_3 \end{cases} .$$

Consequently,  $x_i = \alpha^{a_i} \beta^{b_i}$  for an easily computable pair of sequences  $\{a_i\}$  and  $\{b_i\}$  modulo  $N$ . If we do not know  $N$ , we could not reduce modulo  $N$ , and the space requirements for these elements would be exponential in  $i$ . It suffices to know a small multiple of  $N$ . This sequence will eventually cycle, thus producing a  $\rho$ -shaped sequence. The general outline of the algorithm is given in Algorithm 2.2.1.

---

**Algorithm 2.2.1** Pollard's method

---

1. Let  $x_0 = 1, a_0 = b_0 = 0$ . Compute the sequence of 6-tuples,  $(x_i, a_i, b_i, x_{2i}, a_{2i}, b_{2i})$ ,  $i = 1, 2, 3, \dots$  until we get  $x_i = x_{2i}$ .
  2. Compute  $m = a_i - a_{2i} \bmod N$  and  $n = b_{2i} - b_i \bmod N$ . Consequently  $\alpha^m \equiv \beta^n$ . It thus follows that  $nk \equiv m \bmod N$ , where  $k = \log_\alpha(\beta)$ .
  3. Let  $d = \gcd(N, n)$ . Solve  $\frac{n}{d}k \equiv m \bmod N/d$ .
- 

I show in the appendix that the expected value of  $d = \gcd(n, N)$  is at most  $d(N)$ , the number of divisors of  $N$ , provided  $n$  is selected uniformly at random from the set of integers modulo  $N$ . It is easy to see that  $d(n) < 2\sqrt{N}$  since there is one-to-one correspondence between divisors of  $N$  less than  $\sqrt{N}$  and those greater than  $\sqrt{N}$ . In fact, we know that  $d(N) = N^{o(1)}$  ([HW56], page 262).

So assuming that the expected greatest common divisor of  $n$  and  $N$  is about the same as for a randomly selected integer  $n$  between 1 and  $N$ , the expected time to be able to solve for the logarithm modulo a divisor of size at least  $N/2d(N)$  is 2 repetitions of the pseudo-random walk. At each iteration we randomise by replacing  $\beta$  with  $\beta\alpha^r$  for some randomly chosen positive integer  $r < N$ . Once we have the integer  $k, 0 \leq k < N/d$ , such that  $\beta = \alpha^{k+iN/d}$  for some  $i, 0 \leq i < d$ , we solve for  $i$ . We do this by solving for  $\log_{\alpha_1}(\beta_1)$  modulo  $d$  where  $\alpha_1 = \alpha^{N/d}$  and  $\beta_1 = \beta/\alpha^k$ .

The *collision*  $x_i = x_{2i}$  will occur precisely when  $i$  is a multiple of the length of the cycle part of the  $\rho$ , and greater than the tail length. This method for finding cycles has been attributed to Floyd [Pol78, MvV96]. The first  $i$  for which  $x_i = x_{2i}$  is called the *epact* [Pol78]. Pollard points out that for a true random mapping on  $\mathbf{GF}(p)^*$  the expected value of the epact is close to  $\sqrt{\pi^5 p/288} \approx 1.0309\sqrt{p}$ . His experiments gave a mean value of 1.08 for the constant, with some values as large as  $3\sqrt{p}$ . In his case,  $S_i = \{x | \frac{i-1}{3}p < x < \frac{i}{3}p\}$ . Assuming that the expected epact is  $O(\sqrt{N})$ , and assuming that  $d = \gcd(n, N)$  behaves as it would for a random integer  $n$  from 1 to  $N$ , the expected running time to solve for the logarithm modulo  $N$  is at most  $O(\sqrt{N})$  operations.

However, depending on the choice of sets  $S_i$  and of  $\alpha$  and  $\beta$ , the epact could be  $O(N)$ . With some minor modifications, we get a heuristic probabilistic algorithm. Picking the  $S_i$  randomly is computationally infeasible. The sets must be defined by some simple rule, since we do not want

to affect the time or memory requirements of the algorithm to store them explicitly. A pseudo-random function  $f : G \rightarrow \{1, 2, 3\}$  is a possibility. Some pathological partitions we likely want to avoid are, for example, letting  $S_1$  be a sequence of consecutive multiples of  $\alpha$ , or  $S_3$  a sequence of consecutive multiples of  $\beta$ . This problem could be overcome by randomising the choice of  $\alpha$  and  $\beta$ . More importantly  $S_2$  should not be a sequence of iterated squares. Randomising will not help this problem, but we could slightly modify the random walk so that  $x_{i+1} = \alpha x_i$  if  $x_i \in S_2$ , for example. Pathological partitions seem unlikely, if for example, using the lexicographic order on the binary representation of group elements, we let  $S_1$  be the first third,  $S_2$  be the second third, and  $S_3$  be the rest. This is what Pollard does in his example with  $\mathbf{GF}(p)^*$ .

Algorithm 2.2.2 describes a way to get a heuristic probabilistic algorithm for finding  $\log_\alpha(\beta)$  with expected running time  $O(\sqrt{N})$ . Let  $A$  be the set of generators of  $G$ . Assume there exists some constant  $c_1 > 0$ , such that for our "reasonable" choices of  $S_i$ , there exists a subset of  $A \times G$  of size at least  $c_1|A||G|$ , such that the expected impact for each  $(\alpha, \beta)$  in that subset is at most  $c_2\sqrt{N}$  for some constant  $c_2$ . Another assumption is that for the  $n$  in step 2 of Algorithm 2.2.1, the probability that  $\gcd(n, N) < N^\epsilon$  is at least  $1/2$  for some positive  $\epsilon < 1$ .

---

**Algorithm 2.2.2** Heuristic probabilistic version of Pollard's method

---

1. Randomise the input by selecting uniformly at random a positive integer  $r < N$ , coprime with  $N$  and a non-negative integer  $b < N$ , replace  $\alpha$  with  $\alpha^r$ , and replace  $\beta$  with  $\beta\alpha^b$ .
  2. Compute a sequence of 6-tuples,  $(x_i, a_i, b_i, x_{2i}, a_{2i}, b_{2i})$ ,  $i = 1, 2, 3, \dots$  until we get  $x_i = x_{2i}$  or until  $i = c_2\sqrt{N}$ . If we do not get a match, go back to step 1.
  3. Continue with steps 2 and 3 of Pollard's algorithm, Algorithm 2.2.1.
- 

Given  $\log_{\alpha^r}(\beta\alpha^b) = k \pmod{N/d}$ , we know  $\log_\alpha(\beta) = rk - b \pmod{N/d}$ . We expect to get a match in  $\frac{c_2}{c_1}\sqrt{N}$  steps. Since  $c_2$  and  $c_1$  are fixed, the expected running time is  $O(\sqrt{N})$  for each collision, and now we can apply the earlier analysis to get a heuristic probabilistic algorithm with running time  $O(\sqrt{N})$  group multiplications.

Pollard also outlines a  $\lambda$ -method, which he describes as a method for catching kangaroos. It is useful when the logarithm is known to lie in a specific interval of an arithmetic sequence. Heuristic arguments suggest it should usually work in time  $O(\sqrt{w})$  where the interval contains  $w$  elements, for most elements. Some randomisation could change it into a heuristic probabilistic algorithm.

## 2.3 Subgroup technique

Let  $N = p_1 p_2 p_3 \dots p_t$ . The integers  $p_i$  are not necessarily distinct or prime. Suppose  $k = \log_\alpha(\beta)$ . We know

$$k = b_1 + b_2(p_1) + b_3(p_2 p_1) + \dots + b_{t-1}(p_{t-2} p_{t-3} \dots p_1) + b_t(p_{t-1} p_{t-2} \dots p_1)$$

for unique integers  $b_i$  satisfying  $0 \leq b_i < p_i$ . Let  $\alpha_i = \alpha^{N/p_i}$ . Let

$$\beta_i = \alpha^{b_i(p_{i+1} p_{i+2} \dots p_t) + \dots + b_2(p_3 p_4 \dots p_t) + b_1(p_2 p_3 \dots p_t)}.$$

Note that

$$\beta_i^{p_1 p_2 \dots p_{i-1}} = \alpha^{b_i N / p_i} = \alpha_i^{b_i}.$$

This explains step 2 of Algorithm 2.3.1. We have  $\beta_1 = \beta$  and for  $i > 1$  we can compute  $\beta_i$  from  $\beta_{i-1}$  once we know  $b_{i-1}$ .

---

**Algorithm 2.3.1** Subgroup technique

---

1. Set  $\beta_1 = \beta$ ,  $\alpha_1 = \alpha^{N/p_1}$ ,  $i = 1$ .
  2. Compute  $b_i = \log_{\alpha_i}(\beta_i^{p_1 p_2 \dots p_{i-1}})$ .
  3. If  $i = t$ , go to step 4. Otherwise, set  $\beta_{i+1} = \beta_i \alpha^{-b_i (p_{i+1} \dots p_t)}$ . Compute  $\alpha_{i+1} = \alpha^{N/p_{i+1}}$ . Increment  $i$  and go to step 2.
  4. The logarithm is  $k = b_1 + b_2 p_1 + b_3 (p_2 p_1) + \dots + b_{t-1} (p_{t-2} p_{t-3} \dots p_1) + b_t (p_{t-1} p_{t-2} \dots p_1)$ .
- 

The algorithm we describe here is a generalisation of what is commonly referred to as the Silver-Pohlig-Hellman algorithm. Pohlig and Hellman [PH78] restrict their description to prime power factorisations, solve modulo each prime power separately, and then combine the answers with the Chinese remainder theorem. Pollard [Pol78] briefly but undoubtedly describes the technique I give here, giving an example with  $N$  factored into any two numbers. According to Pollard [Pol78], the idea of using the factorisation of the group order is due to Silver. Pohlig and Hellman write that this idea was discovered independently by Silver, and later by Schroepel and H. Block.

Pohlig, Hellman [PH78], and Pollard [Pol78] describe the idea of solving the logarithm modulo factors of  $N$  using techniques that take time proportional to the square root of the factors. These algorithms were described for  $\mathbf{GF}(p)^*$  where  $p$  is a prime, but the techniques apply to any cyclic group. Thiong Ly [Thi93] gives a "serial version" of the Pohlig-Hellman algorithm, which is essentially what I presented here. Both [Thi93] and [PH78] specify using the baby-step giant-step algorithm and a prime factorisation.

One possible advantage of the Pohlig-Hellman description of this algorithm is that it is easily distributed to several processors, and the answer can be combined as the various partial results are computed. More generally, let  $N = \prod_{i=1}^r n_i$ , where the  $n_i$  are pairwise coprime. Then using Algorithm 2.3.1 with any factorisation of  $n_i$ , solve for  $l_i = \log_{\alpha^{N/n_i}}(\beta^{N/n_i})$ , for  $i = 1, 2, \dots, r$ , on  $r$  different processors.

If we use the baby-step giant-step algorithm as a subroutine, Algorithm 2.3.1 is deterministic with running time  $O(t \log(N) + \sum_{i=0}^t \log(n_i) \sqrt{n_i})$  group operations and requires  $O(\max\{\sqrt{n_i}\})$  group elements of space. If we use Pollard's algorithm as a subroutine, we get a heuristic expected running time of  $O(t \log(N) + \sum_{i=0}^t \sqrt{n_i})$  group operations and a  $O(1)$  space space requirement.

I made the following observation in the case of finite fields [Mos95] and state it here for a general group.

**Observation 1** *The discrete logarithm problem in a cyclic group of order  $N$  can be reduced to the discrete logarithm problem in a collection of subgroups with the property that each prime factor of  $N$  divides the order of at least one of the subgroups.*

If we do not have a factorisation of  $N$  into primes, we could replace the word *prime* with *effectively irreducible*. I refer to methods of this sort as *subgroup techniques*.

## 2.4 Subgroup techniques applied to finite fields

It was mentioned in [Odl85] that fields with large subfields should be avoided for use in cryptography because the added structure might be exploitable by an adversary. Cited as an example is that the field  $\mathbf{GF}(p^2)$  is about as safe as  $\mathbf{GF}(p)$  if  $p + 1$  is smooth, and thus one would be better off using  $\mathbf{GF}(q)$  for some prime  $q$  of size roughly  $p^2$ . I describe here how this is so, and generalise to any field with non-trivial subfields. The reason boils down to Observation 1. Note that if the bottleneck subgroup of  $\mathbf{GF}(q)^*$  is a subgroup of the multiplicative group of a subfield of  $\mathbf{GF}(q)$ , we only need to implement index calculus techniques in the subfield. This explains why  $GF(p^2)$  is about secure as  $GF(p)$  if  $p + 1$  is smooth.

The index calculus techniques which I describe in Chapter 3, are usually much faster than the square root techniques unless the order of the field is smooth. But for fields  $\mathbf{GF}(p^n)$  with subfields, it may be possible, though unlikely as  $n \rightarrow \infty$ , that the only *large* prime factors of  $p^n - 1$  are actually factors of  $p^a - 1$  for some  $a \mid n, a < N$ . In this case, we could apply the index calculus methods on some of the subfields, and find the logarithm modulo the remaining factors of  $p^n - 1$  by square root methods.

For example, for the field  $\mathbf{GF}(2^{186})$ , we notice that

$$2^{186} - 1 = 7 * 658812288653553079 * 2147483647 * 3^2 * 529510939 * 2903110321 * 715827883,$$

$$2^{93} - 1 = 7 * 658812288653553079 * 2147483647,$$

$$2^{62} - 1 = 3 * 2147483647 * 715827883,$$

$$2^{31} - 1 = 2147483647,$$

$$2^6 - 1 = 3^2 * 7,$$

$$2^3 - 1 = 7,$$

$$\text{and } 2^2 - 1 = 3.$$

The bottleneck in using the square root methods would be finding the logarithm modulo 658812288653553079 which would require approximately  $10^9$  field operations. We observe that the only factors which do not appear in the order of a proper subfield are 529510939 and 2903110321. The logarithm can be solved modulo these factors using square root methods in roughly  $10^5$  field operations. We can solve modulo 715827883 by square root methods or by implementing index calculus methods in  $\mathbf{GF}(2^{62})$ . We can solve modulo 2147483647 by square root methods or by index calculus methods in  $\mathbf{GF}(2^{31})$ , or we could do more linear algebra work with the  $\mathbf{GF}(2^{62})$  database, but it should be faster to work in  $\mathbf{GF}(2^{31})$ . We can solve modulo 7 and  $3^2$  using the subgroup methods with any algorithm as a subroutine. Lastly, to solve modulo 658812288653553079, we could apply Coppersmith's index calculus technique in the subfield  $\mathbf{GF}(2^{93})$ , which my calculations suggest is possible with less than  $10^5$  smoothness tests and a factor base of size roughly  $10^3$ .

If, on the other hand, we just apply Coppersmith's technique in  $\mathbf{GF}(2^{186})$ , my calculations suggest that it will require roughly  $10^7$  smoothness tests before we find enough equations for our database of size roughly  $10^4$ .

I now characterise precisely what number we can solve modulo, by solving the logarithm in *proper* subfields. The idea is then to decide if it is worthwhile to solve the logarithm with square root methods or index calculus techniques for the various factors of  $N = p^n - 1$ . We can solve the logarithm modulo  $p^a - 1$  for any  $a$  dividing  $n$  and less than  $n$ , by only working in  $GF(p^a)$ . Consider the polynomial  $x^n - 1$ . We know that  $x^n - 1 = \prod_{i=1}^n (x - \xi^i)$  for some primitive  $n$ th root of unity,  $\xi$ . The  $n$ th cyclotomic polynomial is

$$\Phi_n(x) = \prod_{(i,n)=1, 0 \leq i < n} (x - \xi^i).$$

For  $a|n$ , we have

$$x^a - 1 = \prod_{i=0}^{a-1} (x - \xi^{\frac{in}{a}}).$$

So  $x^a - 1$  divides  $(x^n - 1)/\Phi_n(x)$  for all positive  $a < n$ ,  $a|n$ . In fact,

$$\text{lcm}_{a|n, a < n} (x^a - 1) = (x^n - 1)/\Phi_n(x).$$

Thus any prime factor of  $(p^n - 1)/\Phi_n(p)$  must divide  $p^a - 1$  for some proper divisor,  $a$ , of  $n$ . We can solve for the logarithm modulo those primes by only working in proper subfields of  $\mathbf{GF}(q)$ .

**Definition 4** Let  $A(n, p)$  be the largest factor of  $\Phi_n(p)$  which is coprime to  $(p^n - 1)/\Phi_n(p)$ .

**Observation 2** Solving for logarithms in  $\mathbf{GF}(q)^* = \langle \alpha \rangle$  can be reduced to finding logarithms in its cyclic subgroup of order  $A(n, p)$  and in the multiplicative groups of its subfields.

To solve logarithms in  $\langle \alpha^{N/A(n,p)} \rangle$ , we could implement index calculus methods, but only do the linear algebra modulo the known factors of  $A(n, p)$ , or we could apply square root methods in its known subgroups.

I made these observations during my research at the University of Waterloo in the summer of 1995.

## Chapter 3

# Index Calculus Techniques

The techniques of Chapter 2 do not exploit any additional structure that might exist in the finite group. Index Calculus methods can be advantageous when we have relatively fast methods for representing group elements as products of elements from a small subset of the group. In the first section, I describe the index calculus technique in general. In the second section, I discuss the linear algebra phase of the algorithm. In the third section, I give details of a rigorous implementation for fields of the form  $\mathbf{GF}(p^n)$ , represented as polynomials over  $\mathbf{GF}(p)$  modulo an irreducible of degree  $n$ . Such fields with characteristic 2 are used in the applications mentioned in [CW94] and [LB92]. Some cryptosystems use these fields because  $\mathbf{GF}(2)$  and its extensions are very convenient to work with on binary computers. I also describe modifications to the rigorous method that makes it more practical for implementation in large fields, and suggest some other techniques as well. The subsequent sections overview other implementations of index calculus methods for the discrete logarithm problem in finite fields.

We can easily identify an  $n$ -tuple over a ring  $R$  with either an  $n \times 1$  or a  $1 \times n$  matrix over  $R$ . I do this implicitly throughout this chapter.

Detailed surveys of discrete logarithm methods for finite fields include [Odl85], [McC90], [Odl94], and [SWD96].

### 3.1 Basic Method

The first task is to identify a subset  $S$  of the group  $G$  which will be used as a *factor base*. Once this set has been picked, the algorithm has three phases which are described in Algorithm 3.1.1. Let  $N$  be the order of the group. This algorithm succeeds with probability at least  $1/2$  on every input, and thus can be repeated to yield a probabilistic algorithm.

The larger we select the factor base  $S$ , the easier it is to obtain these relations, however we will consequently need more of them for Phase 3 to work with probability  $\geq 1/2$ . We must balance these two effects when selecting  $S$ . Note that this method does not use the fact that  $\alpha$  is a generator. Full rank is not necessary. We are only interested in the tuple for  $\beta$  lying in the column span. Suppose the tuples generated in Phase 1 and in Phase 3 are all generated independently at random with the identical distribution. By  $\Omega(N)$  we denote the number of prime divisors of  $N$ ,

---

**Algorithm 3.1.1** Basic INDEX CALCULUS algorithm

---

**Phase 1: Equation Generation**

Search for multiplicative relations between elements of  $S$  and  $\alpha$ . These provide linear relations modulo  $N$  between the logarithms of elements of  $S = \{s_i\}$ . For example, if we find that  $s_1^{a_1} s_2^{a_2} s_3^{a_3} = s_4^{a_4} s_5^{a_5} \alpha^y$ , then it follows that

$$a_1 \log_\alpha(s_1) + a_2 \log_\alpha(s_2) + a_3 \log_\alpha(s_3) \equiv a_4 \log_\alpha(s_4) + a_5 \log_\alpha(s_5) + y.$$

Let  $k = |S|$ . From the  $j$ th relation, extract a  $k$ -tuple  $(a_{j,1}, a_{j,2}, \dots, a_{j,k})$  and a coefficient  $y_j$  satisfying  $\prod_{i=1}^k s_i^{a_{j,i}} = \alpha^{y_j}$ .

Create a matrix  $A$  whose columns are the  $k$ -tuples of integers  $\mathbf{a}_j$ , also referred to as the database of relations.

Let  $\mathbf{a}$  be the  $k$ -tuple with  $a_i = \log_\alpha s_i$ , which therefore satisfies  $\mathbf{a}A = \mathbf{y} \pmod N$ .

Find enough relations so that the  $k$ -tuple we find in Phase 2 has probability at least  $1/2$  of lying in the space spanned by the columns of the database.

**Phase 2: Represent  $\beta$  as a tuple**

Represent a given element  $\beta$  as a product of integer powers of elements of  $S$  and  $\alpha$ . Given  $\beta = \alpha^k \prod s_i^{b_i}$  it follows that  $\log_\alpha(\beta) = \sum b_i \log_\alpha(s_i) + k = (\mathbf{a} \cdot \mathbf{b} + k) \pmod N$ .

**Phase 3: Represent  $\mathbf{b}$  as a Linear Combination of Database Relations**

It may not be possible to solve  $\mathbf{a}A = \mathbf{y}$  for  $\mathbf{a}$  uniquely, but this is not necessary if we only want  $\mathbf{a} \cdot \mathbf{b}$ .

Solve the system  $Az = \mathbf{b}$ . Given a solution  $\mathbf{z}$ , it follows that  $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}(Az) = (\mathbf{a}A)\mathbf{z} = \mathbf{y} \cdot \mathbf{z}$ . Since we know  $\mathbf{y}$ , we can find  $\mathbf{a} \cdot \mathbf{b}$ , which gives us the answer we seek.

---

counted with multiplicity. Lovorn-Bender and Pomerance [BP96] show that if we have  $2\Omega(N)|S|$  relations from Phase 1, then the probability that a new  $k$ -tuple (namely  $\mathbf{b}$ ) lies in the span of the existing ones is greater than  $1/2$ . This result was modelled after a result in [Lov92]. Hellman and Reyneri [HR83] have a similar result using the Chebyshev inequality (see appendix for a correction).

I have not yet said anything about how to obtain these relations. Schnorr [Sch93], for example, in the case of  $\mathbf{GF}(p)$ , where  $p$  is prime, suggests a method using Diophantine approximation which reduces to finding low-weight elements of a lattice. He solves the logarithm problem modulo a prime of size roughly  $2^{40}$ . The lattice basis reduction problem to which he reduces the DLP for a prime of size roughly  $2^{512}$  seems to be intractable with currently known lattice basis reduction algorithms. It is not clear to me how to select parameters optimally, or how this technique compares to other algorithms.

A common thread between what are called *Index Calculus* algorithms is the idea of a factor base, between whose elements we find multiplicative relations. We then perform linear algebra operations on the tuples corresponding to the exponents of the factor base elements in these multiplicative relations and derive useful information.

Such use of a factor base seems to date back to the work of Kraitchik (see [Pom85]). The application to computing logarithms is attributed to A. E. Western by J. C. P. Miller in [Mil75], and is described in [WM68]. Its application to factoring is described in [Mil75]. In [Pol78], Pollard notes the use of the techniques in [WM68] by Miller for factoring in [Mil75], and describes how they would be used for computing discrete logarithms in  $\mathbf{GF}(p)$ . Odlyzko [Odl85] and McCurley [McC90] note that this algorithm was also rediscovered by Adleman and Merkle.

## 3.2 Linear Algebra

Gaussian elimination, with some minor modifications if  $N$  is not prime (see [Lov92] or [McC90] for example), solves Phase 3 in  $O(M^3)$  operations in the integers modulo  $N$  where  $M$  is the larger dimension of the system. These linear systems are usually sparse, that is, most coefficients are 0. In the discrete logarithm algorithms I describe, the total number of non-zero entries is  $M^{1+o(1)}$ . If the entries are elements of a finite field, other methods, both heuristic and rigorous, are known which require expected time  $M^{2+o(1)}$ , as  $M \rightarrow \infty$  [Wie86, KS91, LO91b]. When we try to implement algorithms with running times of  $M^{2+o(1)}$ , the problems associated with singular or non-square systems, and with working modulo composites become less trivial to overcome, at least rigorously.

We can reduce the matrix solution step to solving the system modulo each prime power factor of  $N$ , or any set of relatively prime factors of  $N$ , and then applying the Chinese Remainder Theorem. However, we might not have the factorisation of  $N$ , and it would be nice not to make this step depend on factoring  $N$  completely. I follow the suggestion in [COS86], [Pom87] and later in [Gor93], where we have to solve a system modulo  $N$  without knowing the prime factorisation of  $N$ . We can efficiently test to see if  $N$  is a perfect power. By *perfect power* I mean  $n^a$  for some integers  $n, a > 1$ . We can then apply Hensel lifting (see appendix A.2) if the system is non-singular modulo  $n$ . For the case that  $N$  is not a perfect power, the idea is to proceed modulo  $N$  as if  $N$  was prime, using the Extended Euclidean Algorithm to compute inverses whenever

necessary. The claim is that this procedure will either produce a solution modulo  $N$ , or will stop because an inverse could not be computed. The latter case gives us a non-trivial factor of  $N$ , and we can continue modulo the new factors now known. The same idea can be applied with Gaussian elimination in the case of composite  $N$ , and it is likely that this can be done with other sparse matrix techniques.

For any such technique, it needs to be checked that with some expected running time, the technique will either succeed in solving the system, or fail but uncover a factor of  $N$ . It is conceivable that some algorithms might run much longer modulo composites, or might not uncover factors when they fail. We still need to deal with the case of singular systems. As I mention in section 3.2.1, I am not yet convinced that we have a rigorous method with running time  $M^{2+o(1)}$  for solving a system (with larger dimension equal to  $M$ ) modulo  $p^a$ ,  $a > 1$  if it does not have full rank considered modulo  $p$  for each prime,  $p$ , dividing  $n$ . In section 3.2.2 I note that this is not a problem for the index calculus algorithm I describe. The reason is that we can solve modulo a factor of  $N$  which has each distinct prime factor of  $N$  as a divisor.

### 3.2.1 Wiedemann's algorithm

I now discuss Wiedemann's techniques [Wie86] for solving sparse systems. These techniques only make use of the fact that multiplying an  $M \times M$  matrix by an  $M \times 1$  matrix can be done in fewer than  $O(M^2)$  coefficient operations in the case of sparse matrices. Other matrices, such as Toeplitz matrices, also have fast algorithms for multiplication (see appendix A.5).

Sparse matrices will be represented as a list of elements and their co-ordinates, so a sparse matrix with only  $\omega$  entries only requires space for  $\omega$  coefficients and coordinates. For our purposes, we will assume that  $\omega = M^{1+o(1)}$ , and thus the matrix multiplications can be done in  $M^{1+o(1)}$  coefficient operations. For example, to compute  $\mathbf{y} = \mathbf{A}\mathbf{x}$ , initialise  $\mathbf{y}$  to  $\mathbf{0}$ , and then go through the list of coefficients. For each  $(i, j, c)$ , with  $c$  being a coefficient with coordinates  $i$  and  $j$ , add  $cx_j$  to the  $i$ th entry of  $\mathbf{y}$ .

The entire algorithm has an expected running time of  $M^{2+o(1)}$  coefficient operations if the coefficient ring is a field. By the Chinese Remainder Theorem it suffices to solve a system modulo a set of relatively prime factors of  $N$ . However, we might not know the factorisation of  $N$ . Since we have algorithms, rigorous ones as well as heuristic and practical ones, for finding all the prime factors of  $N$  less than  $M^{4+o(1)}$  in time  $M^{2+o(1)}$ , we could assume that  $N$  is either a prime power, or composite with more than one distinct prime factor, each of size at least  $M^{4+o(1)}$ . To preserve rigour and the  $M^{1+o(1)}$  space requirement, we assume that  $N$  is either a prime power, or composite with more than one prime factor, each of size at least  $3M^2$ .

When we refer to the product of two sparse matrices, we do not actually multiply them together, since the product might no longer be sparse. We store both matrices in sparse format, and to compute  $\mathbf{A}\mathbf{B}\mathbf{x}$  we use the associativity of matrix multiplication and compute  $\mathbf{A}(\mathbf{B}\mathbf{x})$ .

I describe Wiedemann's *algorithm 1*, which is a probabilistic algorithm for solving  $\mathbf{A}\mathbf{x} = \mathbf{b}$  over a field. He also gives deterministic *algorithm 2*. His algorithm seeks a minimal polynomial,  $m(x)$ , of  $A_S$  where  $A_S$  is the linear transformation induced by  $A$  on the subspace spanned by  $\{\mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}^2\mathbf{b} \dots\}$ .

Let  $\mathbf{F}$  be a finite field. A sequence of elements  $r_0, r_1, \dots$  from a vector space over  $\mathbf{F}$  is said to obey a *linear recurrence of length  $n$*  if there exist  $c_0, c_1, \dots, c_{n-1} \in \mathbf{F}$ ,  $c_n = 1$ , with the property that  $c_0 r_j + c_1 r_{j+1} + \dots + c_n r_{j+n} = 0$ , for all  $j \geq 0$ . The minimal polynomial of a linearly recurrent sequence of elements is the monic polynomial  $\sum c_i x^i$  of minimum degree such that  $c_0 r_j + c_1 r_{j+1} + \dots + c_n r_{j+n} = 0$ , for all  $j \geq 0$ . Note that  $m(x)$ , the minimal polynomial of  $A_S$ , is also the minimal polynomial of  $\mathbf{b}, A\mathbf{b}, A^2\mathbf{b}, \dots$

Working over a field, Wiedemann makes clever use of the fact that the minimal polynomial for  $A_S$  is a multiple of the minimal polynomial of the sequence  $\mathbf{c} \cdot \mathbf{b}, \mathbf{c}A\mathbf{b}, \mathbf{c}A^2\mathbf{b}, \dots$ , where  $\mathbf{c}$  is any randomly chosen row tuple. The Berlekamp-Massey algorithm, well-known in coding theory, will recover the minimal polynomial of a sequence of degree  $\leq k$  in  $O(k^2)$  field operations given the first  $2k$  terms of the sequence. I break up the analysis into several cases.

### Non-singular, square, modulo $p$

With Algorithm 3.2.1, we address the case of  $A$  an  $M \times M$  matrix, non-singular modulo the prime  $p$ , and of solving  $A\mathbf{x} = \mathbf{b}$  modulo  $p$ .

---

#### Algorithm 3.2.1 Wiedemann's probabilistic algorithm 1

---

1. Set  $\mathbf{b}_1 = \mathbf{b}$ ,  $i = 1$ .
  2. Select a random  $1 \times M$  matrix  $\mathbf{c}$ , and use the Berlekamp-Massey algorithm to recover the minimal polynomial,  $f_i(x)$ , of the sequence  $\mathbf{c} \cdot \mathbf{b}_i, \mathbf{c}A\mathbf{b}_i, \mathbf{c}A^2\mathbf{b}_i, \dots$
  3. Apply  $f_i(A)$  to  $\mathbf{b}_i$  to produce  $\mathbf{b}_{i+1} = f_i(A)\mathbf{b}_i$ .
  4. If  $\mathbf{b}_{i+1} = \mathbf{0}$ , stop. Otherwise, increment  $i$  and go to step 2.
- 

Note that the minimal polynomial of  $\mathbf{b}_i$  is  $m(x)/\prod_{j < i} f_j(x)$  and  $m(x) = \prod f_i(x)$ . The expected number of repetitions with random choices of  $\mathbf{c}$  is  $O(1)$  [Wie86, KS91]. If  $A$  is non-singular then so is  $A_S$  and the constant term of its minimal polynomial will be invertible. We can thus solve for  $\mathbf{x}$  by noting that if  $c_0\mathbf{b} + c_1A\mathbf{b} + c_2A^2\mathbf{b} + \dots + c_nA^n\mathbf{b} = \mathbf{0}$ , then  $\mathbf{x} = c_0^{-1}(c_1\mathbf{b} + c_2A\mathbf{b} + \dots + c_nA^{n-1}\mathbf{b})$ , satisfies  $A\mathbf{x} = \mathbf{b}$ .

Algorithm 3.2.2 follows the description of the Berlekamp-Massey algorithm in [LN86]. Given a sequence  $s_0, s_1, \dots, s_{2k-1}$ , with minimal polynomial of degree  $k$ , let  $G(x) = \sum_{i=0}^{2k-1} s_i x^i$ . The algorithm involves recursively computing two polynomials,  $g_i(x)$  and  $h_i(x)$ . After the  $2k$  steps, the minimal polynomial is derived from the reciprocal of  $g_{2k}(x)$ . A proof that this polynomial is actually the minimal polynomial can be found in [LN86].

These methods will work to solve a non-singular system modulo a prime.

### Non-singular, square, modulo $n^a$ , $a > 1$

If the system is non-singular modulo  $n$ , we can find a solution modulo  $n$ , then lift it to a solution modulo  $n^a$ , by a procedure that is referred to as Hensel lifting (see appendix A.2). We thus reduce this case to the following situation.

### Non-singular, square, modulo $N$ , $N$ not a perfect power

Let  $N = \prod p_j^{a_j}$  be the prime power factorisation of  $N$ ,  $N$  not a perfect power. To solve the system modulo  $N$  it suffices to solve it modulo  $p_j^{a_j}$  for each  $p_j$  and apply the Chinese Remainder

---

**Algorithm 3.2.2** Berlekamp-Massey Algorithm

---

1. Initialise  $g_0(x) = 1, h_0(x) = x$ , and  $m_0 = 0$ .
2. Let  $b_i$  be the coefficient of  $x^i$  in  $g_i(x)G(x)$ . (Note that this corresponds to  $\sum_{j=0}^{d_i} g_{i,j}s_{i-j}$ , where  $g_i(x) = \sum_{j=0}^{d_i} g_{i,j}x^j, d_i \leq i$ .)
3. We then recursively compute, for  $i$  from 1 to  $2k - 1$ :
4.  $g_{i+1}(x) = g_i(x) - b_i h_i(x)$ , and

$$h_{i+1} = \begin{cases} b_i^{-1}x g_i(x) & \text{if } b_i \neq 0 \text{ and } m_i \geq 0, \\ \text{or } x h_i(x) & \text{otherwise,} \end{cases}$$

$$u_{i+1} = \begin{cases} -u_i & \text{if } b_i \neq 0 \text{ and } u_i \geq 0, \\ \text{or } u_i + 1 & \text{otherwise.} \end{cases}$$

5. Set  $r = \lfloor k + \frac{1}{2} - \frac{u_{2k}}{2} \rfloor$ .
  6. The minimal polynomial of the sequence is  $x^r g_{2k}(1/x)$ .
- 

Theorem. However, we might not have the full factorisation of  $N$ . I claim that if we follow the above algorithm modulo  $N$ , either nothing will go wrong and we find a solution modulo  $N$  in the  $2k$  steps, or we will uncover a non-trivial factor of  $N$ .

If we run the Berlekamp-Massey algorithm on a sequence of integers modulo  $N$ , one of two things will happen. Either  $\gcd(b_i, N) = 1$  or  $N$ , for all  $i$ , in which case every step is carried out just as it would had we been working modulo  $p_j$  for each prime factor  $p_j$  of  $N$ . That means none of the  $u_i$  change, the sequence of updates for  $h_i(x)$  do not change, and  $r$  remains the same. In this case, we will eventually obtain a polynomial which, considered modulo  $p_j$  is the minimal polynomial of the sequence considered modulo  $p_j$ . Suppose that after  $a$  iterations of the Berlekamp-Massey algorithm we obtain the minimal polynomial of the sequence modulo  $p_j$ . Then  $\mathbf{b}_{a+1} \equiv \mathbf{0} \pmod{p_j}$ . If we continued, for any choice of  $c$ , the subsequent  $b_i$  values in the Berlekamp-Massey algorithm would all be  $0 \pmod{p_j}$  and thus by the assumption that  $\gcd(b_i, N) = 1$  or  $N$ , we have that  $b_i = 0 \pmod{N}$ . This means we must have  $\mathbf{b}_{a+1} = \mathbf{0} \pmod{N}$ . We can now solve  $A\mathbf{x} = \mathbf{b}$ .

The other possibility is that during one of the runs through the Berlekamp-Massey algorithm  $\gcd(b_i, N)$  is neither 1 nor  $N$  for some  $i$ . We can then obtain a non-trivial factorisation of  $N$  into coprime factors, and we can continue with the Berlekamp-Massey algorithm and Wiedemann's algorithm modulo these new factors. We can efficiently test to see if any of these factors are perfect powers.

**Non-square or singular, modulo  $p$** 

If  $A$  is non-square or singular, these techniques will find linear dependencies between rows and columns, and thus we can use these methods to reduce a non-square singular matrix to a square non-singular one [Lov92]. However, if the number of rows and columns that must be eliminated is not  $M^{o(1)}$ , this will not yield a probabilistic  $M^{2+o(1)}$  algorithm. In particular, if the number

of rows and columns to be removed is  $M^{1+o(1)}$ , as will be the case with the rigorous versions of the index calculus method we present, this is comparable to Gaussian elimination, and inferior to other known non-sparse matrix techniques.

Wiedemann describes several probabilistic methods that will reduce this problem to one of solving a square non-singular system, which can then be done in time  $M^{2+o(1)}$  field operations. These reductions require  $M^{o(1)}$  field operations.

I describe the method that will work even in the worst case, which is when  $A$  does not have full rank. Suppose the rank of  $A$  is  $r$ , and suppose there does exist a solution to  $Ax = \mathbf{b}$ . Wiedemann's method is to select sparse matrices  $P$  and  $Q$  such that  $PAQ$  is an  $r \times r$  non-singular matrix. We solve  $PAQy = P\mathbf{b}$  modulo  $p$  for the unique solution. Let  $\mathbf{x} = Qy$ . We then know that  $P(Ax - \mathbf{b}) = \mathbf{0}$ . It might not be clear why  $Ax - \mathbf{b}$  equals  $\mathbf{0}$  and not any other element of the kernel of  $P$ . The reason is that  $AQ$  has the same column span as  $A$ , thus  $Ax = \mathbf{b}$  has a solution if and only if  $AQy = \mathbf{b}$  has a solution. We assumed that the former had a solution, thus we know that that latter does. This solution must be the solution that we find to  $PAQy = P\mathbf{b}$  modulo  $p$  since  $PAQ$  is non-singular. Wiedemann presents a probabilistic algorithm for finding such sparse matrices  $P$  and  $Q$ .

Kaltofen and Saunders [KS91] describe a technique in which  $P$  and  $Q$  are upper and lower triangular unit Toeplitz matrices (see appendix A.5) whose entries are selected uniformly at random from the coefficient field. Note that Toeplitz matrices are not sparse, but the important point is that multiplying by them can be done in time  $M^{1+o(1)}$  where  $M$  is the dimension of the matrix. This is done by Fast Fourier Transform methods for polynomials. The first  $r$  leading principal minors of  $PAQ$  will be non-zero with probability

$$1 - \frac{r(r+1)}{p},$$

where  $r$  is the rank, and we select entries for our Toeplitz matrix uniformly at random from the integers modulo  $p$ . The original system can now be solved by working with the leading  $r \times r$  submatrix of  $PAQ$ . Both these techniques require knowing the rank of  $A$ . Wiedemann describes a binary search method for finding the rank, and refers to [RMK<sup>+</sup>80] for methods of coping with errors in binary searches. I have not analysed this method. Kaltofen and Saunders [KS91] describe a method involving multiplication by a random diagonal matrix,  $X$ , and finding the minimal polynomial. The rank of  $A$  modulo  $p$  will be  $\deg(f^{AX} \bmod p) - 1$  with probability at least  $1 - \frac{M(M-1)}{2p}$ . By  $f^{AX} \bmod p$  I mean the minimal polynomial modulo  $p$  of  $AX \bmod p$ . For both of these algorithms to work, it is required that the size of the coefficient field is large. If  $p$  is not big enough, Kaltofen and Saunders [KS91] and Wiedemann [Wie86] suggest embedding the coefficients in a field extension.

#### **Non-square, linearly independent columns modulo $p$ , modulo $p^a$**

In this case, we can apply the same techniques as above, except we do not need to find the rank and we can take  $Q$  to be the identity matrix. Again, we assume that  $Ax = \mathbf{b}$  has a solution. We solve  $PAx = P\mathbf{b} \bmod p^a$  by Hensel lifting. Since we can only get one such solution for a square non-singular  $PA$ , this must be the solution to  $Ax = \mathbf{b}$  if it exists.

#### **Non-square, linearly independent columns modulo $n$ , modulo $n^a$ , $a \geq 0$**

Here we assume  $n$  is composite with at least two distinct prime factors, and each of its prime factors is greater than  $3M^2$ . In this case, we can apply the same techniques as above, without

finding the rank and we can take  $Q$  to be the identity matrix. We assume that  $Ax = \mathbf{b}$  has a solution. We know that  $PA$  will be non-singular modulo  $p$  for some prime,  $p$ , dividing  $n$ , with probability at least  $1 - M(M + 1)/p > 1/2$ . Assume  $PA$  is non-singular modulo  $n$ , and attempt to solve  $PAx = P\mathbf{b}$  modulo  $n$  by Hensel lifting. With probability at least  $1/2$  there will be a solution modulo  $p^a$ , and when we apply Wiedemann's algorithm modulo  $n$ , we will either get a solution modulo  $n^a$  or a non-trivial factorisation of  $n$ . If we do get a solution modulo  $n^a$ , it must be the solution to  $Ax = \mathbf{b}$  since we can only get one such solution for a square non-singular  $PA$ .

**Non-square, rows linearly independent modulo  $p$ , modulo  $p^a$**

Again, we can apply the same techniques as above, take  $P$  to be the identity matrix, and assume  $AQ$  is non-singular modulo  $p$ . We solve  $AQ\mathbf{y} = \mathbf{b} \pmod{p^a}$  by Hensel lifting. We will be able to find a solution since  $AQ$  has full span modulo  $p$ , and  $\mathbf{x} = Q\mathbf{y} \pmod{p^a}$  will give us the answer we seek.

**Non-square, rows linearly independent modulo  $n$ , modulo  $n^a$ ,  $a \geq 0$**

As in the case with linearly independent columns, with probability at least  $1/2$  there will be a solution modulo  $p^a$  for some prime factor,  $p$ , of  $n$ , and while applying Wiedemann's algorithm we will either find a solution modulo  $n^a$ , or a non-trivial factorisation of  $n$ .

**Not full rank, modulo  $n^a$**

**We have a problem** if we apply the method described in the previous paragraphs since we are no longer guaranteed that  $\mathbf{x} = Q\mathbf{y}$  must be a solution to  $A\mathbf{x} = \mathbf{b}$ .

In the case of solving modulo a prime  $p$  we argued that  $AQ$  had the same column span as  $A$ , and thus  $A\mathbf{x} = \mathbf{b}$  has a solution if and only if  $AQ\mathbf{y} = \mathbf{b}$  has a solution. But now  $AQ$  would have the same column span as  $A$  only modulo  $n$ . The span modulo  $n^a$  might be diminished. We still have that  $A\mathbf{x} = \mathbf{b}$  has a solution if  $AQ\mathbf{y} = \mathbf{b}$  has a solution, but it was the *only if* part that we needed to conclude that the solution we find to  $PAQ\mathbf{y} = P\mathbf{b}$  actually gave us a solution to  $AQ\mathbf{y} = \mathbf{b}$ .

For example, consider modulo  $p^a$ ,  $a > 1$ ,

$$A = \begin{bmatrix} 1 & 1 \\ 0 & p \end{bmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 1 \\ p \end{pmatrix}.$$

Here,  $A$  has rank 1 modulo  $p$ . Suppose

$$P = [ 1 \quad 1 ]$$

and

$$Q = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Let  $\mathbf{y} = (a)$  be the solution we obtain by solving  $PAQ\mathbf{y} = P\mathbf{b}$ . We then get

$$\mathbf{x} = Q\mathbf{y} = \begin{pmatrix} a \\ a \end{pmatrix}.$$

Consequently,

$$Ax = \begin{pmatrix} 2a \\ pa \end{pmatrix}$$

which cannot be congruent to  $b$ , since this would require that  $a \equiv 1 \pmod{p}$ , which produces the contradiction  $2 \equiv 1 \pmod{p}$ . The solution we get is correct if we consider the system modulo  $p$ , but it is not one of the solutions that lift to a solution modulo  $p^2$ .

I am currently looking at ways of overcoming this problem, although I have managed to show (section 3.2.2) that it is not necessary for the purposes of finding discrete logarithms. I suspect there is a solution involving the Reeds and Sloane [RS85] version of the Berlekamp-Massey algorithm modulo prime powers, but I still have to work through the details.

### **Not full rank, modulo $N$ , $N$ not a perfect power**

We can apply Kaltofen and Saunderson's rank-finding algorithm. The method is to multiply  $A$  by a random diagonal matrix, and find the minimal polynomial. By random we mean that the entries were picked independently and uniformly at random from the integers modulo  $p$ . In the case of composite  $N$ , we pick the entries independently and uniformly at random from the integers modulo  $N$ . Note that modulo  $p_i$ , for  $p_i|N$ , this also corresponds to picking uniformly at random from the integers modulo  $p_i$ . By the Chinese remainder theorem, we know that this corresponds to picking them uniformly at random for all the  $p_i$  independently. The rank of  $A$  modulo  $p_i$  will be  $\deg(f^{AX} \pmod{p_i}) - 1$  with probability at least  $1 - \frac{M(M-1)}{2p_i}$  which is at least  $5/6$  in our case.

We thus assume that we have the correct rank for at least one of the primes, say  $p_1$ . Proceed as in the *non-square or singular modulo  $p$*  case, and select random upper and lower triangular Toeplitz matrices  $P$  and  $Q$  (selecting random entries mod  $N$ ) such that the leading  $r \times r$  submatrix is non-singular modulo  $p_1$  with probability at least  $1 - \frac{r(r+1)}{p_1}$ . When we try to solve the system corresponding to the leading  $r \times r$  submatrix modulo  $N$ , by an analysis similar to the *non-singular modulo  $N$*  case, we will either get a solution modulo  $N$  or uncover a non-trivial factor of  $N$ .

The above techniques apply to any linear system in the integers modulo  $N$ , with maximum dimension  $M$ . With an expected  $\Omega(N)M^{2+o(1)}$  operations with the integers modulo  $N$  or modulo factors of  $N$ , the techniques will produce a solution modulo a collection of factors of  $N$  whose product contains each distinct prime factor of  $N$  as a factor. I am working on a rigorous technique to solve the system modulo the perfect powers.

### **3.2.2 Linear Algebra Issues for Index Calculus**

Assume that we have the database produced by Phase 1. With an expected  $M^{2+o(1)}$  operations as  $M \rightarrow \infty$ , we can find the factors of  $N$  less than  $M^{4+o(1)}$  and solve for the logarithms modulo those factors using the subgroup techniques and square root algorithms of chapter 2. This will be a faster method of finding logarithms for factors much less than  $M^{4+o(1)}$ . If we want rigour, however, and we want to maintain a  $M^{1+o(1)}$  space requirement, we could use the baby-step giant-step algorithm and solve for the logarithms modulo factors of size up to  $M^{2+o(1)}$ .

In the previous section I did not claim to have an algorithm which solved a linear system with maximum dimension  $M$  in time  $M^{2+o(1)}$  when working modulo a perfect power,  $n^a$  if the system did not have full rank modulo  $n$ .

However, I make the following observation. *It is not necessary, for the rigorous index calculus algorithm I describe (section 3.3.1), to solve the linear system modulo  $n^a$ .*

It suffices to find a solution to the linear system modulo  $n$ , even if it does not lift to a full solution modulo  $n^a$ . In fact, it is not necessary for it to even have a solution modulo  $n^a$ . The solution modulo  $n$  will give us the logarithm modulo  $n$ . We can then lift this solution for the logarithm to a solution modulo  $n^a$ . This is similar to the subgroup technique of section 2.3. Algorithm 3.2.3 is an example with  $a = 2$ . Generalising this technique will give an algorithm for lifting our solution for the logarithm to a solution modulo  $n^a$ .

---

**Algorithm 3.2.3** Solving modulo  $n^2$

---

1. Find a tuple for  $\beta$ , say  $\mathbf{b}$ .
  2. Solve the system  $A\mathbf{x} = \mathbf{b}$  modulo  $n$ . This gives an integer  $k_0$ , such that  $\beta = \alpha^{k_0 + nk_1}$  for some integer  $k_1$ .
  3. Compute  $\beta' = (\beta\alpha^{-k_0})^{N/n} = \alpha^{k_1}$ .
  4. Find a new tuple for  $\beta'$ , say  $\mathbf{b}'$ .
  5. Solve the system  $A\mathbf{x} = \mathbf{b}'$  modulo  $n$ . From this solution we find the integer  $k_1$ .
- 

Another option for the rigorous analyses, is what Pomerance does in [Pom87], which is to guarantee full row rank with a high probability. In that case, we can solve linear systems modulo perfect powers in expected time  $M^{2+o(1)}$ .

If we assume that the system will have full row rank, there is an alternate formulation of index calculus Algorithm 3.1.1. Many descriptions [BFHMOV84, Cop84, Od185] of the index calculus method describe the last two phases slightly differently. Phase 2 becomes the solution of the system for  $\mathbf{a}$ . Then Phase 3 entails finding  $\mathbf{b}$  and computing  $\mathbf{b} \cdot \mathbf{a}$  to find the logarithm. An advantage of this modified method is that once the first two phases are completed, finding the logarithm of any new element only requires finding a new relation, which in practice is usually much faster than the other two phases. Also, we no longer require the relation database. With the algorithm described in section 3.3, the linear algebra must be repeated each time. For applications where the postcomputation of logarithms is most important, we can apply Phase 3 to solve for the logarithms of the database elements, and thus produce the tuple  $\mathbf{a}$ , which is a database of logarithms for factor base elements. We can also use the database of logarithms to find logarithms for a larger factor base, that is, the factor base can be used to enlarge itself.

### 3.3 Polynomial ring techniques

Lovorn-Bender and Pomerance [BP96] describe rigorous discrete logarithm computations in finite fields  $\mathbf{GF}(q)$ , where  $q = p^n$ , represented as polynomials modulo an irreducible of degree  $n$ . I make some minor modifications in light of the potential problem with incomplete rank and perfect powers. Algorithm 3.3.1 is a subroutine which, given an integer  $N$  dividing the order of the field, will with probability at least  $1/2$  return a solution modulo a factor of  $N$  which is divisible by each

distinct prime factor of  $N$ , and perhaps modulo higher powers of primes dividing  $N$ . Thus by the techniques of section 3.2.1 and using Algorithm 3.2.3 if necessary, we can solve the logarithm modulo  $Q = \prod p_i^{a_i}$ , where  $Q|q-1$ , after at most  $\max\{a_i\}$  successful applications of Algorithm 3.3.1 with various factors of  $Q$  and elements  $\beta'$ .

One fact used in [BP96] is that the logarithm of elements in  $\mathbf{GF}(p)$  are multiples of  $\frac{q-1}{p-1}$ , and thus we can ignore scalars if we do our linear algebra modulo  $\frac{q-1}{p-1}$ . We can easily lift our solution to the solution modulo  $q-1$  by solving a logarithm in  $\mathbf{GF}(p)$ .

In fact, as noted in section 2.4, we could solve the logarithm modulo a certain factor of  $\Phi_n(p)$  and reduce the logarithm problem to find logarithms in subfields of  $GF(p^n)$ .

Let  $\alpha(x)$  and  $\beta(x)$  be representatives of  $\alpha$  and  $\beta$ .

---

**Algorithm 3.3.1** Rigorous Index Calculus Algorithm for  $\mathbf{GF}(p)[x]/(f(x))$

---

**Step 0**

Let  $S$  be the set of monic irreducibles (not including 1) of degree at most  $m$  for some smoothness bound,  $m$ . The choice of  $m$  depends on the size of  $p$  with respect to  $n$ . This is discussed later.

**Step 1**

To find relationships between the residues mod  $f(x)$ , select uniformly at random an integer  $y_j \in \{1, 2, 3, \dots, 2^n - 1\}$ , and compute  $\alpha(x)^{y_j} \bmod f(x)$ . This gives an element of  $\mathbf{GF}(q)^*$  with uniform probability distribution. If this residue is  $m$ -smooth, we get a linear relation. If we wish to solve modulo  $N$ , repeat this until we get  $2\Omega(N)|S|$  relations, yielding a  $|S| \times 2\Omega(N)|S|$  system  $A$ , and  $|S|$ -tuple  $\mathbf{y}$ . If we do not know  $\Omega(N)$ , we can use an upper bound for it, like  $\log_r(N)$  where we know  $N$  has no prime factor less than  $r$ .

**Step 2**

To find a relation for  $\beta(x)$ , select uniformly at random an integer  $b \in \{1, 2, 3, \dots, 2^n - 1\}$ , and compute  $\beta(x)\alpha(x)^b \bmod f(x)$ . This also gives an element of  $\mathbf{GF}(q)^*$  with uniform probability distribution, which is the same distribution as in Step 1, as required for the rigorous analysis to work. Repeat this step until the result is  $m$ -smooth. Let the tuple corresponding to  $\beta(x)$  be  $\mathbf{b}$  and the corresponding exponent of  $\alpha(x)$  be  $b$ .

**Step 3**

Try to solve  $A\mathbf{z} = \mathbf{b}$  modulo  $N$ . By the analysis described in section 3.1 this will have a solution with probability at least  $1/2$ . If it does, then  $\log_\alpha(\beta) = \mathbf{y} \cdot \mathbf{z} - b \bmod N$ . We will be able to solve this modulo a factor of  $N$  containing each of the prime factors of  $N$ .

---

**Definition 5** Let  $P_q(n, m)$  denote the probability that a monic polynomial over  $\mathbf{GF}(q)$  of degree exactly  $n$  chosen uniformly at random is  $m$ -smooth.

Note that  $P_q(n, m)$  is a lower bound for the probability of a monic polynomial over  $\mathbf{GF}(q)$  of degree at most  $n$  chosen uniformly at random being  $m$ -smooth. The actual probability is  $\frac{q-1}{q^{n+1}-1} \sum_{i=0}^n q^i P_q(i, m)$ . In practice, especially for small  $q$ , we should account for this difference when choosing the optimal  $m$ . In many of the heuristic algorithms, we often choose elements of size at most  $n$  with non-uniform probability distribution, with the assumption that the probability that the elements are  $m$ -smooth is still roughly  $P_q(n, m)$ .

The expected time spent on Step 1 and 2 is at most  $(2\Omega(N)|S| + 1)/P_q(n-1, m)$ , and the running time of Step 3 is  $(\Omega(N)|S|)^{2+o(1)}$ . We wish to minimise the sum of the running times.

**Definition 6** Let  $N_q(n, m)$  be the number of monic  $m$ -smooth polynomials of degree exactly  $n$  over  $\mathbf{GF}(q)$ .

Theorem 2.1 of [BP96] which follows from Theorem 1.4 of [Sou93] gives us

**Theorem 1** Let  $u = \frac{n}{m}$  and assume  $1 \leq m \leq n$ . Uniformly for all prime powers  $q \geq (n \log(n)^2)^{\frac{1}{m}}$ , we have  $N_q(n, m) = q^n / u^{(1+o(1))u}$  as  $m \rightarrow \infty$  and  $u \rightarrow \infty$ .

With smoothness bound  $m$  the expected running time as  $q \rightarrow \infty$  is bounded above by

$$\frac{\Omega(N)q^{(1+o(1))m}}{P_q(n-1, m)} + (\Omega(N)q^m)^{2(1+o(1))}.$$

Let  $r = n \log(n) / \log(p)$ . When  $r \rightarrow \infty$ , Lovorn-Bender and Pomerance select  $m = \lceil \sqrt{\frac{n \log \log(q)}{2 \log(q)}} \rceil$ , which by Theorem 1 yields an expected running time of  $e^{(\sqrt{2}+o(1))(\log(q) \log \log(q))^{1/2}}$  for the equation generation and linear algebra phases. Phase 2 only takes an expected time of  $e^{(\sqrt{1/2}+o(1))(\log(q) \log \log(q))^{1/2}}$ .

Theorem 2.2 of [BP96] states

**Theorem 2** Let  $u = \frac{n}{m}$ . For all prime powers  $q$  and all positive integers  $m$  and  $n$  with  $m \leq n^{\frac{1}{2}}$ , we have  $N_q(n, m) \geq \frac{q^n}{n^u}$ .

When  $r$  is bounded from above, they use Theorem 2 and choose  $m$  to be the positive integer satisfying  $m^2 - m < r < m^2 + m$ . If  $r \geq 1/2$ , the expected running time is bounded by  $e^{(2+o(1))(\log(q) \log \log(q))^{1/2}}$ . If  $r$  is less than  $1/2$  then we can only choose  $m = 1$ , and this restriction makes the running time estimate larger. If  $r < 1/2$  and bounded away from 0, the expected running time is bounded by  $e^{(\sqrt{2/r}+o(1))(\log(q) \log \log(q))^{1/2}}$ . Lastly, if  $r = o(1)$ , the total running time is  $e^{(2+o(1)) \log(q)/n}$ .

All of these running times are  $q^{o(1)}$  provided  $n \rightarrow \infty$ .

Factoring in the fact that this algorithm has to be repeated an expected  $O(\max\{a_i\})$  times does not affect these running time estimates. We are still left with the task of solving a logarithm modulo  $p$ , which can be done with the rigorous algorithm due to Pomerance [Pom87] in time  $e^{(\sqrt{2}+o(1))(\log(p) \log \log(p))^{1/2}}$  and thus does not affect the running time estimates.

### 3.3.1 Practical Improvements

#### Fewer relations

In practice, people [BFHMV84, LO91b, Cop84, SWD96] seem to use only a few more relations than there are elements in the database. Rigorous analysis of these more practical methods seems difficult. If the tuples generated for the database were selected uniformly at random from all possible tuples, it would be easy to prove that we had a high probability of finding a spanning set of relations. This is not the case, however. Our tuples are very sparse, which is important if we

want to do the linear algebra in time  $M^{2+o(1)}$ . Also, there are usually some coordinates with a relatively smaller probability of having a non-zero coefficient. Odlyzko [Odl85] shows that in the Coppersmith implementation, which we describe later, if only  $2|S|$  relations are generated, many of the rows in the matrix  $A$  will be all zero. In practice this is not a problem since we simply omit the variables that are never used. It is much more likely that the system has full row rank in this case. I still suspect, however, that some of the *lighter* rows, corresponding to factor base elements that are less likely to occur, might have a significant chance of being dependent upon each other. Suppose the rows are tuples selected at random from the module of tuples by randomly placing a non-zero coefficient in a coordinate with some non-zero probability, and otherwise setting it to 0. If the probability of each coordinate being non-zero is sufficiently low, we can expect many zero rows. We can efficiently remove these. But if we also expect roughly  $\sqrt{N}$  or more rows with only one non-zero coefficient, then there is a good chance that some of them will be dependent.

In [Odl85] and [LO91b] Odlyzko and LaMacchia recommend performing some *structured* or *intelligent Gaussian elimination* before applying the sparse matrix techniques. These techniques quickly reduce the dimensions of the matrix. I believe the column and row deletions of the structured Gaussian elimination will eliminate the problem of dependent rows in practice.

We thus find the logarithms of most elements in  $S$ , and the third phase requires representing  $\beta$  as a product of these elements. Usually the elements of the original  $S$  that tend not to occur in any of the database relations will tend not to occur in the relation for  $\beta$ .

### Increase likelihood of smoothness

One clear way to increase the speed of the index calculus technique is to find linear relations faster. To accomplish this, we could test polynomials with a higher chance of being smooth. Blake, Fuji-Hara, Mullin and Vanstone [BFHMV84] observed that any degree  $n$  polynomial over  $\mathbf{GF}(2)$  could be represented as  $a(x)b(x)^{-1} \bmod f(x)$  efficiently using the Extended Euclidean Algorithm, where  $a(x)$  and  $b(x)$  have degree about  $\frac{n}{2}$ . The probability that one degree  $n$  polynomial is  $k$ -smooth, for the  $k$  of interest, is roughly  $(\frac{k}{n})^{\frac{n}{k}}$ , whereas the probability that two independent polynomials of degree around  $\frac{n}{2}$  are simultaneously  $k$ -smooth is roughly  $(\frac{k}{n})^{\frac{n}{2k}}$ . Assuming the polynomials  $a(x)$  and  $b(x)$  have about the same probability of being simultaneously smooth as two independent polynomials of degree around  $\frac{n}{2}$ , we would expect a significant improvement. This sped up the index calculus technique enough that logarithms in  $\mathbf{GF}(2^{127})$  became tractable, rendering some existing cryptosystems vulnerable (Odlyzko [Odl85] mentions that Hewlett-Packard and MITRE had such systems). However, the estimate of the asymptotic running time is the same, the improvement hidden in the  $o(1)$  error term of the exponent in the running time estimate  $e^{(\sqrt{2}+o(1))(\log(q) \log \log(q))^{1/2}}$ .

Another idea the above group, the *Waterloo group*, used was to exploit the structure of the chosen polynomial  $f(x)$  to generate many relations systematically. They used  $f(x) = x^{127} + x + 1$ , which meant that  $x^{128} \equiv x^2 + x$ , and consequently for any  $v(x) \in S$ ,  $v(x)^{128} \equiv v(x^2 + x)$ . The left hand side is smooth, and the right hand side is much more likely to be smooth than a random polynomial of degree  $n$  or even  $\frac{n}{2}$ , and thus many equations were generated at a much faster rate. However, at most half of the necessary equations can be generated this way. Also note that this method applies equally well to Phase 2.

Other ways of speeding up equation generation in this polynomial representation are mentioned in [Odl85], and actually improve the constant in the exponent of the running time estimate.

Coppersmith discovered a method for systematically generating polynomial relations of the form  $w_1(x)^{2^k} \equiv w_2(x)$ , where  $w_1(x)$  and  $w_2(x)$  are of degree roughly  $n^{\frac{2}{3}}$ . The probability of random polynomials of this degree being smooth is much higher than for polynomials of degree near  $n$  or even  $n/2$ . Thus, provided the probabilities of smoothness for these polynomials are roughly the same as for random ones of the same degree, equations can be generated much more quickly. Gordon and McCurley [GM93] explain why these probabilities might not be identical, but they still seem to be close in their experiments. The Coppersmith method was originally described with the field represented as  $\mathbf{GF}(2)[x]/(f(x))$  for an irreducible  $f(x)$ . I describe it here for the field  $\mathbf{GF}(q_1^{n_1})$  represented as  $\mathbf{GF}(q_1)[x]/(f(x))$  where the characteristic of  $q_1$  is  $p$  and the degree of  $f(x)$  is  $n_1$ . The first step is to find an irreducible  $f(x)$  of the form  $x^{n_1} + f_1(x)$  where the degree of  $f_1(x)$  is very small. Note that it suffices for  $f(x)$  to be a degree  $n_1$  factor of a polynomial of the form  $x^{n_1+r} + f_1(x)$ , where  $r$  and the degree of  $f_1(x)$  are both small. Since approximately  $\frac{1}{n_1}$  of all degree  $n_1$  polynomials are irreducible, it seems reasonable that there should exist such an irreducible  $f(x)$  with the degree of  $f_1(x)$  roughly  $\log_{q_1}(n)$ . It suffices for the analysis in appendix A.3 that  $\deg(f_1(x))$  is  $o(n_1^{1/3} \log(n_1)^{2/3})$ . Select integer parameters  $k, d_1, d_2$ , and set  $h = \lceil \frac{n_1}{p^k} \rceil$ . Let  $w_1(x)$  be of the form  $v_1(x)x^h + v_2(x)$ , where  $\gcd(v_1(x), v_2(x)) = 1$ ,  $v_1(x)$  is monic, and  $v_2(x)$  is non-zero. Computing  $w_2(x) \equiv w_1(x)^{p^k}$  modulo  $f(x)$  yields

$$w_2(x) \equiv w_1(x)^{p^k} \equiv v_1(x)^{p^k} x^{hp^k} + v_2(x)^{p^k} \equiv v_1(x)^{p^k} x^{hp^k - n} f_1(x) + v_2(x)^{p^k}.$$

If  $v_1(x)$  and  $v_2(x)$  share a common factor  $d(x)$ , any factor base relation produced will be the same as will be produced by  $v_1(x)/d(x)$  and  $v_2(x)/d(x)$ . An interesting question is how many relatively prime ordered pairs  $(v_1(x), v_2(x))$  of degree at most  $d_1$  and  $d_2$ , respectively, there are. The answer is provided for the case of unordered pairs with  $q = 2$  and  $d_1 = d_2$  in [BJMV84], and I extend it to the general case (see appendix A.7). The number of relatively prime ordered pairs is equal to  $(q_1^{d_1+d_2+1} - 1)/(q_1 - 1)$ .

We must pick our parameters so that the  $(q^{d_1+d_2+1} - 1)$  relatively prime pairs  $v_1(x)$  and  $v_2(x)$ , with  $v_1(x)$  monic but  $v_2(x)$  not necessarily so, will produce enough equations. With such parameters, the running time will be  $q_1^{(1+o(1))(d_1+d_2)} + q_1^{(2+o(1))m}$ . For this analysis, we fix  $q_1$  and let  $n \rightarrow \infty$ . It would also be interesting to let  $q_1 \rightarrow \infty$ , and even  $p \rightarrow \infty$ , although  $p$  would certainly have to be less than  $n/2$  for the algorithm to be any better than the Waterloo version. A generalisation of Odlyzko's analysis of the  $q = 2$  case (see appendix A.3), suggests parameters should be of the form  $p^k = (\gamma_1 + o(1))n^{1/3} \log(n)^{-1/3}$ ,  $d_2 = (\gamma_2 + o(1))n^{1/3} \log(n)^{2/3}$ , and  $m = (\gamma_3 + o(1))n^{1/3} \log(n)^{2/3}$  for some positive bounded reals  $\gamma_1, \gamma_2, \gamma_3$ , also bounded away from 0.

Applying Coppersmith's methods to Phase 2 is more tricky. We define a sequence of intermediate degree bounds  $B_0 > B_1 > \dots > B_k = m$ . The first step for representing  $\beta(x)$  as a product of factor base elements is like that of the usual Phase 2, but we only represent  $\beta(x)$  as a product of elements of degree at most  $B_0$ . We could do this by first selecting a random integer  $t$  and using the Extended Euclidean Algorithm to represent  $\alpha(x)^t \beta(x)$  as a product of two polynomials of degree roughly  $n/2$ . For each  $B_0$ -smooth irreducible factor,  $a(x)$ , select  $w_1(x)$  polynomials of the form used in Phase 1, with  $a(x) | w_1(x)$ . Different values for  $k$  and  $h$  than in Phase 1 are possible. Then

if  $w_1(x)$  and  $w_2(x) \equiv w_1(x)^{p^r}$  are both  $B_1$ -smooth, we have succeeded in representing  $a(x)$  as a product of irreducible elements of degree at most  $B_1$ . We then repeat this procedure to represent the  $B_i$ -smooth elements as a product of  $B_{i+1}$ -smooth elements until we get  $\beta(x)$  represented as a product of elements in the factor base. Odlyzko [Odl85] and Coppersmith [Cop84] conclude that this step will cost much less than the other phase for the case  $q = 2$ . I believe the same conclusion holds for the general case with  $q$  fixed. These methods could be optimised further, and this would be useful for applications where many postcomputations are necessary.

I note that these are only conjectured improvements since we only *assumed* that the probabilities of smoothness will be like those for randomly selected polynomials. Also, we have no guarantee that the relations these techniques provide will produce a useful linear system. These methods have been implemented effectively in some cases [BFHMV84, Cop84, GM93] so there is evidence to support the claim that they improve the running time.

Another way of speeding up the algorithm for some of these fields of order  $p^n$  was developed by Semaev [Sem91]. His method is applicable when the order of  $p$  in  $\mathbf{Z}_{2n+1}^*$  is either  $n$  or  $2n$ , or when  $GF(p^n)$  is generated by an  $r$ th root of unity for some  $r|p^n - 1$ . The idea seems to have been inspired by the method of Coppersmith of generating relations by mapping (via exponentiation) one set of elements that are likely to be smooth to another set of elements that are likely to be smooth. In the event that they are both smooth, we get a relation. It is claimed to work as long as the characteristic is bounded by a polynomial in  $n$ . Note that Coppersmith's algorithm requires  $p$  to be much smaller. It must certainly be less than  $n/2$  for it to be any better than the Waterloo version. Semaev estimates the expected running time of his algorithm to be  $e^{(c+o(1))(\log(q))^{1/3}(\log \log(n))^{2/3}}$  where  $c = (\frac{4\omega^4}{9(\omega-1)^2})^{1/3}$  and a linear system of dimension  $M$  can be solved in time  $M^{\omega+o(1)}$ .

Adleman [Adl94] introduced a Function Field Sieve algorithm for finding logarithms with a heuristic running time of  $e^{O(\log(q)^{1/3} \log \log(q)^{2/3})}$  for  $\mathbf{GF}(q) = \mathbf{GF}(p^n)$  with  $p$  and  $n$  satisfying  $\log(p) \leq n^{g(n)}$  where  $g$  is any function from  $\mathbf{N} \rightarrow (0, .98)$  which approaches 0 as  $n \rightarrow \infty$ . In [SWD96] they describe Coppersmith's method as a special case of the Function Field Sieve. A summary of the Function Field Sieve discrete logarithm algorithm can be found in [SWD96].

## Smoothness testing

A simple way to test polynomials for smoothness is to factor them into their irreducible factors and look at the degrees. This can be done in polynomial time, but there are much faster methods available.

Note that  $x^{q^i} - x$  is the product of all irreducibles in  $\mathbf{GF}(q)[x]$  of degree dividing  $i$ . We know  $f'(x)$  contains all multiple factors of  $f(x)$ . If  $h(x)$  is a factor of  $f(x)$  with multiplicity  $a$ , then  $h(x)$  is a factor of  $f'(x)$  with multiplicity  $a - 1$ , unless  $a$  is a multiple of the  $p$ . In this last case the multiplicity is  $a$ . Hence a better way to test if a polynomial  $f(x)$  is  $k$ -smooth is to test if  $\gcd(f'(x) \prod (x^{q^i} - x), f(x)) = f(x)$  where the product is over a collection of  $i \in \{1, 2, \dots, k\}$  with the property that every number from 1 through  $k$  divides at least one of those  $i$ . Coppersmith uses the set  $\{\lceil k/2 \rceil, \lceil k/2 \rceil + 1, \dots, k\}$ . If the greatest common divisor is  $f(x)$ , then it is very likely that  $f(x)$  is  $m$ -smooth. The only time we would get a false report is if  $f(x)$  had a factor of degree  $> m$  with multiplicity a multiple of  $p$ . This is very unlikely, and would be detected in the factorisation stage of the algorithm anyway. In order to test if the gcd is  $f(x)$  it suffices to test if  $f'(x) \prod (x^{q^i} - x) \equiv 0 \pmod{f(x)}$ , which can be done efficiently.

Another method is sieving, a common tool in factoring algorithms and in the discrete logarithm algorithms for  $\mathbf{GF}(p)$ . The sieving methods were adapted by Gordon and McCurley for  $\mathbf{GF}(2)[x]/(f(x))$  [GM93]. They use a Gray code to efficiently step through the multiples of a given polynomial. We can adapt their sieve to  $\mathbf{GF}(q)[x]/(f(x))$ , where  $q$  is power of 2, but it is more complicated.

Sieving is useful for finding the smooth elements in a list when we know how to quickly find all the elements in the list that are divisible by a given power of an irreducible. It saves us the trouble of testing each element individually. Testing a list of  $l$  elements with a sieve will require at least time  $l$ . Explicit smoothness testing will cost at most  $l$  smoothness tests. Thus the savings is at most by a factor equal to the cost of one smoothness test. In the Coppersmith algorithm, the savings will be absorbed in our error term. In practice, such a speed-up is very significant. Gordon and McCurley [GM93] produced the factor base relations and solved for the logarithms of the factor base of elements for  $\mathbf{GF}(2^n)$ ,  $n = 227, 313$ , and 401. In the case of  $\mathbf{GF}(2^{401})$ , they sieved through roughly  $2^{44} w_1(x)$  polynomials. They did this by fixing  $v_1(x)$  of degree up to 20 and sieve an array containing the polynomials of degree up to 22 to find the  $v_2(x)$  such that  $w_1(x) = v_1(x)x^h + v_2(x)$  is 19-smooth. This can be distributed to several processors, each sieving through an array for a particular  $v_1(x)$ . When a smooth  $w_1(x)$  is found, we can explicitly test  $w_2(x)$  for smoothness. They have also found what should be enough relations for  $\mathbf{GF}(2^{503})$ , but the linear algebra modulo a 96 digit prime factor of  $q - 1$  poses a problem.

## Other improvements

Other improvements which might reduce the expected running time in practice, but which do not affect the asymptotics are also known, and discussed in [Odl85] and [GM93], for example. They include "early-abort" strategies during smoothness testing, and holding on to relations which only contain a few irreducibles of degree just above the degree bound  $m$ . In the latter case, we hope to cancel out the large degree terms with other such relations and produce a relation for the database. Another idea is to force some smooth factors into  $w_1(x)$  and  $w_2(x)$ .

### 3.3.2 Some new improvements to the above techniques

These improvements do not make a serious dent in the heuristic asymptotic running time estimates of the above algorithms. The difference is absorbed in the  $o(1)$  error term in the exponent. In practice the improvement could be quite significant.

#### Several smooth $w_1(x)$ polynomials for the price of one

This method applies to the Coppersmith algorithm or similar techniques where a set of polynomials that are likely to be smooth are mapped to another set of polynomials that are likely to be smooth. Observe that if  $w_1(x)$  is smooth then so is  $w_1(\alpha x)$  for any  $\alpha$  in  $\mathbf{GF}(q_1)^*$ . So while searching for smooth  $w_1(x)$  it suffices to search through a representative set of size roughly  $1/(q_1 - 1)$  times the set of all  $w_1(x)$ . When we find a smooth  $w_1(x)$ , we can test the  $w_2$  polynomials corresponding to  $w_1(\alpha x)$  for each  $\alpha \in \mathbf{GF}(q_1)^*$ . To find these representative sets we can consider the  $2^{d_1+d_2} - 2^{d_1} - 2^{d_2} + 1$  equivalence classes on the  $w_1(x)$  polynomials where two polynomials

are equivalent if and only if they have zero coefficients in precisely the same coordinates. Within each equivalence class,  $C_k$ , find two coordinates for non-zero coefficients,  $i(k)$  and  $j(k)$ , such that  $\gcd(i(k) - j(k), q_1 - 1) = d$  is minimum over all such pairs. Our representative set will consist of all the polynomials with the  $i(k)$ th coefficient fixed to 1, the  $j(k)$ th coefficient running over  $\alpha^{k(q_1-1)/d}$  for  $k = 0, 1, \dots, d - 1$ , and all the other non-zero coefficients in the class  $C_k$  running over all possible combinations. This will speed up the algorithm by a factor of approximately  $q_1 - 1$ , which is more helpful for larger  $q_1$ .

I made this observation during my research at the University of Waterloo in the summer of 1995. An observation then made by Zuccherato [Zuc95] which should improve matters even when  $q_1 = 2$ , is that the reciprocal polynomial corresponding to  $w_1(x)$ ,  $w_1^R(x) = x^k w_1(1/x)$  where  $k = \deg(w_1(x))$ , is smooth if and only if  $w_1(x)$  is smooth. The structure of the reciprocal is similar to that of  $w_1(x)$ , but not exactly, since  $d_2$  is usually a bit bigger than  $d_1$  (see appendix A.3). If the reciprocal has the same structure as a  $w_1(x)$  then this will slightly increase the rate at which we find relations, but this advantage might be reduced or eliminated if there is no easy way to avoid duplications. One way, if we are not sieving, is to only test a candidate  $w_1$  if it is less than its reciprocal with respect to some easily computable order relation, like lexicographic order.

In practice, however, there will not be much overlap between the set of  $w_1$  polynomials and their reciprocals since  $d_1$  and  $d_2$  will differ. The degree of the  $w_2$  polynomial corresponding to  $w_1^R(x)$  will be slightly higher, but, assuming that the probability of these polynomials being smooth is close to the probability of a random polynomial of the same degree being smooth, we expect that the  $w_2$  polynomials corresponding to these reciprocals are more likely to be smooth than a random  $(w_1, w_2)$  pair being simultaneously smooth. Another advantage, is that this would increase the number of linear relations we can find with a given choice of  $d_1$  and  $d_2$ . This means that in some cases we could use a slightly smaller  $d_1$  or  $d_2$ , which we expect would increase the probability of smoothness, and decrease the running time further. To avoid duplications we could simply check that  $w_1^R(x)$  is not of the same form as the  $w_1$  polynomials. This only needs to be done when  $w_1(x)$  is found to be smooth.

As with the previous *improvements*, these are only conjectured improvements.

### Use subfield identification to get many relations easily

If we could quickly find elements corresponding to elements in subfields  $\mathbf{GF}(q_2)$  of  $\mathbf{GF}(q)$ ,  $q = p^n$ , that are not in  $\mathbf{GF}(q_1)$ , and assuming we can find logarithms in the subfield quickly, then we could look through these subfield elements as polynomials over  $\mathbf{GF}(q_1)$ , and see if they are smooth. If they are, we get a relation for the database. The advantage is that we do not have to hope for two polynomials to be simultaneously smooth. In fact, let us assume we are only interested in using this database to solve the logarithm modulo some factor of  $(q - 1)/(q_2 - 1)$ , like  $\Phi_n(p)$ . Then we do not have to bother solving the logarithm for the subfield element each time. We might even be able to find the smooth polynomials directly, in which case the cost of getting each of these relations is polynomial in  $\log(q)$ .

Suppose  $q_1 = p^{n_1}$ , and  $q = p^{n_1 n_2}$ . Consider  $\mathbf{GF}(q)$  as  $\mathbf{GF}(q_1)[x]/(f(x))$  where  $f(x)$  is an irreducible in  $\mathbf{GF}(p)[x]$ . This is possible if and only if  $\gcd(n_1, n_2) = 1$  ([LN86], Corollary 3.47). Then  $\mathbf{GF}(q_2) = \mathbf{GF}(p^{n_2})$  will correspond to the polynomials with coefficients in  $\mathbf{GF}(p)$ . For all  $k \leq m$ , where  $m$  is the degree bound, factor all irreducibles of degree  $kn_1$  as a product

of  $n_1$  irreducibles of degree  $k$  over  $\mathbf{GF}(q_1)$  ([LN86], Theorem 3.46). We obtain one relation for every  $n_1$  irreducibles in our database. These are certainly not random relations, but this sort of non-randomness might be advantageous, since it guarantees each irreducible occurs in a relation.

Again, this is an observation I made at the University of Waterloo in the summer of 1995.

### Friendly automorphisms

Let  $GF(q)$  be represented as  $GF(p^{n_1})[x]/(f(x))$  where  $f(x)$  is an irreducible of degree  $n_2$ . Consider the automorphism of  $GF(q)$  defined by  $\phi : x \rightarrow x^{p^r}$ . If  $f(x)$  divides  $x^{p^r} + \gamma x + \eta$  for some  $\gamma, \eta \in GF(p^{n_1})$ , then  $\phi$  would not change the degree of any of the representatives of the elements of  $GF(q)$ . This would give us sequences of relations for our database. The automorphism will of course eventually cycle for each element, but the length of this cycle will be up to  $\frac{n_1 n_2}{\gcd(r, n_1 n_2)}$ . Thus this would give up to

$$\frac{\frac{n_1 n_2}{\gcd(r, n_1 n_2)} - 1}{\frac{n_1 n_2}{\gcd(r, n_1 n_2)}}$$

of the relations for our database at a polynomial cost for each one, assuming that the average cycle length for a factor base element is roughly the same as for the entire field.

Since with this automorphism an irreducible of degree  $d$  gets mapped to another irreducible of the same degree, it might be more fruitful to use these relations to restrict our database to one representative from each of these cycles. The appropriate representatives, when the remaining relations are found, can be found in polynomial time. One disadvantage is that the relation matrix will now have large entries. I have not analysed this trade-off. The asymptotic running time estimate will be the same, but there could be a huge savings hidden in the  $o(1)$  error term.

A similar technique could be applied if  $f(x)$  divides  $x^{p^r+1} + \gamma x + \eta$ , using reciprocal polynomials. I am still analysing the applicability of these methods, but I can cite some examples. First we have from [LN86] (Theorem 3.75)

**Theorem 3** Let  $t \geq 2$  be an integer and  $\gamma \in GF(q)^*$ . The binomial  $x^t - \gamma$  is irreducible in  $GF(q)[x]$  if and only if the following two conditions are satisfied:

- (i) each prime factor of  $t$  divides the order  $e$  of  $\gamma$  in  $GF(q)^*$ , but not  $(q-1)/e$
- (ii)  $q \equiv 1 \pmod{4}$  if  $t \equiv 0 \pmod{4}$ .

Suppose  $p = 2, n_2 = 31, n_1 = 5$ . Let  $\gamma$  be a generator for  $GF(2^5)$ . Then  $f(x) = x^{31} - \gamma$  is irreducible. In this case  $r = 5$  is the smallest candidate. We have the relation  $x^{32} \equiv \gamma x$ . This will create cycles of length dividing 31. For example,  $(x+1)^{32} = (\gamma x + 1)$ , followed by  $(\gamma x + 1)^{32} = (\gamma^2 x + 1)$ , and so on, until we get back to  $(\gamma^{30} x + 1)^{32} = (x + 1)$ .

Another example is  $p = 2, n_2 = 127, n_1 = 7$ . Here we let  $\gamma$  be generator for  $GF(2^7)$ , and it follows that  $f(x) = x^{127} - \gamma$  is irreducible. The cycles will now have length 1 or 127. The elements with cycle length 1 will be the elements of  $GF(2^{127})$ , which are not in the database. My computations indicate to me that the probability that a monic polynomial selected uniformly at random from all the monic polynomials of degree up to 25 over  $GF(2^7)$  is 3-smooth is roughly  $2.8568 * 10^{-7}$  and for a degree 19 monic over  $GF(2^7)$  I get  $6.254 * 10^{-5}$ . These computations were done with software I wrote in C, in 1992, on a research term with Scott Vanstone

and Ron Mullin. The probabilities were calculated using recurrence relations similar to those in [BFHMV84]. These probabilities suggest that implementing the Coppersmith index calculus method as described in section 3.3.1, over  $GF(2^7)$ , might be possible if we select our smoothness bound to be 3,  $d_1 = d_2 = 3$ ,  $2^k = 8$ , and  $h = 16$ . In  $GF(2^{127})[x]$  there are 707264 monic irreducibles of degree at most 3. This includes 1 plus 5569 sets of 127 elements which lie in the same cycle of this automorphism. We can thus use this set of 5569 elements as our database. The approximately  $2^{49}$  coprime  $(v_1(x), v_2(x))$  pairs we test should give us about 10000 smooth  $(w_1(x), w_2(x))$  pairs, assuming the polynomials we use behave as random ones of degree at most 19 and 25. Of course  $2^{49}$  is a formidable number of pairs to test. However Gordon and McCurley [GM93] effectively tested  $2^{43}$  pairs in their experiments. Also, if we implement the suggestion of section 3.3.2, we only need to test about 1/127th of those pairs for smoothness. We could fix the values of  $v_1(x)$  and the leading coefficient of  $v_2(x)$ , and sieve through an array of  $2^{21}$  elements. We need to do this about  $2^{21}$  times. Note that sieving does not allow us to avoid  $(v_1(x), v_2(x))$  pairs that are not coprime, so whenever we do find a smooth  $w_1(x)$  we should check that  $v_1(x)$  and  $v_2(x)$  are coprime.

An alternative is to implement the index calculus techniques with the field represented as  $GF(2)[x]/(f(x))$ . In [Odl85], Odlyzko suggests that just for completing the equation generation phase, optimal parameters for applying Coppersmith's algorithm to  $GF(2^{880})$  would require  $d_1 = 27$ ,  $d_2 = 29$ ,  $m = 36$ ,  $2^k = 8$ , and  $h = 110$ , and would require about  $2^{62}$  basic operations. This does not take into account the possibility of sieving, but does take advantage of other techniques. We would still have to sieve through  $2^{58}$   $(v_1(x), v_2(x))$  pairs, whereas with the modifications I have described, we would only need to sieve about  $2^{42}$  pairs. This factor of  $2^{16}$  between the two implementations will likely diminish when we consider the fact that working over  $GF(2^7)$  is more complicated and not as convenient on readily available hardware. We must test the smoothness of  $2^7 - 1$  possible  $w_2(x)$  candidates every time we find a smooth  $w_1(x)$ , but this should not be significant since smooth  $w_1(x)$  polynomials will be infrequent, and most of our time is spent finding them.

I made this observation at the University of Waterloo in the summer of 1995.

### 3.4 Techniques for $GF(p)$

The techniques described for the case of  $GF(p)[x]/(f(x))$  carry over to  $\mathbf{Z}/p\mathbf{Z}$ . The simpler index calculus implementations for  $GF(p)$  set factor bases to be the set of all primes less than some bound  $m$  and use factoring to test smoothness. Pomerance does this in his rigorous algorithm in [Pom87]. This algorithm has an expected running time of  $e^{(\sqrt{2}+o(1))(\log(p)\log\log(p))^{1/2}}$ . Smoothness testing is obtained by elliptic curve factoring methods. He chooses  $m$  to be  $e^{\sqrt{1/2}(\log(p)\log\log(p))^{1/2}}$ . Note that unlike the case of polynomials over a finite field, we do not know how to factor efficiently with respect to the size of the numbers being factored.

As with the polynomial ring case, there are many ways to improve the running time in practice, many of which are direct analogues of the methods I have described in the previous sections. We could include  $-1$  in the factor base, and we can again use the Extended Euclidean Algorithm to represent any integer modulo  $p$  as a quotient of two integers of size roughly  $\sqrt{p}$ . Three heuristic algorithms are presented in [COS86] with running time  $e^{(\sqrt{2}+o(1))(\log(p)\log\log(p))^{1/2}}$ . Mc-

Curley [McC90] describes their *Gaussian* integers method. LaMacchia and Odlyzko [LO91a] implemented this method for  $\mathbf{GF}(p)$  with  $p$  a 58 digit prime.

The Number Field Sieve, adapted from the integer factorisation algorithms, has been applied to the discrete logarithm problem to produce algorithms with heuristic expected running time  $e^{((64/9)^{1/3} + o(1))(\log(q))^{1/3}(\log \log(q))^{2/3}}$ . The survey [SWD96] describes these methods, and provides references. These methods have been implemented [Web96] to find the logarithms of the first ten primes to the base 7 modulo a 65 digit (215 bit) prime. The techniques have also been used to compute the logarithm of several small primes to the base 7 modulo a 129 digit (427 bit) number.

### 3.5 A More General Look at Index Calculus

There will not always be a natural factor base in a group  $G$ , one with which we can quickly detect smooth elements and quickly factor into the factor base elements. For example, suppose  $G$  is a proper subgroup of  $\mathbf{GF}(p)^*$  of order  $N$ . There does not seem to be a natural way to select a factor base in  $G$  that will yield an algorithm that runs in time proportional to the running time of index calculus algorithms in  $\mathbf{GF}(p')$  where  $p'$  is a prime of size close to  $N$ . This observation is key to the security of the U.S. Government NIST Data Signature Standard (see [MvV96]). One possibility is to embed  $G$  into a group where there is a natural factor base. In this case, it would be the group  $\mathbf{GF}(q)^*$ . Another example is the additive group of points on an elliptic curve over  $\mathbf{GF}(q)$ . There does not seem to be a natural factor base for this group either. However, for the class of supersingular curves, Menezes, Okamoto and Vanstone [MOV93] showed how to reduce the discrete logarithm problem to one in a subgroup of an extension of  $\mathbf{GF}(q)$  of degree at most 6, yielding a subexponential algorithm. However there is still no such algorithm known for non-supersingular curves.

A cyclic subgroup  $G$  of a cyclic group  $H$  is also the homomorphic image of the group  $H$ . This might be a more fruitful way of considering the relationship between  $G$  and  $H$ . Let our group  $G$  be the image of the group  $H$  under some group homomorphism,  $\phi$ . Thus  $G \approx H / \ker(\phi)$ . We no longer require  $G$  to be a subgroup of  $H$ . For example, finite fields can be represented as  $R/I$  for some maximal ideal  $I$  of an integral domain  $R$ . Here  $I$  is the kernel of a ring homomorphism  $\phi : R \rightarrow \mathbf{GF}(q)$ . For example,  $R = \mathbf{GF}(p)[x]$ , and  $I = (f(x))$  for some irreducible  $f(x)$ , or  $R = \mathbf{Z}$ , and  $I = (p)$ . In these two cases, with the algorithms I described in this chapter, we use the natural factor bases in  $\mathbf{GF}(p)[x]$  and  $\mathbf{Z}$  and a homomorphism onto the fields to create relations in the finite field. The factor bases consist of elements of *size* up to  $m$  for some selected smoothness bound  $m$ . We could let  $R$  be a ring of algebraic integers, for instance, and the factor base could be elements with norms up to a given size. Smoothness testing could consist of testing the norm of an element for smoothness. However, if  $R$  is not a principal ideal domain, smoothness of the norm will be a necessary but not sufficient condition for the element to have a factorisation into irreducible elements of small norm. It seems more natural in this case to use the set of prime ideals with smooth norms as a factor base. Performing linear algebra over the integers will ultimately give multiplicative relations between principal ideals, which translate into relations between elements of  $R$  modulo the group of units. In the case of  $\mathbf{GF}(p)[x]$  and  $\mathbf{Z}$ , the group of units do not pose a problem, but they can in other rings.

Working with ideals, it is not clear that we can do the linear algebra modulo  $N$ , the order of the group, and get useful information. One option, for example, when  $R$  is a ring of algebraic integers, is to do the linear algebra modulo  $Nh$ , where  $h$  is the ideal class number of  $R$ . Since the  $h$ th power of a prime ideal is principal, and  $N$ th powers of elements in  $R \setminus I$  get mapped to 1 by  $\phi$ , we again only have the units left to worry about. It suffices to do the linear algebra modulo  $NM$  where  $h = cM$  and  $c$  is coprime with  $N$ . Lovorn [Lov92] works modulo a number that we know will be a multiple of  $NM$ . Other methods are possible which permit us to work modulo  $N$ . See [SWD96] for a description and more references.

### 3.6 Other implementations

So far, the rigorous algorithms I have described will solve the discrete logarithm problem in a field of size  $q = p^n$  in time  $q^{o(1)}$  provided  $n \rightarrow \infty$  or  $n = 1$ . Lovorn [Lov92] provides a rigorous algorithm with running time  $e^{(\sqrt{6}+o(1))(\log(q) \log \log(q))^{1/2}}$  for solving logarithms in  $GF(p^2)$ . She uses smooth-normed prime ideals of the ring of integers of an algebraic number field as her factor base.

El Gamal gives a similar heuristic algorithm for the case of  $n$  fixed [ElG85b, ElG85a]. Adleman and DeMarrais [AD93] describe a heuristic algorithm for  $n < p$ . Semaev [Sem95] gives a heuristic algorithm which he claims should work in all finite fields. All of these algorithms use prime ideals of the ring of integers of an algebraic number field in the factor base and have heuristic expected running time  $e^{O((\log(q) \log \log(q))^{1/2})}$ . Adleman and DeMarrais [AD93] also describe a polynomial ring algorithm for the case  $n \geq p$  to get a heuristic algorithm for all fields with same running time.

With these algorithms, as well as the Function Field Sieve mentioned in section 3.3.1 and the Number Field Sieve mentioned in section 3.4, additional work is done to handle the problems mentioned in the previous section.

# Chapter 4

## Quantum Computers

### 4.1 What is a quantum computer?

A quantum computer is a computing device which exploits quantum mechanical effects. Details can be found in several articles including [DiV95] and [EJ96].

Given a system with  $N$  possible observable states,  $S_1, S_2, \dots, S_N$ , at any point in time we expect the system to be in exactly one of those states. Suppose however, that the system can exist in several of those states at the same time. We say the system is in a *superposition* of those states. And suppose that when we observe the system it then enters one of the  $N$  states. The model used is the following.

The state of a system is an element in the  $N$  dimensional complex vector space with the  $S_i$  as basis elements.

We denote this by

$$\sum_{i=1}^N \alpha_i |S_i\rangle.$$

We have the restriction that

$$\sum_{i=1}^N |\alpha_i|^2 = 1.$$

The  $\alpha_i$  are called amplitudes. Further, when we observe the system, it will enter state  $S_i$  with probability  $|\alpha_i|^2$ .

While on a classical computer a bit of information is either a 0 or a 1, on a quantum computer the value of a bit is a superposition of 0 and 1, denoted  $\alpha_0|0\rangle + \alpha_1|1\rangle$ , with  $|\alpha_0|^2 + |\alpha_1|^2 = 1$ . We call this quantum version of a bit a *qubit*.

Likewise, the state of an  $n$ -qubit system will be a superposition of the form

$$\sum_{c \in \{0,1\}^n} \alpha_c |c\rangle,$$

where

$$\sum_{c \in \{0,1\}^n} |\alpha_c|^2 = 1.$$

The system will exist in this superposition until we observe the qubits. Then the system will enter one of the  $2^n$  states of the superposition with probability proportional to the absolute value of the square of the amplitude associated with that state.

We can associate with an  $n$ -qubit system, a Hilbert space in the complex vector space with the  $2^n$  qubit configurations as a basis.

To perform computations we apply transformations to these *qubits*. These transformations map each  $n$ -qubit state to some superposition of other  $n$ -qubit states. States can get mapped to by many different states, and with various amplitudes. The amplitude of a given state after the transformation will be sum of these amplitudes. Such transformations can be represented by a *transition matrix*, which is a  $2^n \times 2^n$  matrix whose  $(i, j)$  entry is the amplitude with which state  $|i\rangle$  gets mapped to state  $|j\rangle$ . This is the matrix representation of the linear transformation with respect to our basis. The laws of quantum mechanics stipulate that any such transformation must be *unitary*, that is, the transition matrix must have as inverse its own conjugate transpose. Such computations are reversible.

So we can imagine a quantum computer to be a device which lets us walk through the elements of this vector space, and the steps are unitary linear transformations.

We can perform these transformations by means of *quantum circuits*, constructed as a network of a finite set of basic quantum gates. It has been shown that the controlled-NOT gate, which is a two-bit gate, together with one-bit rotations [BBC<sup>+</sup>95] can be used to approximate any quantum circuit. These gates take only one or two qubits as input, and leave the remaining ones unchanged. The complexity of computations will be measured by the number of basic quantum gates needed to implement them.

A quantum computation would work as follows. Set a register of *qubits* to a desired starting position, apply a unitary transformation to these qubits, using elementary quantum gates, and then observe the qubits. We can repeat such experiments several times, and derive whatever information we may from the values that we observe. If we only measure one qubit, and observe a 0 for example, the system will then be in a scaled superposition of all the states which had a 0 value for that qubit. So another possibility in our computations is to observe only some of the qubits, and then continue computing with the remaining superposition.

This might seem like nothing more than a glorified randomised algorithm. In the current context, a classical randomised algorithm corresponds to flipping a quantum coin, observing it, and then using it in the computations. In this manner the registers will always be in one state and not in a superposition of many states. When we have a superposition of states, however, and we apply a unitary gate, each element in the superposition is transformed into a new superposition of states. All of the new superpositions are added up. The coefficients are complex numbers, and thus can interfere constructively or destructively.

More specifically, the probability associated with a state  $|a\rangle$  of amplitude  $\gamma$  is  $|\gamma|^2$ . If the state came about several different ways, for example two different states simultaneously transformed to state  $|a\rangle$  with amplitudes  $\gamma_1$  and  $\gamma_2$ , then the amplitude of  $|a\rangle$  is  $\gamma_1 + \gamma_2$ , with corresponding probability  $|\gamma_1 + \gamma_2|^2$ . The probability of observing  $|a\rangle$  is not simply  $|\gamma_1|^2 + |\gamma_2|^2$ , but can range anywhere from  $|\gamma_1|^2 - 2|\gamma_1||\gamma_2| + |\gamma_2|^2$  (destructive interference) to  $|\gamma_1|^2 + 2|\gamma_1||\gamma_2| + |\gamma_2|^2$  (constructive interference), depending on the angle between  $\gamma_1$  and  $\gamma_2$  in the complex plane.

A possible advantage of having registers which remain in a superposition of states during computation is that we might be clever enough to perform unitary transformations such that the amplitudes of the states we are interested in interfere more constructively than the amplitudes of the other states. If this happens, when we perform a measurement, we are more likely to observe a state we are interested in. This is what the quantum factoring and discrete logarithm algorithms do.

Potential physical realisations of such a computing device are discussed in [EJ96] and [DiV95], for example. One key problem is that such a computer would interact with the environment and errors would be introduced. Error-correcting codes solve this problem in classical situations, and similar solutions are being sought for quantum computers.

The running times stated are the number of quantum operations, selected from a finite set of basic quantum gates.

## 4.2 Quantum Computation Tools

If we compute a function on a superposition of states, we will get a superposition of the value of that function on those states. Shor [Sho94a] points out that a deterministic computation can be implemented on a quantum computer if and only if it is reversible.

I state three quantum computer algorithms, which are components of the quantum discrete logarithm algorithm. Only the first two are needed in the case of finite fields.

**The Discrete Fourier Transform**,  $A_q$ , is a unitary operation which for a number  $a \in \{0, 1, \dots, q-1\}$ , maps state  $|a\rangle$  to

$$\frac{1}{\sqrt{q}} \sum_{c=0}^{q-1} \xi_q^{ac} |c\rangle$$

where  $\xi_q = e^{\frac{2\pi i}{q}}$ .

It is shown by Shor [Sho94b] that if the prime power factors of  $q$  were of size polynomial in  $\log(q)$ , then  $A_q$  could be implemented efficiently. Coppersmith and Deutsch independently showed how to efficiently construct  $A_{2^n}$ , and Cleve [Cle94] showed that  $A_q$  can be efficiently implemented if all the prime factors of  $q$  are of size polynomial in  $\log(q)$ . We use the fact that  $A_{2^n}$  can be implemented with a circuit of size  $O(n^2)$ .

Note that  $A_q$  maps  $|0\rangle$  to

$$\frac{1}{\sqrt{q}} \sum_{c=0}^{q-1} |c\rangle,$$

which is an equally weighted superposition of the integers modulo  $q-1$ . Now suppose we want an **equally weighted superposition** of the integers modulo  $N$  for an integer  $N$  which is not smooth. Algorithm 4.2.1 does this with a network of size  $O(\log(N))$ , as suggested to me by Ekert [Eke96].

We can also use a quantum computer to **find the order of an element in group**. In Shor's [Sho94b] description of the integer factorisation algorithm, he gives a polynomial time algorithm for finding the order of an element in a cyclic group, provided the group operation can be carried out in polynomial time. This lets us find the order of elements in  $\mathbf{Z}_N^*$ , which gives a probabilistic algorithm for factoring  $N$ .

---

**Algorithm 4.2.1** Obtaining a Uniform Superposition

---

1. Set  $L + 1$  qubits to 0.
2. Apply a rotation to the first  $L$  qubits which maps  $|0\rangle$  to  $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ .

This gives us

$$\frac{1}{\sqrt{2^L}} \sum_{a=0}^{2^L-1} |a, 0\rangle.$$

3. Apply a function mapping  $|a, 0\rangle$  to  $|a \bmod N, \lfloor a/N \rfloor\rangle$ .
  4. Perform a measurement on the  $(L + 1)$ st qubit. If the result is a 0 we know the first  $L$  bits will be in a superposition of all the states that had a 0 in the  $(L + 1)$ st qubit. This gives us precisely the superposition we sought. This will occur with probability at least  $1/2$ . Otherwise, go to step 1.
- 

### 4.3 Quantum discrete logarithm algorithm

I now discuss how to find a discrete logarithm,  $\log_\alpha(\beta)$ , in the cyclic group  $\langle \alpha \rangle$  of order  $N$ . For  $GF(q)^* = \langle \alpha \rangle$ , we have  $N = q - 1$ . For a group where the order is not known, use classical methods or Shor's algorithm to find it. I am following the technique given by Shor [Sho94b], but with a few modifications. The first one is that I describe it for any group. Boneh and Lipton describe and analyse a similar experiment in [BL95] to solve an even more general problem.

Firstly, to avoid the difficulties in the analysis caused by small prime factors of  $N$ , we solve classically for  $\log_\alpha(\beta)$  modulo all prime power factors of  $N$  with the prime less than  $2^{10} \log(N)^2$ . This can be done very efficiently using the techniques from Chapter 2 in  $O(\log(N)^2)$  classical group operations. This will also reduce the expected number of quantum operations necessary.

Let  $Q = 2^L$  be a power of 2 satisfying  $N < Q \leq 2N$ , so that  $A_Q$  can be applied efficiently. We have three registers with  $Q$  qubits each. The discrete logarithm algorithm is described in Algorithm 4.3.1.

#### 4.3.1 Analysis

After performing the experiment, with probability at least  $\frac{1}{120+o(1)}$ , the values of  $c$  and  $d$  that we read will allow us to compute  $k = \log_\alpha(\beta)$  modulo  $N$ . Algorithm 4.3.2 describes the method.

By an analysis almost identical to Shor's, we get the expected running time of this algorithm to be at most  $120 + o(1)$  repetitions of the quantum experiment, which is thus  $O(\log(N)M(N))$  basic quantum operations.

I now summarise why this algorithm is claimed to work.

The probability of observing the state  $|c, d, \alpha^k\rangle$  is

$$\left| \frac{1}{NQ} \sum_{a \equiv k+rb} \xi_Q^{ac+bd} \right|^2.$$

---

**Algorithm 4.3.1** Discrete Logarithm Experiment

---

**Step 1:** Initialise the registers to 0. Assume this takes time  $O(\log(N))$ .

**Step 2:** Set the first two registers to a uniform superposition of all the residues modulo  $N$  :

$$\frac{1}{N} \sum_{a=0}^{N-1} \sum_{b=0}^{N-1} |a, b, 0\rangle.$$

**Step 3:** Then compute  $\alpha^a \beta^{-b}$ , and store the result in the third register to yield

$$\frac{1}{N} \sum_{a=0}^{N-1} \sum_{b=0}^{N-1} |a, b, \alpha^a \beta^{-b}\rangle.$$

This takes a network of size  $O(\log(N)M(\log(N)))$ , where  $M(n)$  is the complexity of multiplying two  $n$  bit integers.

**Step 4:** Apply the Fourier transform  $A_Q$  on the first two registers to get

$$\frac{1}{NQ} \sum_{c,d=0}^{Q-1} \sum_{a,b=0}^{N-1} \xi_Q^{ac+bd} |c, d, \alpha^a \beta^{-b}\rangle.$$

This takes a network of size  $O(\log(N)^2)$ .

**Step 5:** Lastly, observe the state of the registers. This takes  $O(\log(N))$  steps.

---

---

**Algorithm 4.3.2** Quantum Discrete Logarithm Algorithm

---

1. Run the discrete logarithm experiment. Let  $c$  and  $d$  be the values observed the first two registers.
  2. Let  $l = \lfloor \frac{cN}{Q} \rfloor$ .
  3. Let  $m$  be the closest integer to  $\frac{dN}{Q}$ .
  4. Let  $k = \gcd(N, l)$ . If  $k > 1$  go to step 1. (Alternatively, we could solve  $r \frac{l}{k} \equiv m$  modulo  $N/k$  in the next step).
  5. Solve  $rl \equiv m$  modulo  $N$ .
  6. Compute and compare  $\alpha^r$  and  $\beta$ .  
If they are not equal, return to step 1.  
If they are equal, stop.
-

The sum is over all  $a, b \in \{0, 1, 2, \dots, N-1\}$  satisfying  $a \equiv k + rb$  modulo  $N$ .

Let  $l = \lfloor \frac{cN}{Q} \rfloor$ , and  $L = \frac{r}{N}lQ + d$ .

Shor shows [Sho94b] that this amplitude has absolute value squared greater than  $\frac{1}{3Q^2}$  if

$$\min_{i \in \mathbf{Z}} |L - iQ| \leq \frac{1}{2} \quad (4.1)$$

and

$$|(cN \bmod Q)| \leq \frac{Q}{20}. \quad (4.2)$$

Since  $\gcd(N, Q) = 1$ , equation (4.2) holds for at least  $\frac{Q}{20}$  of all  $c \in \{0, 1, \dots, Q-1\}$ , and for each such  $c$ , there is at least one  $d$  for which (4.1) holds.

So for each of the  $N \geq \frac{Q}{2}$  choices of  $k$ , there are  $\geq \frac{Q}{20}$   $(c, d)$  pairs for which equations (4.1) and (4.2) hold. Thus the total number of states satisfying (4.1) and (4.2) is at least  $\frac{NQ}{20}$ . By definition these parameters occur with probability at least  $\frac{1}{3Q^2}$ . These observations tell us that the probability that the state we observe satisfies (4.1) and (4.2) is at least  $\frac{1}{120}$ . Note also that each  $(c, d)$ -pair which satisfies (4.1) and (4.2), does so for every  $k$ , and thus with probability at least  $\frac{1}{6Q}$ .

We know  $N, l, Q$ , and  $d$ , and we know  $|\frac{rlQ}{N} + d|$  is within  $\frac{1}{2}$  of some multiple of  $Q$ . Equivalently,  $|rl + \frac{dN}{Q}|$  is within  $\frac{Q}{2}$  of some multiple of  $N$ . Since  $l$  and  $r$  are integers, the solution,  $r$ , must satisfy  $xl \equiv m$  modulo  $N$  where  $m$  is the closest integer to  $\frac{dN}{Q}$ . This will have precisely  $\gcd(l, N)$  solutions, each congruent to  $r$  modulo  $N/\gcd(l, N)$ . This explains the above algorithm.

We will not know beforehand if (4.1) and (4.2) are satisfied. We just expect it to happen at least once every 120 times, in which case we can solve and verify the valid solution.

We still need to be sure that  $l$  has a high probability of being coprime with  $N$ . Recall that at least  $\frac{1}{20}$  of the possible  $c \in \{0, 1, \dots, Q-1\}$  satisfy (4.1) and (4.2) and each of these are read with probability at least  $\frac{1}{6Q}$ . All of these  $c$  correspond to at least  $\frac{1}{20}$  of the possible  $l \in \{0, 1, \dots, N-1\}$ , and each of these  $l$  will occur with probability at least  $\frac{1}{6Q}$ . Since  $N$  has no prime factor less than  $2^{10} \log(N)^2$ , the proportion of  $l$  not coprime with  $N$  is at most  $\frac{1}{2^{10} \log(N)}$  and thus  $o(1)$  as  $N \rightarrow \infty$  and always negligible compared to  $1/20$ . Thus the probability, for each run, of getting a reading satisfying (4.1) and (4.2) and for which  $l$  is coprime with  $N$ , is roughly  $\frac{1}{120}$  or better.

We will solve for  $r \bmod N$  in an expected  $120 + o(1)$  repetitions of this quantum experiment. Shor expects  $120t$  repetitions of the quantum experiment, for some constant  $t > 2$ , but his classical computations only require  $O(\log(N))$  group operations.

## 4.4 Concluding remarks

The overall algorithm I describe requires  $O(\log(N)M(N))$  basic quantum operations, and  $O(\log(N)^2)$  classical group operations. The classical component could be reduced as in Shor's paper [Sho94b],

at the cost of more complicated analysis, as well as possibly more repetitions of quantum operations, and larger registers. None of these differences increase the asymptotic running time estimate in terms of quantum operations. However, especially in the first phases of ever implementing quantum computers, quantum operations will be much more expensive than classical ones, and so these classical precomputations will be worthwhile, and should perhaps be increased to remove many other small factors from  $N$ .

Note that two of the three quantum registers require  $\lceil \log_2(N) \rceil$  qubits, and the third requires however many bits are necessary to store a group element. The size of the first two registers can be decreased by computing the logarithm separately modulo any factors of  $N$  we may know. It may be worthwhile to apply factoring algorithms, classical and quantum, to factor  $N$  and thereby allow us to use two smaller registers for the discrete logarithm computations.

If we are computing modulo factors of  $N$ , it may be possible to simplify the group representation for that subgroup, thereby reducing the size of the third register as well as simplifying the group operation. One case where this is possible is with fields with subfields, as described in Chapter 3.

## Chapter 5

# Conclusions

There are rigorous deterministic techniques for computing logarithms in any finite cyclic group of order  $N$  in time proportional to  $\sqrt{N} \log(N)$  and space for  $\sqrt{N}$  group elements. There are heuristic probabilistic algorithms for performing the same task with expected running time  $O(\sqrt{N})$  and requiring  $O(1)$  group elements of space.

The discrete logarithm problem in finite fields can be solved not only by these methods, but by *Index Calculus* techniques which involve building up a collection of linear relations between elements from a small collection of field elements called a *factor base*. By then expressing a given element of the field as a product of elements in the factor base, we can extract information from the linear relations that enables us to solve for the logarithm. The two central components of this algorithm are generating linear relations between elements of the factor base, and linear algebra over the integers modulo  $N$ . There are rigorous probabilistic methods for performing these two tasks. In practice people use heuristic techniques to speed up the solution of these two steps. I have described several of the heuristic methods used to increase the rate at which we should be able to find relations between factor base elements. Many of these have been implemented and tested. I have suggested a few other techniques that could speed up the equation generation phase for some finite fields.

Gaussian elimination is an obvious method for solving the linear algebra phase, but other techniques, including Wiedemann's coordinate recurrence methods will solve the problem asymptotically faster, and provide us with the fastest rigorous and heuristic index calculus algorithms for finding discrete logarithms. I have detailed how we can apply Wiedemann's method in the case that we do not know the prime factorisation of  $N$ . I have brought some attention to potential problems with applying Wiedemann's techniques (which are used in the rigorous algorithms) to systems which do not have full rank modulo higher powers of some prime divisor of  $N$ . This problem can be remedied for the purposes of the index calculus algorithm, but it is an interesting problem which I am still studying.

I have also shown that the discrete logarithm problem in any group of order  $N$ , can be reduced in polynomial time to solving discrete logarithms in a collection of subgroups whose orders contain each of the distinct prime factors of  $N$ . Any combination of *square root*, *index calculus*, or other methods can then be applied in these individual subgroups.

The possibility of a *quantum computer* which would exploit quantum mechanical effects has

been investigated for over a decade. Such a device would permit the computation of discrete logarithms in finite fields in a polynomial number of quantum operations. Whether or not a quantum computer can be realised is an open question.

# Appendix A

## A.1 The expected value of $\gcd(n, N)$

We are interested in the expected value of  $\gcd(n, N)$  where  $n$  is selected uniformly at random from the integers modulo  $N$ . Suppose  $N = \prod p_i^{a_i}$ .

Firstly consider the prime power case. Since  $p_i^{a_i} | N$ , the value of  $n \bmod p_i^{a_i}$  also has a uniform probability distribution. The expected value of  $\gcd(n, p_i^{a_i})$  is  $1(\frac{p-1}{p}) + p(\frac{p-1}{p^2}) + \dots + p^{a-1}(\frac{p-1}{p^a}) + p^a \frac{1}{p^a} = a \frac{p-1}{p} + 1$ .

By the Chinese Remainder Theorem, the values of  $\gcd(n, p_i^{a_i})$  for different  $i$  are independent, and thus the expected value of the product is the product of the expected values. So the expected value of  $\gcd(n, N)$  equals  $\prod (a_i(\frac{p_i-1}{p_i}) + 1)$ , which is less than  $\prod (a_i + 1)$ . This last product equals  $d(N)$ , the number of divisors of  $N$ .

## A.2 Hensel lifting in non-singular matrices

We are given a matrix  $A$ , and a tuple  $\mathbf{b}$ , defined modulo  $n^a$ ,  $a > 1$ , and we wish to solve the system  $A\mathbf{x} = \mathbf{b}$ .

Let  $\mathbf{b}_0 = \mathbf{b}$ . We first solve  $A\bar{\mathbf{x}}_0 = \mathbf{b}_0$  modulo  $n$ . Then pick any  $\mathbf{x}_0$  modulo  $n^a$  such that  $\mathbf{x}_0 = \bar{\mathbf{x}}_0 \bmod n$ .

Let  $\mathbf{b}_1 = (\mathbf{b}_0 - A\mathbf{x}_0)/n$ . Then solve  $A\bar{\mathbf{x}}_1 = \mathbf{b}_1 \bmod n$ , and pick any  $\mathbf{x}_1$  modulo  $n^{a-1}$  such that  $\mathbf{x}_1 = \bar{\mathbf{x}}_1 \bmod n$ .

Continue this procedure, for  $i < a - 1$ , setting  $\mathbf{b}_{i+1} = (\mathbf{b}_i - A\mathbf{x}_i)/n$ , solving  $A\bar{\mathbf{x}}_{i+1} = \mathbf{b}_{i+1} \bmod n$ , picking any  $\mathbf{x}_{i+1}$  modulo  $n^{a-i-1}$  such that  $\mathbf{x}_{i+1} = \bar{\mathbf{x}}_{i+1} \bmod n$ , and incrementing  $i$ .

An induction argument shows that  $\mathbf{x} = \mathbf{x}_0 + n\mathbf{x}_1 + \dots + n^{i-1}\mathbf{x}_{i-1}$  is a solution to  $A\mathbf{x} \equiv \mathbf{b}$  modulo  $n^i$ , for  $i \leq a$ .

## A.3 Selecting Parameters for Coppersmith's algorithm

We must pick our parameters so that the approximately  $q^{d_1+d_2}$  relatively prime pairs  $v_1(x)$  and  $v_2(x)$  will produce enough equations. The running time is thus  $q^{(1+o(1))(d_1+d_2)} + q^{(1+o(1))2m}$ . For this analysis, we fix  $q$  and let  $n \rightarrow \infty$ .

If we assume that the probability of smoothness is the same for polynomials of this form as for random polynomials of the same degree, the probability of each trial producing a relation is roughly  $P_q(\max(d_2, d_1 + h), m)P_q(\max(p^k d_2, d_1 p^k + p^k + \deg(f_1)), m)$ .

Thus to produce enough equations requires

$$q^{d_1+d_2} P_q(\max(d_2, d_1 + h), m) P_q(\max(p^k d_2, d_1 p^k + p^k + \deg(f_1)), m) \geq q^m.$$

We seek parameters which minimise the running time of the three main phases. With the parameters given below which were chosen to minimise the running time of the database generation and linear algebra phases, it turns out that the running time of the third phase is negligible.

In Odlyzko's analysis [Odl85], he assumes that  $d_1 = d_2 = d$ , which makes things simpler, and as we show here, this has no adverse affect on the end result. In any useful application of this method, we must have  $d_2 < h$ . Thus the degree of  $w_1(x)$  is at most  $d_1 + h$ . The degree of  $w_2(x)$  is at most  $\max\{p^k d_2, d_1 p^k + (h p^k - n) + \deg(f_1)\}$ . For a fixed  $d_1 + d_2$ , there is no advantage to keeping  $p^k d_2 < d_1 p^k + (h p^k - n) + \deg(f_1)$  since increasing  $d_2$  and decreasing  $d_1$  would decrease the degree (and thus we assume the likelihood of smoothness) of both  $w_1(x)$  and  $w_2(x)$ . We thus assume that the degree of  $w_2(x)$  is at most  $p^k d_2$ . Our constraint now becomes

$$q^{(1+o(1))(d_1+d_2)} P_q(d_1 + h, m) P_q(p^k d_2, m) \geq q^{(1+o(1))m}.$$

If we assume the hypotheses of Theorem 1 are met, computations show that we must have  $d_2 = n^{1/3+o(1)}$ ,  $m = n^{1/3+o(1)}$ ,  $p^k = n^{1/3+o(1)}$  and thus  $h = n^{2/3+o(1)}$ . The hypotheses of Theorem 1 are satisfied with these parameters. Since  $d_1 \leq d_2$ , we see that  $d_1$  will have no effect in the asymptotic estimate we have for  $P_q(d_1 + h, m)$ . A generalisation of Odlyzko's analysis of the  $q = 2$  case, suggests parameters to be of the form  $p^k = (\gamma_1 + o(1))n^{1/3} \log(n)^{-1/3}$ ,  $d_2 = (\gamma_2 + o(1))n^{1/3} \log(n)^{2/3}$ , and  $m = (\gamma_3 + o(1))n^{1/3} \log(n)^{2/3}$  for some positive bounded reals  $\gamma_1, \gamma_2, \gamma_3$ , also bounded away from 0. Ignoring the fact that  $d_1, d_2, m$  and  $k$  are integers yields optimal values of  $\gamma_3 = \gamma_2 = (2/(3 \log(q)))^{2/3}$ ,  $\gamma_1 = 1/\gamma^{1/2} = (3 \log(q)/2)^{1/3}$  and  $d_1 = (1 + o(1))d_2$  (we assume  $\deg(f_1(x))$  is  $o(d_2)$ ).

The error terms compensate for the fact the we actually need integers, except in the case of  $p^k$ . These constants serve as lower bounds for the upper bound of the running time of the Coppersmith improvement. Odlyzko states in the case  $q = 2$  that the constant is periodic in  $\log_2(n^{1/3} \log(n)^{2/3})$  and gives an upper bound of  $(2/\log(2))^{2/3}/2$ . A similar upper bound should exist in the general case as well.

These serve as heuristic upper bounds for the entire algorithm only assuming that the third phase does not take too long. I have not analysed the third phase very closely for  $q > 2$ , but it does seem to get more difficult for smaller degree extensions because there are fewer choices for the intermediate degree bounds (see section 3.3.1).

## A.4 Chebyshev correction

Equation (5) on page 9 of [HR83] reads

$$P\left(\sum_{i=1}^{N_2} X_i \leq N_1\right) < \frac{\delta(1-\delta)}{N_2(N_1/N_2)^2}.$$

This is supposed to be an upper bound on the probability that fewer than  $N_1$  successes are obtained in  $N_2$  Bernoulli trials, each with probability of success  $\delta$ . It seems incorrect intuitively, since increasing the number of successes makes the upper bound smaller.

Note that

$$P\left(\sum_{i=1}^{N_2} X_i \leq N_1\right) = P\left(\sum_{i=1}^{N_2} (\delta - X_i) \geq \delta N_2 - N_1\right)$$

which, provided  $N_1 < \delta N_2$ , Chebyshev's inequality implies is less than

$$\frac{\delta(1-\delta)}{N_2(\delta - N_1/N_2)^2}.$$

The values they end up using have  $N_1/N_2 = \delta/2$  so the error does not affect their result.

Even assuming  $N_1 < \delta N_2$ , their inequality fails for  $\delta = 1/2$ ,  $N_2 = 5$ ,  $N_1 = 2$ , for example.

## A.5 Toeplitz matrices

A unit lower triangular Toeplitz matrix is a matrix of the form

$$P = \begin{bmatrix} 1 & & & & & \\ a_2 & 1 & & & & \\ \vdots & & \ddots & & & \\ a_{n-2} & \dots & 1 & & & \\ a_{n-1} & \dots & a_2 & 1 & & \\ a_n & \dots & a_3 & a_2 & 1 & \end{bmatrix}.$$

We similarly define unit upper triangular Toeplitz matrices.

One important property (see [KS91]) is that the space requirements are only  $O(n)$  elements and for a column tuple  $\mathbf{v}$ ,  $\mathbf{y} = P\mathbf{v}$  can be computed using polynomial multiplication with  $O(n \log(n) \log \log(n))$  ring operations, and the identity

$$(y_1 + y_2x + \dots + y_nx^{n-1}) \equiv (1 + a_2x + \dots + a_nx^{n-1})(v_1 + v_2x + \dots + v_nx^{n-1}) \pmod{x^n}.$$

## A.6 Finding the order of an element using discrete logarithms

We assume here that we have a black-box which given  $\alpha$  and  $\beta \in \langle \alpha \rangle$ , will return *an* integer  $k$  such that  $\alpha^k = \beta$ , not necessarily the smallest such integer, or even positive. The only assumption is that it takes polynomial time to do so, and thus that  $k$  also has size polynomial in the size of  $\alpha$  and  $\beta$ .

The set of all such solutions  $k$  to  $\alpha^k = \beta$ , is a coset modulo  $N$ . Start with  $i = 1$ . Select a random integer  $r \in \{1, 2, \dots, 2^i\}$ , for  $i = 1, 2, \dots$ , compute  $\alpha^r = \beta$ , and then solve for  $k = \log_\alpha(\beta)$ . Keep increasing  $i$  until we find that  $k$  does not correspond to  $r$ . Then we know that  $M = k - r$  is a non-trivial multiple of  $N$ . Continue picking random  $r \in \{0, 1, \dots, M\}$  solving for  $k = \log_\alpha(\alpha^r)$ , and then resetting  $M = \gcd(M, k - r)$ . If  $M = lN$ , for  $l > 1$ , then since we randomly select our exponent  $k$ , the probability that  $lN$  divides  $k - r$  is  $1/l \leq 1/2$ . Thus, if  $M > N$ , after each repetition  $M$  will decrease by a factor of 2 or more with probability at least  $1/2$ . After  $O(\log(1/\epsilon))$  repetitions with  $M$  not decreasing, the probability that  $M \neq N$  is less than  $\epsilon$  for any positive  $\epsilon$ .

This method is described in [McC90].

## A.7 The Number of Relatively Prime $(v_1(x), v_2(x))$ Pairs

Let  $P_{k_1, k_2}$  denote the number of relatively prime ordered pairs of monic non-zero polynomials  $A(x), B(x) \in GF(q)[x]$  of degrees at most  $k_1$  and  $k_2$  respectively. Without loss of generality, we can assume that  $k_1 \leq k_2$ .

There are precisely  $q^i$  monic polynomials of degree exactly  $i$ . Thus there are precisely  $(q^{i+1} - 1)/(q - 1)$  monic polynomials of degree at most  $i$  in  $GF(q)[x]$ .  $P_{k_1, k_2}$  satisfies the following recurrence:

$$\sum_{i=0}^{k_1-1} q^i P_{k_1-i, k_2-i} = \left( \frac{q^{k_1+1} - 1}{q - 1} \right) \left( \frac{q^{k_2+1} - 1}{q - 1} \right).$$

Solving this recurrence yields

$$P_{k_1+1, k_2+1} = \frac{q^{k_1+k_2+3} - 1}{q - 1}$$

for  $k_1$  and  $k_2$  any two non-negative integers.

# Bibliography

- [AD93] L. M. Adleman and J. DeMarrais. A subexponential algorithm for discrete logarithms over all finite fields. *Mathematics of Computation*, 61:1–15, 1993.
- [AdI94] L. M. Adleman. The function field sieve. In *Algorithmic Number Theory, Lecture Notes in Computer Science*, volume 877, pages 108–121. Springer-Verlag, 1994.
- [BBC<sup>+</sup>95] B. A. Barenco, C.H. Bennett, R. Cleve, D.P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Phys.Rev.A*, 52(5):3457–3467, 1995.
- [Ber70] E. R. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of Computation*, 24(111):713–735, 1970.
- [BFHMV84] I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone. Computing logarithms in finite fields of characteristic two. *SIAM Journal on Algebraic Discrete Methods*, 5(2):276–285, 1984.
- [BJMV84] I. Blake, M. Jimbo, R. C. Mullin, and S. A. Vanstone. Computational algorithms for certain shift register sequence problems. *Final Report, Project No. 307-16*, 1984. Prepared for Supply and Services, Canada.
- [BL95] Dan Boneh and Richard J. Lipton. Quantum cryptanalysis of hidden linear functions. In Don Coppersmith, editor, *Advances in Cryptology - Proceedings of Crypto '95, Lecture Notes in Computer Science*, pages 424–437. Springer-Verlag, 1995.
- [BP96] R. Lovorn Bender and C. Pomerance. Rigorous discrete logarithm computations in finite fields via smooth polynomials. *preprint*, 1996.
- [Cle94] Richard Cleve. A note on computing Fourier transforms by quantum programs. *unpublished paper*, 1994.
- [Cop84] D. Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Transactions on Information Theory*, IT-30(4):587–594, 1984.
- [COS86] D. Coppersmith, A. M. Odlyzko, and R. Schroepfel. Discrete logarithms in  $GF(p)$ . *Algorithmica*, 1:1–15, 1986.
- [CW94] D. Clark and L. Weng. Maximal and near-maximal shift register sequences: Efficient event counters and easy discrete logarithms. *IEEE Transactions on Computers*, 43(5):560–567, 1994.

- [CZ81] D. G. Cantor and H. Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(154):587–592, 1981.
- [DiV95] David P. DiVincenzo. Quantum computation. *Science*, 270:255–261, 1995.
- [EJ96] Artur Ekert and Richard Jozsa. Quantum computation and Shor’s factoring algorithm. *preprint*, 1996.
- [Eke96] Artur Ekert, Summer, 1996.
- [ElG85a] T. ElGamal. On computing logarithms over finite fields. In H. C. Williams, editor, *Advances in Cryptology - Proceedings of Crypto ’85, Lecture Notes in Computer Science*, volume 218. Springer-Verlag, 1985.
- [ElG85b] T. ElGamal. A subexponential algorithm for computing discrete logarithms over  $GF(p^2)$ . *preprint*, 1985.
- [GM93] D. S. Gordon and K. S. McCurley. Massively parallel computations of discrete logarithms. In R. A. Rueppel, editor, *Advances in Cryptology - Proceedings of Crypto ’92, Lecture Notes in Computer Science*, volume 740, pages 312–323. Springer-Verlag, 1993.
- [Gor93] D. M. Gordon. Discrete logarithms in  $GF(p)$  using the number field sieve. *SIAM Journal on Discrete Mathematics*, 6(1):124–138, 1993.
- [Hei92] R. Heiman. A note on discrete logarithms with special structure. In R. A. Rueppel, editor, *Advances in Cryptology - Proceedings of Eurocrypt ’92, Lecture Notes in Computer Science*, volume 658, pages 454–457. Springer-Verlag, 1992.
- [HR83] Martin E. Hellman and Justin M. Reyneri. Fast computation of discrete logarithms in  $GF(q)$ . In R. Rivest D. Chaum and A. Sherman, editors, *Advances in Cryptology - Proceedings of Crypto ’82*, pages 3–13. Plenum Press, 1983.
- [HW56] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. University Press, Oxford, 1956.
- [Joh84] D. S. Johnson. The NP-completeness column: An ongoing guide. *Journal of Algorithms*, 5:433–447, 1984.
- [Knu73] D. E. Knuth. *The Art of Computer Programming - Sorting and Searching*, volume 3. Addison-Wesley, 1973.
- [KS91] E. Kaltofen and B. D. Saunders. On wiedemann’s method of solving sparse linear systems. In *Springer Lecture Notes in Computer Science*, volume 539, pages 29–38, 1991.
- [LB92] S. Lloyd and J. Burns. Finding the position of a subarray in a pseudo-random array. *IEEE Transactions on Computers*, 5(2):237–253, 1992.
- [Len91] H. W. Lenstra Jr. Finding isomorphisms between finite fields. *Mathematics of Computation*, 56(193):329–347, 1991.

- [LN86] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and their Applications*. Cambridge University Press, Cambridge, 1986.
- [LO91a] B. A. LaMacchia and A. M. Odlyzko. Computation of discrete logarithms in prime fields. *Designs, Codes and Cryptography*, 1:47–62, 1991.
- [LO91b] B. A. LaMacchia and A. M. Odlyzko. Solving large sparse linear systems over finite fields. In A. J. Menezes and S. A. Vanstone, editors, *Advances in Cryptology - Crypto '90, Lecture Notes in Computer Science*, volume 537, pages 109–133. Springer-Verlag, 1991.
- [Lov92] R. Lovorn. *Rigorous, Subexponential Algorithms for Discrete Logarithms over Finite Fields*. PhD thesis, University of Georgia, Athens, Georgia, 1992.
- [LP92] H. W. Lenstra Jr. and Carl Pomerance. A rigorous time bound for factoring integers. *Journal of the American Mathematical Society*, 5(2):483–516, 1992.
- [McC90] K. S. McCurley. The discrete logarithm problem. In C. Pomerance, editor, *Cryptology and Computational Number Theory (Boulder, Colorado), Proceedings of Symposia in Applied Mathematics*, volume 42, pages 49–74, 1990.
- [Mil75] J. C. P. Miller. On factorisation, with a suggested new approach. *Mathematics of Computation*, 29(129):155–172, 1975.
- [Mos95] Michele Mosca. A simple attack on logarithms in Galois fields with orders that are smooth in the right places. 1995. unpublished report of findings during NSERC Undergraduate Research term with the Data Encryption Group.
- [MOV93] A. Menezes, T. Okamoto, and S. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39:1639–1646, 1993.
- [MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, 1995.
- [MvV96] A. Menezes, P. vanOorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [Nie91] H. Niederreiter. Recent advances in the theory of finite fields. In *Collection: Finite fields, coding theory, and advances in communications and computing (Las Vegas, NV), Lecture Notes in Pure and Applied Mathematics*, volume 141, pages 153–163, 1991.
- [Od185] A. M. Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Advances in Cryptology - Proceedings of Eurocrypt '96, Lecture Notes in Computer Science*, volume 209, pages 224–314. Springer-Verlag, 1985.
- [Od194] A. M. Odlyzko. Discrete logarithms and smooth polynomials. In *Finite Fields: Theory, Applications and Algorithms, Contemporary Mathematics*, volume 168, pages 269–278. American Mathematical Society, 1994.

- [PH78] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over  $GF(p)$  and its cryptographic significance. *IEEE Transactions on Information Theory*, IT-24(1):106–110, 1978.
- [Pol78] J. M. Pollard. Monte carlo methods for index computation (mod  $p$ ). *Mathematics of Computation*, 32:918–924, 1978.
- [Pom82] C. Pomerance. Analysis and comparison of some integer factoring algorithms. pages 89–139, 1982.
- [Pom85] C. Pomerance. The quadratic sieve factoring algorithm. In T. Beth, N. Cot, and I. Ingemarsson, editors, *Advances in Cryptology - Eurocrypt '84, Lecture Notes in Computer Science*, volume 209, pages 169–182, 1985.
- [Pom87] C. Pomerance. Fast, rigorous factorization and discrete logarithm algorithms. *Discrete Algorithms and Complexity*, pages 119–143, 1987.
- [RMK<sup>+</sup>80] R. L. Rivest, A. R. Meyer, D. J. Kleitman, K. Winklmann, and J. Spencer. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20:396–404, 1980.
- [RS85] J. A. Reeds and J. A. Sloane. Shift-register synthesis (modulo  $m$ ). *SIAM Journal on Computing*, 14(3), 1985.
- [Sch93] C. P. Schnorr. Factoring integers and computing discrete logarithms via diophantine approximation. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 13:171–181, 1993.
- [Sem91] I. A. Semaev. An algorithm for evaluation of discrete logarithms in some nonprime finite fields. *preprint*, 1991. according to [SWD96] to appear in *Math. Comp.*
- [Sem95] I. A. Semaev. Computation of discrete logarithms in an arbitrary field. *Discrete Mathematics and Applications*, 5(2):107–116, 1995.
- [Sho94a] Peter W. Shor. Algorithms for quantum computation: Discrete log and factoring. 1994. postscript file.
- [Sho94b] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. In *Los Alamos preprint archive (quant-ph/9508027 25 Jan 96)*, 1994. A preliminary version appeared in the Proceedings of the 35th Annual Symposium on Foundations of Computer Science.
- [Sou93] K. Soundararajan. Asymptotic formulae for the counting function of smooth polynomials. *preprint*, 1993. according to [SWD96] to appear in *J. London Math. Soc.*
- [SWD96] O. Schirokauer, D. Weber, and T. Denny. Discrete logarithms: the effectiveness of the index calculus method. In *Algorithmic Number Theory II, Lecture Notes in Computer Science*, volume to appear. Springer-Verlag, 1996.
- [Thi93] J. A. Thiong Ly. A serial version of the Pohlig-Hellman algorithm for computing discrete logarithms. *Applicable Algebra in Engineering, Communication and Computing*, 4:77–80, 1993.

- [Web96] Damian Weber. Computing discrete logarithms with the general number field sieve. In *Algorithmic Number Theory II (Bordeaux), pre-proceedings*, pages 377–389, 1996.
- [Wie86] D. H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Transactions on Information Theory*, IT-32(1):353–356, 1986.
- [WM68] A. E. Western and J. C. P. Miller. Tables of indices and primitive roots. 1968.
- [Zie74] N. Zierler. A conversion algorithm for logarithms on  $GF(2^n)$ . *Journal of Pure and Applied Algebra*, 4:353–356, 1974.
- [Zuc95] R. J. Zuccherato. personal communication, 1995.